UNIVERSIDAD NACIONAL DEL SUR

Tesis de Magíster
en Ciencias de la Computación

Reutilización de Software
basado en Servicios Web

Cristian Damián Pacifico

Bahía Blanca Argentina

2009

UNIVERSIDAD NACIONAL DEL SUR

Tesis de Magíster
en Ciencias de la Computación

Reutilización de Software
basado en Servicios Web

Cristian Damián Pacifico

Bahía Blanca Argentina

2009

Prefacio

Esta Tesis se presenta como parte de los requisitos para optar al grado académico de Magíster en Ciencias de la Computación, de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Ciencias e Ingeniería de la Computación durante el período comprendido entre el mes de agosto de 2001 y el mes de junio de 2009, bajo la dirección del Dr. Pablo Rubén Fillottrani, Profesor Adjunto del Departamento de Ciencias e Ingenierías de la Computación.

Cristian Damian Pacifico
cripac@ai.fcad.uner.edu.ar
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, 12 de junio de 2009

UNIVERSIDAD NACIONAL DEL SUR
Secretaria General de Postgrado y Educación Continua

La presente tesis ha sido aprobada el: ...../...../.....,

mereciendo la calificación de .....(.....).

Resumen

En esta Tesis se desarrolla el tema de la reusabilidad de software dentro del paradigma de Servicios Web. La principal motivación para ello es analizar el profundo impacto que tienen ambos conceptos en la Industria Informática e Ingeniería de Software. Entender y descubrir si los Servicios Web son el remedio para la enfermedad que inhibe a la Reusabilidad de Software es un problema todavía abierto en las Ciencias de la Computación. Se utiliza la temática relacionada con la integración de aplicaciones como eje conductor para explicar los conceptos de Servicios Web y Reusabilidad. Esta opción se fundamenta en comprender que los requerimientos de la integración concuerdan con los de reuso.

Luego de desarrollar los conceptos de la tesis se puede concluir que existe una vinculación natural entre la Reusabilidad y los Servicios Web, especialmente potenciada por la integración de aplicaciones, de datos y de conocimiento. Se entiende que los Servicios Web y la Orientación a Servicios son fuertes paliativos para la falta de reusabilidad. Sin embargo la adopción de cada uno requiere un gran esfuerzo, que puede involucrar a transformaciones significativa en la infraestructura informática de una empresa.

No se puede llegar a una conclusión definitiva respecto si los Servicios Web y la Orientación a Servicio son la "cura" a la falta de reusabilidad. Por un lado, es necesario que determinadas tecnologías maduren, como los Servicios Web Semánticos, para valorar su éxito en la automatización del reuso. Por otro lado, no se pude hablar de "cura" definitiva, ya que la complejidad de la reusabilidad, como se explicó, va generando nuevos requisitos y escenarios.

En síntesis, los Servicios Web, y en especial su aporte a la integración e interoperatividad, favorecieron a brindar requerimientos realistas y soluciones viables a cuestiones sobre reuso. Ambos conceptos presentan una sinergia propia. A la luz de lo investigado, los Servicios Web y la Orientación a Servicios resultan tener altas probabilidades de éxito para la adopción y práctica de la Reusabilidad de Software.

Summary

In this Thesis the author intenteds to study in deep software reusability within the Web Services' paradigm. The main motivation in the subject's choice has to do with the deep impact that has both concepts in the Computer Science and Software Engineering. It is motivating to understand and to discover if the Web Services are the remedy for the disease that inhibits to reusability. Application Integration is the thematic used to explain the concepts of Web Services and Reusability. This option is based on understanding that the requirements of integration agree with those of reuse.

After to developing all concepts of the Thesis, it can be concluded that a natural entailment between Reusability and Web Services exists, especially harnessed by the applications integration, data and knowledge. It is understood that the Web Services and Service Oriented Architectures are strong palliatives for the lack of reusabilidad. Nevertheless the adoption of each requires a concerted effort, that can involve big transformations in the computer science infrastructure of a company.

It's not a definitive conclusion respect to whether Web Services and Service Oriented Architectures are both a cure to the lack of reuse. On the one hand, it is necessary that determined technologies mature, like the Semantic Web Services, to evaluate his success in reuse automatization. Besides, there isn't definitive cure, since the reusability complexity, is generating new requirements and scenes.

Web Services, and especially their contribution to integration and interoperativity, has really favored to offer to realistic requirements and viable solutions to reuse. Both concepts present synergy. Hence, Web Services and Service Oriented Architectures turn out to have good chances of success for Software Reusability adoption and practice.

Índice general

Capítulo 1

Introducción

1.1. Motivación

El presente trabajo es una Tesis que se presenta en el **Departamento de Ciencias e Ingeniería** de la **Universidad Nacional del Sur** para completar los requerimientos académicos del Título de Magister en Ciencias de la Computación.

En el título de la misma ***"Reutilización de Software basado en Servicios Web"*** se muestra la intención del autor de profundizar el estudio de reutilización y reuso de software dentro del paradigma tecnológico propuesto por los Servicios Web. La motivación principal en la elección del tema tiene que ver con el profundo impacto que tienen ambos conceptos en la Industria Informática e Ingeniería de Software.

Reusabilidad. Por un lado, la **problemática de la reutilización**, que tiene sus orígenes en la propia génesis de la Industria del Software, plantea desafíos tendientes a lograr los niveles de productividad y calidad del software adecuados para la demanda de sistemas informáticos que se incrementa exponencialmente desde hace ya varias décadas. La Reusabilidad como calidad en el software, o mejor dicho la falta de ésta, es una de las patologías crónicas de la Industria de Software. Existe una clara conciencia universal sobre los beneficios que la Reusabilidad trae, sin embargo no se ha podido establecer adecuadamente, a los largo de cuatro décadas, una forma de llevarla a cabo.

Se ha puesto de manifiesto a lo largo de los años, algunas ideas fundamentales respecto a la reutilización de software [422]. En primer lugar se reconoce que los principios, métodos y habilidades requeridas para desarrollar software reutilizable no pueden ser aprendidas por generalidades y obviedades. En cambio, los desarrolladores deben aprender habilidades técnicas concretas y ganar experiencia real por medio del desarrollo y uso de componentes de software y marcos de trabajos orientados al reuso en su práctica profesional diaria.

Para obtener un éxito a largo plazo, los esfuerzos orientados a la reutilización deben realizarse teniendo en cuenta las características técnicas y no-técnicas. Si se enfoca solamente en las cuestiones no-técnicas del reuso se descuida la importancia de la formación en la construcción orientada al reuso, importantísima cuando se decide comenzar un proyecto no trivial. Por otro lado, atender solamente a las cuestiones técnicas subestima el profundo impacto que tiene el éxito o el fracaso en la adopción de nuevas tecnologías sobre las fuerzas económicas y procesos empresariales.

Dentro de los requisitos y condiciones esenciales para la práctica del reuso se pueden citar:

- *Soporte gerencial al reuso*: las ventajas competitivas que ofrecen el reuso, exige a toda organización que planifique practicar reusabilidad de software adoptar políticas que definan el marco de desarrollo de esta actividad. Es necesario entonces contar con un plan sistemático de reusabilidad, cuyo alcance se extienda por todos lo niveles de la organización, ya que la adopción del reuso tiene un fuerte impacto en la cultura de producción de software y la

valuación de los sistemas informáticos. En menor o mayor grado se exige una reingeniería en las estructuras organizativas de producción de software.

- *Grupo de profesionales de soporte para reusabilidad*: el grupo humano que llevará a cabo la práctica del reuso de software, será el encargado de crear, adquirir, certificar, clasificar y gerenciar los artefactos reusables. Los roles pueden ser muy diversos, y abarcan de especificadores de componentes de negocios hasta ensambladores de aplicaciones.
- *Artefactos reusables*: su adecuada gestión y uso son en gran medida el fundamento del éxito de las prácticas de reusabilidad. Un repositorio de artefactos orientado al reuso debe contar con un mecanismos para organizar, almacenar, indexar y recuperar artefactos reusables. Sus mecanismos deben ser, en gran medida, automatizados para que asistan efectivamente a la creación de aplicaciones a partir de reuso.
- *Metodologías orientadas al reuso*: según el grado de adopción del reuso elegido, se requiere que se modifique los métodos de construcción de software tradicionales para que incluyan tecnologías y actividades relacionadas con reusabilidad de software; o bien seguir nuevas metodologías orientadas reuso como objetivo primordial. Las metodologías deben proveer de métodos para: (1) guiar a los constructores de artefactos a identificar oportunidades de reuso e indicar cómo debe construirse un componente, y (2) guiar a los ensambladores de aplicaciones en cómo localizar y recuperar artefactos reusables e integrarlos en una nueva aplicación.

Servicios Web. Por otro lado, se reconoce a los Servicios Web como paradigma tecnológico orientado a la producción de aplicaciones distribuidas y dirigido esencialmente a la integración de funcionalidades y capacidades de software, allanando el camino impuesto por la heterogeneidad. Los Servicios Web buscan la interoperatividad a gran escala, entre aplicaciones con diferentes lenguajes de desarrollo, distintas plataformas de implementación y diferentes empresas administradoras. En dicha búsqueda fue necesaria la adopción de estándares tecnológicos para la descripción, vinculación y registros de software; generando el marco de trabajo propicio para hacer frente a las barreras tecnológicas de la reusabilidad.

Vinculando ambos conceptos, es motivador entender y descubrir si los Servicios Web son el remedio para la enfermedad que inhibe a la Reusabilidad. Si los desafíos históricos impuestos por la necesidad de reuso puede ser afrontados por la adopción de un paradigma tecnológico relativamente nuevo y revolucionario. Si los principios e ideas de los Servicios Web pueden refinar y darle una aproximación realista a lo requerimientos de la práctica de la reutilización de software. Si la adopción de las tecnologías de Servicios Web y su marco teórico de Orientación a Servicios son un verdadero frente con probabilidades de éxito en la lucha de la Reusabilidad; o si serán derrotados como lo fueron otras tecnologías previas.

Integración como eje conductor. Con el propósito de darle profundidad a los conceptos estudiados para la Tesis, se utiliza la temática relacionada con la integración de aplicaciones distribuidas como eje conductor para explicar los conceptos de Servicios Web y Reusabilidad. Esta opción tiene su fundamento en la suposición que los desafíos impuestos por la integración de aplicaciones concuerdan en gran medida con los requerimintos de la Reusabilidad, sin embargo brinda problemáticas y escenarios más concretos e interesantes para su resolución mediante Servicios Web.

Para entender la problemática de integración se puede decir que a pesar que el poder de las computadoras y la disponibilidad de las redes se ha incrementado dramáticamente en los últimos años, el diseño y la implementación de aplicaciones distribuidas ha sido oneroso y propenso al error. Mucho de este costo y esfuerzo proviene de la continua reinvensión y redescubrimientos de los patrones principales y los componentes fundamentales en la construcción de software [422].

El objetivo principal es construir aplicaciones distribuidas con las siguientes cualidades:

- *Portabilidad*, para reducir el esfuerzo para soportar aplicaciones en plataformas, sistemas operativos, lenguajes de programación y compiladores heterogéneos.

- *Flexibilidad*, para soportar un creciente número de formatos y tipos de datos multimediales, y extensiones a los requerimientos de calidad de servicios (*QoS: quality of services*)

- *Extensibilidad*, para soportar sucesiones de actualizaciones rápidas y agregados para tomar ventaja de nuevos requerimientos y mercados emergentes

- *Predictibilidad* y *eficiencia* para proveer baja latencia para aplicaciones sensibles en tiempo real y alto desempeño en aplicaciones de uso intensivo de ancho de red

- *Fiabilidad*, para asegurarse que las aplicaciones sean robustas, tolerantes a fallas y altamente disponibles.

Si se tiene en cuenta estas cualidades y entendiendo que existen múltiples plataformas de hardware, diversos sistemas operativos y software de red, construir aplicaciones distribuidas desde cero comienza a ser irrealizable. Desarrollar software que cumpla estas cualidades es difícil; desarrollar componentes y marcos de trabajo de software de alta calidad que sean reutilizables es más difícil aún [182, 183]. Los componentes reutilizables y los marcos de trabajo son inherentemente abstractos, lo cual hace complicado diseñar y especificar su calidad y manejar su producción. Más aún, las aptitudes necesarias para desarrollar, poner en marcha y soportar software reutilizable ha sido tradicionalmente un "arte místico" cautivo en las mentes de desarrolladores expertos. Cuando estos impedimentos técnicos del reuso se mezclan con impedimentos no-técnicos, ya sean organizacionales, económicos, políticos, administrativos y sicológicos, lograr significativos niveles de reutilización de software en una organización se vuelve una tarea difícil de realizar.

Es más fácil y más redituable desarrollar y construir aplicaciones distribuidas sobre la base de sistemas distribuidos reutilizables de middleware. El middleware reside entre las aplicaciones y el sistema operativo, la pila de protocolos de red y el hardware subyacentes. El rol del middleware es funcionar como puente en la brecha que existe entre los programas y el hardware y la infraestructura de software de bajo nivel con el propósito de coordinar cómo las partes de la aplicación se conectan e interoperan, y permitir y simplificar la integración de componentes desarrollados por múltiples proveedores de tecnologías.

1.2. Objetivos

Los objetivos de la presente tesis son:

- Comprender la problemática de la reusabilidad y sus efectos concretos en el desarrollo de software.

- Comprender la problemática de la integración de aplicaciones distribuidas como escenario elegido para visualizar las necesidades de reuso de software.

- Comprender los principios sostenidos por las tecnologías de los Servicios Web y los efectos de su adopción en la construcción de sistemas informáticos distribuidos.

- Relacionar los conceptos anteriores, describiendo escenarios concretos de vinculación, comprendiendo las problemáticas particulares de Reusabilidad e Integración y su solución a través de la tecnologías de Servicios Web.

1.3. Estructura de Tesis

Tesis se estructura en los siguientes capítulos:

- ***Capítulo 1: Introducción***. Es el presente capítulo, en donde se explica la motivación, objetivos y estructura de la Tesis.
- ***Capítulo 2: Sistemas Informáticos Distribuidos***. En el capítulo se presenta el modelo de sistema distribuido, sus principios de diseño, su diferentes arquitecturas y las característica de la comunicación en un ambiente distribuido. Este capítulo se introduce la problemática de integración y su vinculación con el reuso.
- ***Capítulo 3: Middleware***. En este capítulo se tratan las diferentes soluciones para manejar la distribución y heterogeneidad en aplicaciones distribuidas. Se introduce el concepto de middleware como componente esencial para manejar la integración y el reuso. Se realiza un detalle de las diferentes soluciones de middleware que surgieron en la historia y la problemática que solucionó cada una.
- ***Capítulo 4: Tecnologías de la Web***. Es el capítulo en donde se muestra las características del ambiente Web, estableciéndolo como el nuevo escenario de la integración de aplicaciones. También se hace un tratamiento de las soluciones de integración y reuso que usan a la Web como medio de exposición. Estas soluciones son las precursoras de los Servicios Web.
- ***Capítulo 5: Servicios Web y sus Tecnologías Fundamentales***. Se presenta a la tecnología de los Servicios Web: XML para la descripción de la información, SOAP para acceso, WSDL para descripciones y UDDI para registro de servicios web. Se la muestra a los Servicios Web como el siguiente paso en tecnologías de integración de aplicaciones.
- ***Capítulo 6: Segunda Generación de Tecnologías Servicios Web***. En esta capítulo se pone énfasis en la composición y coordinación en Servicios Web. Ya resuelta la problemática de integración, es necesario establecer una mirada a la interacción entre servicios como un escenario y medio concreto de reuso. Se estudian las tecnología estándares para la coordinación de servicios:WS-Coordination, y se presentan a la orquestación y coreografía de servicios como un marco conceptual para la composición (WS-BPEL y WS-CDL respectivamente).
- ***Capítulo 7: Servicios Web y Reusabilidad***. Este capítulo está orientado a estudiar de lleno la vinculación de ambos conceptos, a la luz de lo desarrollado en los anteriores. Se presenta la problemática de la Reusabilidad de Software y se describe las soluciones aportadas para su adopción y práctica por el marco teórico de la Orientación a Servicios. Además se muestran tres escenarios que vinculan los conceptos: la modernización de aplicaciones legadas a través de Servicios Web, la reutilización de ontologías de Web Semántica y el reuso en los Servicios Web Semánticos.
- ***Capítulo 8: Conclusiones***. Se presentan conclusiones del presente trabajo, los resultados obtenidos y se delinean propuestas para futuros trabajos relacionados.

Capítulo 2

Sistemas Informáticos Distribuidos

Los ***Servicios Web*** *o* **WS** *(Web Services)* son una forma de sistemas informáticos distribuidos o aplicaciones distribuidas. Para entender cuáles problemáticas resuelven y cuáles fueron las restricciones de diseño encontradas, se requiere necesariamente saber cómo han evolucionado los sistemas informáticos distribuidos. Una cuestión a tener en cuenta es que, a pesar de que la tecnología ha cambiado, los problemas a resolver son en mayor o menor grado los mismos. Por eso, es requisito indispensable para entender a los Servicios Web desde esta perspectiva, tener un acabado conocimiento de los sistemas informáticos distribuidos [10].

Gran cantidad de definiciones han surgido hasta el presente como productor de innumerables tratados y estudios. Una de las definiciones más amplias considera que un ***sistema informático distribuido*** o ***aplicación distribuida*** es un conjunto de programas, ejecutándose en una o más computadoras, los cuales coordinan sus acciones mediante el intercambio de mensajes [62].

Con el objeto de evitar ambigüedades cabe aclarar las diferencias que existen entre los conceptos de "sistema informático distribuido" y "red de computadoras". Una ***red de computadoras*** es una colección de computadoras interconectadas por hardware el cual soporta directamente el pasaje de mensajes [62]. En una red, los usuarios son conscientes de que existen varias máquinas interconectadas en las cuales la ubicación, el balanceo de carga, las replicaciones de almacenamiento y las funcionalidades que presenta cada una, no son transparentes [480]. La mayoría de los sistemas distribuidos operan sobre redes, sin embargo se pueden construir aplicaciones distribuidas cuyos componentes residan en una misma computadora multitarea, y también, pueden existir sistemas distribuidos en los cuales los flujos de información entre componentes se establecen por otros medios diferentes al intercambio de mensajes [62].

Sin embargo el escenario más común de implementación de un sistema informático distribuido son las redes de computadoras y el intercambio de mensajes.

La génesis de los sistemas distribuidos fue motivada por la reducción de costos, la necesidad de alta disponibilidad y de integración de aplicaciones de diferente naturaleza o de diferentes organizaciones. Un sistema distribuido se puede presentar como oposición a un sistema centralizado, el cual consiste en una sola computadora, con uno o múltiples procesadores, responsable del procesamiento de todas las peticiones requeridas [482]. Por ende, una definición más empírica puede ser dada: un sistema distribuido está compuesto por múltiples computadoras autónomas que coordinan sus actividades y comparten recursos gracias al intercambio de mensajes sobre una red de computadoras [33, 66].

Esta cooperación se logra con algún software de distribución [110], el cual permite a los componentes del sistema coordinarse y compartir sus recursos de hardware, software y fuentes de datos. Un software de sistema distribuido bien desarrollado provee la ilusión de un ambiente único e integrado el cual es implementado por múltiples computadoras en diferentes ubicaciones [480, 481]. En otras palabras, el software le da transparencia de distribución al sistema [482].

Algunos autores [62] denominan a este software de distribución: "protocolo". El término ***protocolo*** hace referencia al algoritmo que gobierna el intercambio de mensajes, mediante el cual es

posible que una colección de procesos puedan coordinarse y comunicar información entre ellos. Si un programa es un conjunto de instrucciones, y un proceso denota la ejecución de tales instrucciones, un protocolo es un conjunto de instrucciones que gobiernan la comunicación en un programa o aplicación distribuida. Por consiguiente, un sistema distribuido es el resultado de ejecutar alguna colección de tales protocolos para coordinar las acciones de una colección de procesos en una red.

Todas las definiciones muestran, de una u otra manera las características esenciales de los sistemas distribuidos. Si un sistema distribuido está formado por varios componentes o programas, estos componentes son concurrentes e independientes entre sí. Si estos componentes residen en diferentes máquinas autónomas queda de manifiesto la ausencia de un reloj global y la independencia de fallas de los componentes [110].

El objetivo del capítulo es introducir los conceptos básicos de las aplicaciones distribuidas: sus diseños, arquitecturas y patrones de comunicación. Los conceptos son presentados en forma teórica, buscando no tener que detallar las características propias de cada implementación o producto existente. El objetivo es establecer determinadas guías de diseño y conceptos para luego poder contrastar los Servicios Web contra los sistemas tradicionales.

El capítulo comienza abarcando los conceptos relacionados con el diseño de sistemas informáticos o aplicaciones. Se discuten los diferentes niveles involucrados y cómo estos pueden ser diseñados en forma top-down o bottom-up. Luego se describen las diferentes arquitecturas de los sistemas informáticos, siguiendo una perspectiva histórica. Se intentará discutir el origen de cada arquitectura, qué ventajas aportó y qué debilidades presentó en la evolución de los sistemas informáticos distribuidos. Por último, se termina con una discusión de las diferencias entre la interacción asincrónica y sincrónica, como así también las consecuencias de usar cada una.

2.1. Diseño de un Sistema Informático

Un sistema informático distribuido es implementado por una máquina de software con alto grado de complejidad y diversidad. Con el propósito de mostrar las características fundamentales de su diseño, se lo explicará como una interrelación de tres niveles conceptuales. Un ***nivel conceptual*** o ***capa de abstracción*** *(abstraction layer, abstraction level)* es una forma de ocultar los detalles de implementación de un conjunto particular de funcionalidades [479].

2.1.1. Niveles de un Sistema Informático

En un nivel conceptual, un sistema informático esta diseñado en tres *niveles (layers)* [192, 408, 116] (véase Fig. 2.1):

- el ***Nivel de Presentación*** *(Presentation Layer)*
- el ***Nivel de Lógica de Aplicación*** *(Application Logic Layer)*
- el ***Nivel de Gestión de Recursos*** *(Resource Management Layer)*

Nivel de Presentación

Al ***Nivel de Presentación***, o simplemente *presentación*, lo conforman todos aquellos componentes que soportan las tareas de comunicación entre el sistema informático y las entidades externas del sistema informático, sean estas personas u otras aplicaciones. Se puede implementar de formas muy variadas, dependiendo de las necesidades; por ejemplo, puede ser una interfaz gráfica para usuarios; o puede ser un módulo que formatea un conjunto de datos en una determinada representación sintáctica, por ejemplo en HTML (véase ejemplo más abajo). En ocasiones se denomina a este nivel *cliente* [110], lo que no es del todo correcto. Un ***Cliente*** es una entidad

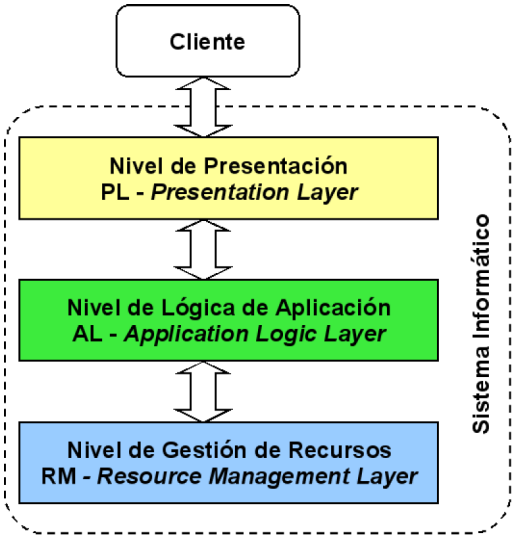

Figura 2.1: Diferentes niveles de un sistema informático

que consume o provee información al sistema informático. En ocasiones el cliente está separado completamente y es una entidad externa al sistema, como ser un navegador web. En otros casos el Nivel de Presentación y el Cliente se fusionan en uno, como es el caso de los Java Applets [475].

Por ejemplo: conforman el Nivel de Aplicación de una aplicación web los interpretes PHP o ASP agregados a un servidor web los cuáles toman el código embebido en una página web residente en el servidor y producen un documento HTML. El cliente de esta aplicación web es el navegador web que consume los documentos HTML producidos [277]. Un ejemplo similar es el entorno de ejecución que propone Java del lado del servidor web el soporte de páginas web dinámicas: los servlets [457].

Por otro lado, un Applet de Java es un programa escrito en el lenguaje JAVA que puede ser incluido en una página HTML como un elemento externo (al igual que una imagen). Esta aplicación sólo puede ejecutarse en un navegador web que tenga asociada la Máquina Virtual Java (Java Virtual Machine o JVM). En este caso el nivel de presentación lo provee la JVM que está agregado al navegador web, que es el cliente [406].

Nivel de Lógica de Aplicación

En la mayoría de los sistemas informáticos existe cierto procesamiento de datos detrás de los resultados que entregan. Dicho procesamiento involucra a todos los programas que implementan las operaciones requeridas por el Cliente a través del Nivel de Presentación. Al grupo de estos programas se los conoce como ***Nivel de Lógica de Aplicación***, o simplemente *lógica de aplicación*. Es común referirse a este nivel como los *servicios* ofrecidos por el sistema informático. Es decir que describen e implementan la lógica de un servicio desde el punto de vista del proveedor del mismo. Dependiendo de la complejidad en la lógica y en la técnica de implementación, a este nivel también se le suele llamar *Procesos de Negocios* [408], *Lógica de Negocios* [226, 305], *Reglas de Negocios* [310] o simplemente *Servidor* [110].

Nivel de Gestión de Recursos

Los sistemas informáticos necesitan datos para procesar y estos datos deben estar almacenados en algún lugar. Los datos pueden residir en bases de datos, sistemas de archivos, un directorio LDAP, u otro tipo de repositorio. El ***Nivel de Gestión de Recursos*** abarca tales elementos del sistema informático. En una forma más abstracta [408], este nivel, trata e implementa las

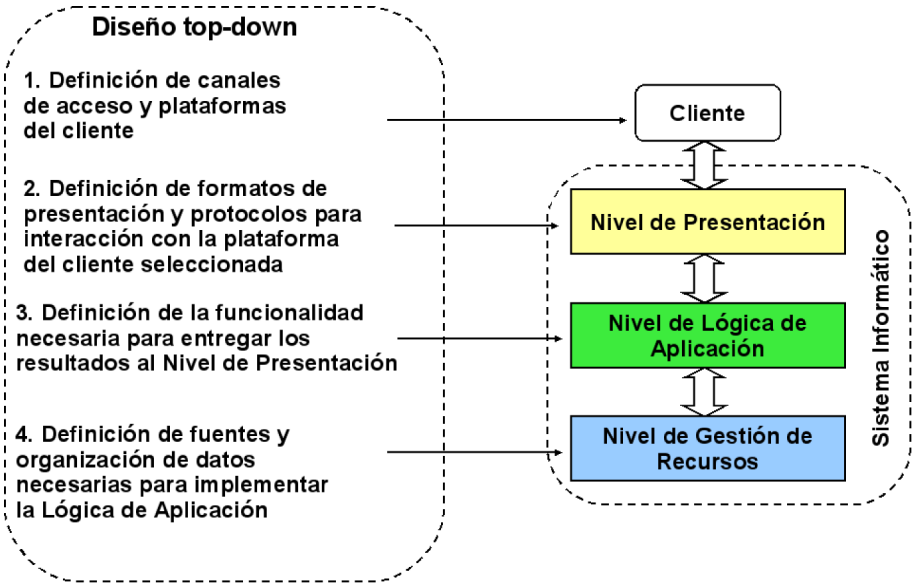

Figura 2.2: Etapas del diseño top-down

diferentes fuentes de datos independientemente de sus naturalezas y arquitecturas.

En una perspectiva más restringida [4], el nivel es llamado *Nivel de Datos* o *Servicio de Datos* para indicar que ha sido implementado con sistemas de gestión de bases de datos o DBMS *(database management systems)*. Desde esta óptica, este nivel tiene a cargo los servicios de persistencia requeridos para la aplicación y los mecanismos de bajo nivel específicos de acceso a bases de datos y de soporte a SQL.

En una perspectiva más amplia [305], el Nivel de Gestión de Recursos, puede incluir sistemas externos que proveen información presentados en la forma de servicios. En estas aplicaciones externas también es posible identificar los tres niveles conceptuales. Esto muestra que un sistema informático se puede constituir en forma recursiva usando composición de otros sistemas de información. Conforman también la Gestión de Recursos los adaptadores, agentes, componentes y librerías que vinculan al sistema de referencia con sistemas externos y repositorios de datos.

2.1.2. Diseño top-down de un Sistema Informático

El ***diseño top-down*** se enfoca en definir primero los objetivos superiores del problema y así proceder a definir todo lo necesario para alcanzar estos objetivos [538, 189, 478]. La idea subyacente de esta técnica es definir la funcionalidad del sistema desde el punto de vista de los clientes y en la forma que estos interactuarán con el sistema. Se puede decir que el diseño está orientado completamente por la funcionalidad que el sistema ofrecerá cuando esté operativo (véase Fig. 2.2).

Una vez que los objetivos superiores del sistema están definidos, comenzará el diseño de la lógica de la aplicación necesaria para proveer dicha funcionalidad. Por último, se definen los recursos necesarios que insumirá esa lógica de aplicación.

Como parte de este proceso de diseño, es fundamental determinar de qué forma el sistema informático será distribuido y en cuántos diferentes nodos. La funcionalidad que se distribuye puede ser de cualquiera de los tres niveles abstractos. Estas decisiones de diseño respecto a la distribución de las funcionalidades, es influenciada por el desarrollo de las ITs que tiene la organización en el dominio de la aplicación y por los sistemas informáticos preexistentes al nuevo sistema.

Por ejemplo, se puede suponer que no se cuenta con estructura informática desarrollada ni aplicaciones conexas existentes; es decir que se deben realizar adquisiciones de infraestructura

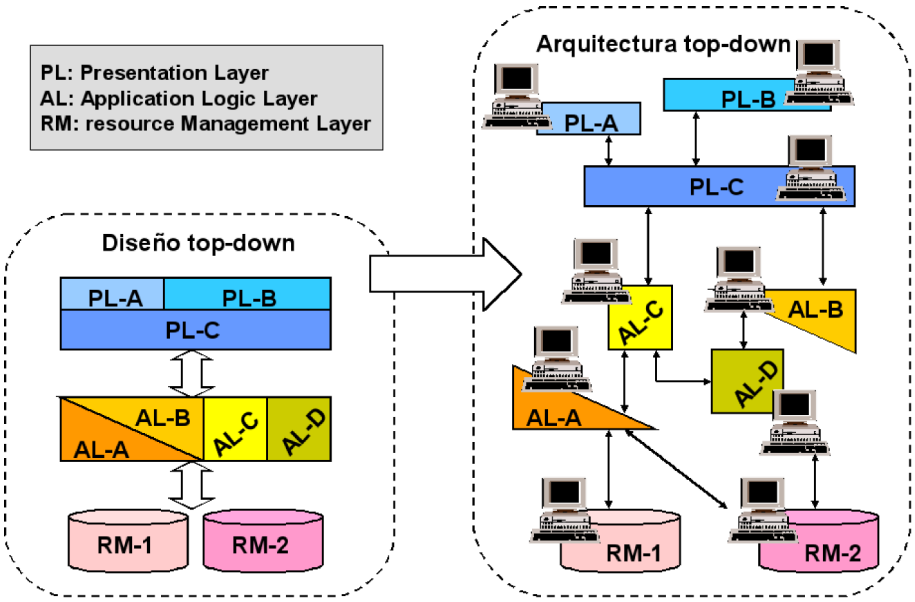

Figura 2.3: Arquitectura de un sistema diseñado por top-down

para soportar el nuevo sistema y el nuevo sistema debe desarrollarse desde cero. En este escenario, es lógico optar por un diseño top-down que prefiera construir un sistema que se ejecute en un entorno de computación homogéneo (por ejemplo PC corriendo en Linux). Si no se tienen políticas de diseño orientado a reuso apropiadas, la situación anterior genera que, a los efectos de simplificar el desarrollo y el mantenimiento, se construyan componentes *estrechamente* o *fuertemente acoplados*, lo que significa que la funcionalidad de un componente depende fuertemente de la funcionalidad implementada por otros componentes. Es decir que los componentes no pueden usarse en forma independiente (stand-alone) fuera del sistema.

El diseño top-down tiene considerables ventajas. En especial, enfatiza la definición de los objetivos finales del sistema y en la forma en que éstos pueden ser alcanzados consiguiendo los requerimientos funcionales (las operaciones que el sistema debe soportar) y no-funcionales (el rendimiento y disponibilidad). Sin embargo, su principal desventaja es que sólo es adecuado su aplicación en la construcción de sistemas desde cero; su filosofía no encuadra la posibilidad de reuso de sistemas ya construidos. Un cambio en los objetivos o requerimientos del sistema generaría una reconstrucción del sistema actual desde sus cimientos.

2.1.3. Diseño bottom-up de un Sistema Informático

El ***diseño bottom-up***, o más propiamente dicho: la ***integración bottom-up*** [396, 447], comienza con la construcción y el testeo de los módulos de más bajo nivel en la estructura de la aplicación. Estos componentes atómicos son integrados desde abajo hacia arriba en el nivel de abstracción, hasta cumplimentar los objetivos y requerimientos de la aplicación final.

En un enfoque construcción de software [295], un proceso bottom-up consiste en construir soluciones robustas y extendibles, las cuales se combinan en ensambles sucesivos, hasta que un ensamble final alcanza la solución al problema original. Cada solución parcial combinada o componente producido, será un subproducto aplicable a ser solución final para futuros problemas.

En la problemática de la integración de aplicaciones, el ***diseño bottom-up*** surge más por necesidad, antes que por opción ingenieril. Luego de haber desarrollados sus primeros sistemas informáticos, posiblemente usando el enfoque top-down, las organizaciones vieron la posibilidad de integrar la información manejada por varios sistemas y tratarlas en forma integral. Surge entonces la necesidad de las organizaciones de construir nuevos sistemas a partir de los sistemas existentes, usualmente llamados *sistemas heredados o legados*. Se puede considerar que un sistema

llega a ser *"heredado"* o *"legado"* cuando es usado para un propósito o en un contexto diferente al original para el que fue creado.

Un ***sistema informático legado*** o ***heredado*** *(legacy system)* puede ser definido como "cualquier sistema informático que resiste las modificaciones y la evolución" [82]. Los sistemas legados son típicamente parte de la columna vertebral del flujo de información dentro de una organización. Una falla en los mismos puede causar un serio impacto en la empresa [48]. Estos sistemas legados, siendo la mayoría de misión crítica, plantean serios problemas a las organizaciones que los mantienen. Los problemas más destacados que se presentan son [64]:

- estos sistemas se soportan en hardware obsoleto el cual es lento y caro de mantener;
- el mantenimiento de los sistemas legados es generalmente costoso, es dificultoso localizar fallas debido a la falta de documentación, al desconocimiento general de lo que el sistema hace internamente y a la falta de soporte técnico [527];
- los esfuerzos de integración son desalentados en tales sistemas debido a la ausencia de interfaces claras;
- y los sistemas legados son difíciles, sino imposible, de expandir.

Sin embargo, en determinados casos es necesario mantener tales aplicaciones por cuestiones prácticas y económicas. En este caso, el problema fundamental es la integración de sistemas legados en una aplicación coherente. La funcionalidad de las aplicaciones legadas está predefinida, y no es usual que pueda ser cambiada. Por esta razón, la tarea de construcción de aplicaciones a partir de sistemas legados no puede ser abordada siguiendo un enfoque top-down, ya que no existe la libertad de seleccionar y modelar las funciones de los sistemas subyacentes. De la misma forma, re-implementar tales sistemas, basándose en un método top-down, de forma tal que se adopten a los nuevos requerimientos, no sería económicamente viable.

En tales casos, cuando se involucran aplicaciones legadas, el diseño del nuevo sistema está muy condicionado con las características de los niveles conceptuales más bajos. El esfuerzo intelectual de diseño se centra en las formas y metodologías de adaptación de los sistemas heredados para conformar un nuevo sistema integrado con nuevos objetivos. Es por eso que el diseño Botton-up, luego de definir los objetivos fundamentales y los canales de interacción con los clientes del sistema, se debe proceder a analizar el Nivel de Gestión de Recursos, en el cual se encuentran los sistemas heredados (véase Fig. 2.4).

En este punto se realiza una valoración de los costos y viabilidad de la obtención de los recursos necesarios para la nueva funcionalidad requerida a partir de los componentes heredados. Tales componentes se "enmascaran" o se "envuelven" para que ofrezcan una interfaz adecuada para ser brindada al Nivel de Lógica de Aplicación (véase Fig. 2.5). Sólo después de diseñar estas *"envolturas" (wrappers)* es posible definir la lógica de aplicación y la presentación del nuevo sistema. De hecho la metodología bottom-up, requiere que el equipo de diseño comience realizando un estudio previo de las aplicaciones existentes y de sus oportunidades de reutilización, seguido de un análisis y reestructuración del dominio del problema hasta que se esclarece cuáles de los objetivos de alto nivel serán alcanzados.

Por diseño, las arquitecturas bottom-up conllevan a sistemas con *bajo grado de acoplamiento*, donde los componentes pueden ser usados como sistemas independientes. De hecho uno de los principales desafíos, es establecer de qué forma un sistema legado puede usarse como componente, sin perder su funcionalidad como sistema *stand-alone*.

No tiene sentido analizar ventajas y desventajas al discutir bottom-up. En muchos casos no hay otra opción. Generalmente esta estrategia de diseño es recomendada en los casos de integración de sistemas legados.

En un amplio contexto, una de las principales ventajas del uso de las tecnologías basadas en Servicios Web es la habilidad para hacer modelos bottom-up en una forma más eficiente, económica y simple para diseñar y mantener.

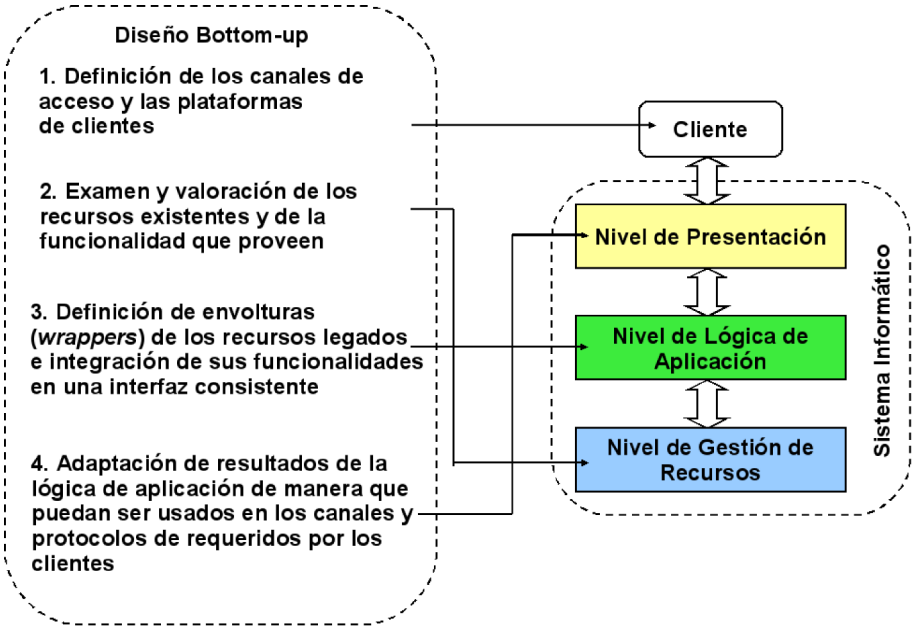

Figura 2.4: Etapas del diseño bottom-up

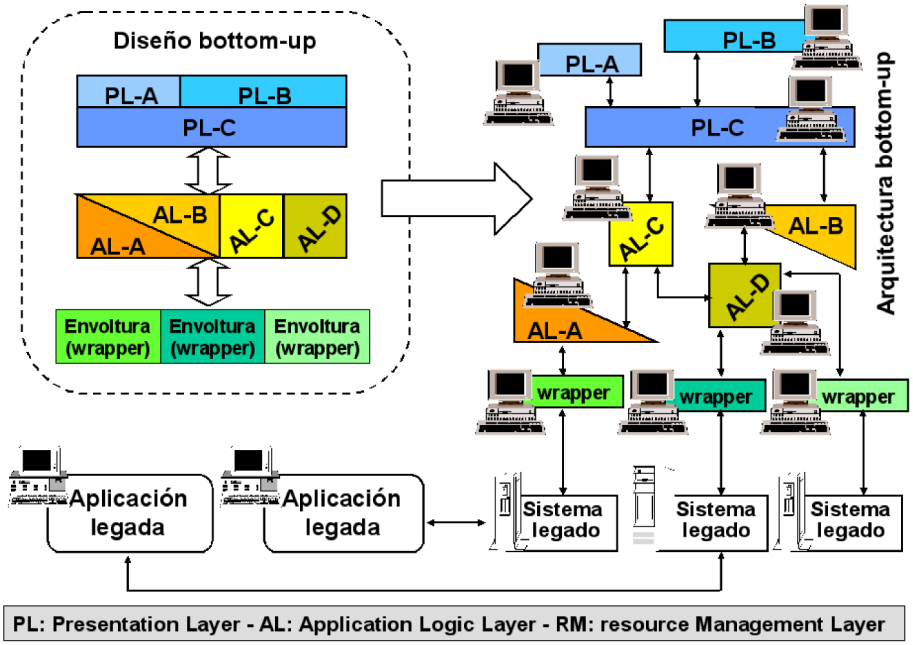

Figura 2.5: Arquitectura de un sistema bottom-up

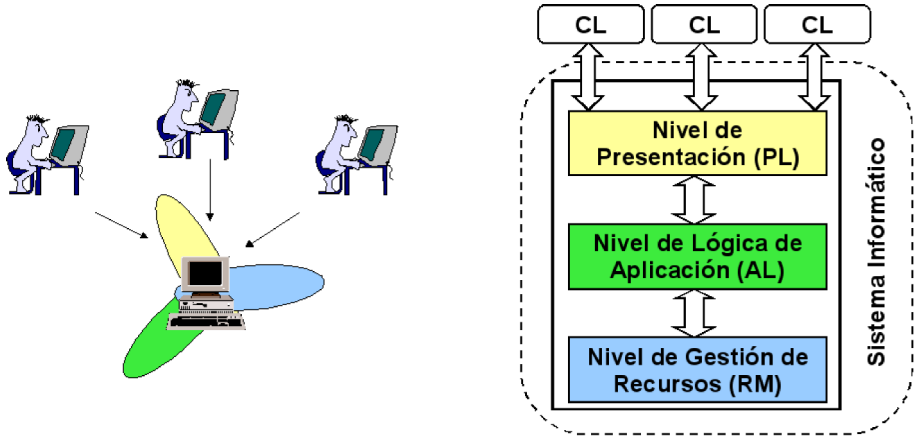

Figura 2.6: Arquitectura 1-capa. Los tres niveles conceptuales están en una sola capa.

El diseño bottom-up es probablemente uno de los escenarios más relevantes para desarrolladores de aplicaciones empresariales debido a que ellos deben tratar con un amplio espectro de aplicaciones existentes, cada una de las cuales han requerido un gran esfuerzo de desarrollo. Exponer estas aplicaciones como servicios -y puntualmente como servicios web disponibles para integrar, promociona el reuso de sistemas ya desarrollados con recursos del pasado.

2.2. Arquitectura de un Sistema Informático

Los tres niveles abstractos descriptos anteriormente son conceptuales y sirven para separar lógicamente la funcionalidad de una aplicación. Al implementarse sistemas reales, los niveles son combinados y distribuidos en diferentes formas; en tales casos no resulta adecuado referirse a estas formas como niveles conceptuales o abstractos, en su lugar se adapta mejor el termino ***capas*** *(tiers)*. Hay cuatro tipos básicos de sistemas informáticos dependiendo de cómo están organizadas las capas: 1-capa, 2-capas, 3-capas y N-capas.

2.2.1. Arquitecturas 1-capa

El ejemplo típico de ***arquitecturas 1-capa*** es un sistema basado en una *mainframe* y con terminales bobas conectadas que muestran información preparada por la mainframe. El principal interés de esta arquitectura, surgida hace varios años, era optimizar la eficiencia del uso de la CPU y del sistema.

En tales casos, los sistemas que se ejecutan en tales arquitecturas son *monolíticos*. Es decir que los tres niveles conceptuales están fusionados en una única capa. Las terminales bobas son meras máquinas con teclados y monitores sin ningún tipo de procesamiento y conforman los clientes del sistema (véase Fig. 2.6).

Estas arquitecturas monolíticas residentes en mainframes son el ejemplo canónico de sistemas legados, ya que los mismas fueron diseñadas en sus orígenes para ser accedidas sólo por terminales bobas. Esta característica es su principal desventaja. Específicamente, estos sistemas carecen de *interfaces para programación de aplicaciones* o *APIs* [1] que permitan a otras aplicaciones interactuar con estos. Por ende se está obligado a tratarlos como cajas negras, y utilizar mecanismos adhoc, como *rascado de pantalla (screen scraping)* [2] , para integrarlos con otros sistemas.

[1] Las interfaces para programación de aplicaciones o APIs *(applications programs interface)* son un conjunto de convenciones de programación que definen cómo se invoca un servicio desde un programa [236]

[2] El *"rascado de pantalla" (screen scraping)* es una técnica que se implementa en una determinada aplicación para interactuar con una aplicación legada que carezca de interfaces. Desde el punto de vista de la aplicación legada, la aplicación que implementa el rascado de pantalla es como un usuario normal que se identifica y accede

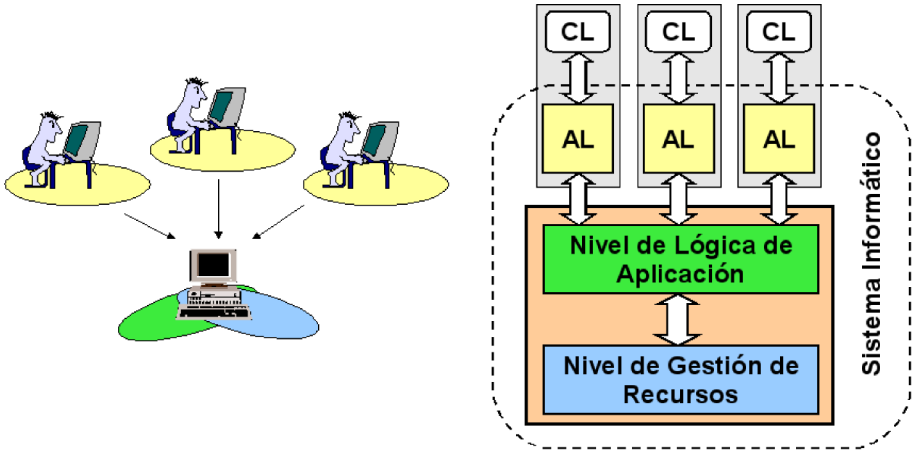

Figura 2.7: Arquitectura 2-capas. Separa el nivel presentación de los otros niveles.

La principal fortaleza de la arquitectura 1-capa, es la performance. Los diseñadores tienen la libertad de fusionar los niveles tanto como deseen, logrando altos niveles de eficiencia. Se logra un sólo ambiente de ejecución, no hay que preocuparse en permitir acoplamiento con otros componentes. Al estar la mayoría de los sistemas implementados en lenguaje *assembler*, si bien lo hace poco portable, también logra un software ultra adaptado para una plataforma determinada. La eficiencia lograda de esta forma, permanece en algunos casos inigualada por otras arquitecturas. Históricamente los sistemas 1-capa no requieren costo para la programación y mantenimiento de clientes, al contrario de otras arquitecturas.

La principal debilidad es que los sistemas 1-capa son piezas de código monolíticas. Su costo de mantenimiento es proporcional a su eficiencia. En los casos de los sistemas legados, son sistemas antiguos que no tienen documentación por consiguiente adaptar el sistema a tales requerimientos es prácticamente imposible, a lo que se suma la dificultad de conseguir programadores calificados [527, 454].

2.2.2. Arquitecturas 2-capas

El verdadero empuje para las ***arquitecturas 2-capas*** lo aportó la aparición de la PC. En lugar de mainframes y terminales bobas, se proponía contar con computadoras de gran porte (servidores y mainframes) y computadoras de pequeño porte con posibilidad real de procesamiento integradas en una red (PCs y estaciones de trabajo). En esta nueva configuración, el Nivel de Presentación podía estar residente en las PCs y estaciones de trabajo junto con los Clientes, en tanto que la Lógica de Aplicación y la Gestión de Recursos permanecer en un servidor. Es decir, se distribuye el sistema y no es necesario que los tres niveles conceptuales estén en el mismo nodo. (véase Fig. 2.7)

El hecho de mover el Nivel de Presentación a las PCs provee dos ventajas importantes:

- Primero, es posible usar para el Nivel de Presentación el poder de procesamiento de las PCs, liberando recursos del servidor que contiene los demás niveles.

- Segundo, es posible adaptar la presentación de la aplicación para diferentes propósitos, sin incrementar la complejidad de todo el sistema. En tal caso, se puede contar con un Nivel de Presentación para propósitos administrativos, y otro para usuarios comunes. Cada faceta de presentación diferente genera un modulo que puede ser desarrollado y mantenido separadamente.

a cada una de sus pantallas, emulando las entradas de teclado y otros dispositivos [15]. Tambien es posible extender el concepto; se puede pensar en screen scraping como un procedimiento para emular la interacción con un sitio web (navegación, llenado de formularios, etc) [215]

Las arquitecturas 2-capas fueron conocidas popularmente en la forma de arquitecturas ***cliente/servidor*** *(C-S)* [369, 274, 246]. Se llama ***cliente*** en estas arquitecturas C-S al Nivel de Presentación más el Cliente real. En tanto que el ***servidor*** en C-S encierra a los niveles de Lógica de Aplicación y de Gestión de Recursos.

El cliente en C-S puede tomar diferentes formas de complejidad e implementar en diferente grado la funcionalidad del sistema. Así es que, teniendo en cuenta la complejidad del cliente los sistemas C-S pueden clasificarse en:

- Arquitecturas con ***cliente liviano*** *(thin client)* el cual sólo implementa una mínima funcionalidad, y
- Arquitecturas con ***cliente pesado*** *(fat client)* con mayor grado de complejidad y funcionalidad.

Los clientes livianos son más fácil de desarrollar, instalar y mantener. Requieren menos capacidad de procesamiento de la plataforma que lo soporta. Los clientes pesados, por otro lado, exigen considerables recursos de plataforma para ofrecer una funcionalidad más rica y compleja.

La adopción de Sistemas C-S han generado un ciclo de evolución positivo para los avances del hardware y las redes. Al volverse más potente las PCs y estaciones de trabajo, el cliente de C-S pudo hacerse más sofisticado. Al volverse más sofisticada esta parte del sistema y por estar distribuida en estaciones de trabajo, se exigió una demanda en conectividad y velocidad en las redes.

La evolución de los sistemas C-S ha realizado importantes aportes a la ingeniería de sistemas distribuidos, entre los cuales se pueden citar:

- Las arquitecturas Cliente-Servidor tienen relación íntima con las ***llamadas de procedimientos remotos*** *o* ***RPCs*** *(remote procedure call)* [63, 236], que son mecanismos de programación y comunicación que permiten al cliente y servidor interactuar en forma de invocaciones de procedimientos transparentes a los detalles del canal de comunicación.
- Por otro lado, las arquitecturas Cliente-Servidor impusieron una nueva modalidad de desarrollo a través del concepto de ***interfaces públicas***. Si se piensa que los clientes C-S son módulos desarrollados y mantenidos en forma independiente, el servidor debe contener interfaces estables y conocidas, para que se pueda interactuar con éste. Así surge el concepto de ***interfaz para programación de aplicaciones*** *o* ***APIs*** *(application programming interface)* [236]. Esto concepto ha cambiado radicalmente la ingeniería en desarrollo de sistemas informáticos. Una API especifica cómo invocar un servicio (al servidor), qué respuesta se puede esperar de éste, y cuáles serán los efectos de tal invocación en el estado interno del servidor. De esta forma, al tener los servidores APIs estables y públicas, es posible construir los clientes en base a las mismas. Por otro lado, mientras se mantenga las especificaciones de las APIs, se puede optimizar y evolucionar los servidores sin afectar los clientes.

Por estos conceptos las arquitecturas C-S se convirtieron en el punto de partida de aspectos cruciales en los sistemas informáticos modernos.

Los programas individuales responsables de la Lógica de Aplicación, se conforman como ***servicios*** ejecutándose en un ***servidor***. La ***interfaz de servicio*** define cómo interactuar con un determinado servicio abstrayendo de los detalles de implementación del mismo. La colección de interfaces de servicios disponibles para los clientes conforma la ***API del servidor***.

El énfasis en las interfaces generó la necesidad de estandarización, proceso que continua hasta hoy. En muchos aspectos, los Servicios Web son el último resultado de los esfuerzos de estandarización.

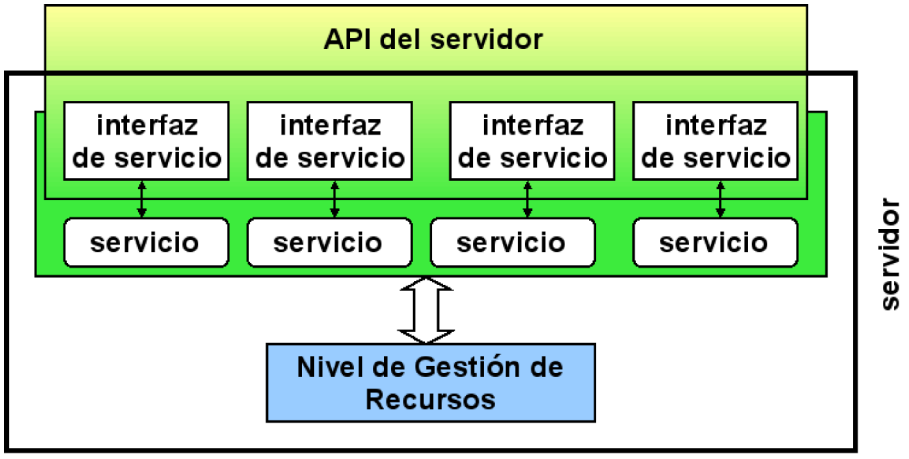

Figura 2.8: Organización interna del nivel de lógica de aplicación en un sistema cliente-servidor.

Ventajas de arquitecturas 2-capas Las arquitecturas 2-capas tienen la ventaja, con respecto a las 1-capa, de ser portables a variadas plataformas debido al desacoplamiento del Nivel de Presentación respecto al resto de los niveles. Es decir, que se puede pensar en diferentes implementaciones del Nivel de Presentación para diferentes plataformas de clientes. Además, al igual que las 1-capa, se puede preservar cierto tipo de optimización en la performance ya que los Niveles de Lógica de Aplicación y el de Gestión de Recursos se mantienen juntos en el servidor.

Las características más importantes del modelo cliente-servidor son la simplicidad, modularidad, extensibilidad y flexibilidad [246]. La simplicidad queda de manifiesto al vincular estrechamente el flujo de datos con el flujo de control, utilizando mecanismos comúnmente aceptados como las llamadas a procedimientos. La modularidad es lograda por organizar y distribuir operaciones en diferentes máquinas como servicios individuales, los cuales se desarrollan y mantienen en forma separada. Si todo el sistema ha sido desarrollado bajo el esquema cliente-servidor, puede ser fácilmente extendible agregando nuevos servicios en la forma de nuevos servidores. Los servidores que ya no satisfacen los requerimientos pueden ser fácilmente modificados o removidos; la única exigencia consiste en mantener las interfaces entre cliente y servidores actualizadas.

Desventajas de arquitecturas 2-capas Existen varias desventajas conocidas en el modelo cliente-servidor, algunas son propias del modelo, otras no son intrínsecas a la arquitectura en sí.

La primera desventaja es el hecho de que el control de los recursos individuales está centralizado en un único servidor. Esto significa que si la computadora que soporta el servidor falla, entonces falla el elemento de control. Tal situación no es tolerable cuando la función de control es crítica para las operaciones del sistema distribuido, como lo pueden ser un servidor de nombres, un servidor de archivos, un servidor de autenticación. De esta forma, la disponibilidad y fiabilidad de un sistema que contenga a muchos servidores es producto de la disponibilidad de todas las computadoras, dispositivos y líneas de comunicación.

Lo segundo es que un servidor puede soportar sólo un número limitado de clientes. Este límite decrece a medida que la interacción entre cliente y servidor es más compleja. Por cada cliente, el servidor debe mantener un contexto de autenticación y de conexión para soportar la interacción individual con cada uno de estos. Además de los recursos computacionales exigidos por esto, el servidor debe soportar la Lógica de Aplicación y la Gestión de Recursos. Como los servidores están instalados en máquinas menos poderosas que una mainframe, se genera la percepción que las arquitecturas 2-capas tienen limitada escalabilidad.

El tercer problema deviene cuando múltiples implementaciones de funciones similares son utilizadas para mejorar la performance y disponibilidad de servicio, distribuyendo la carga de peticiones de los clientes. El esquema cliente-servidor exige que se mantenga la consistencia entre

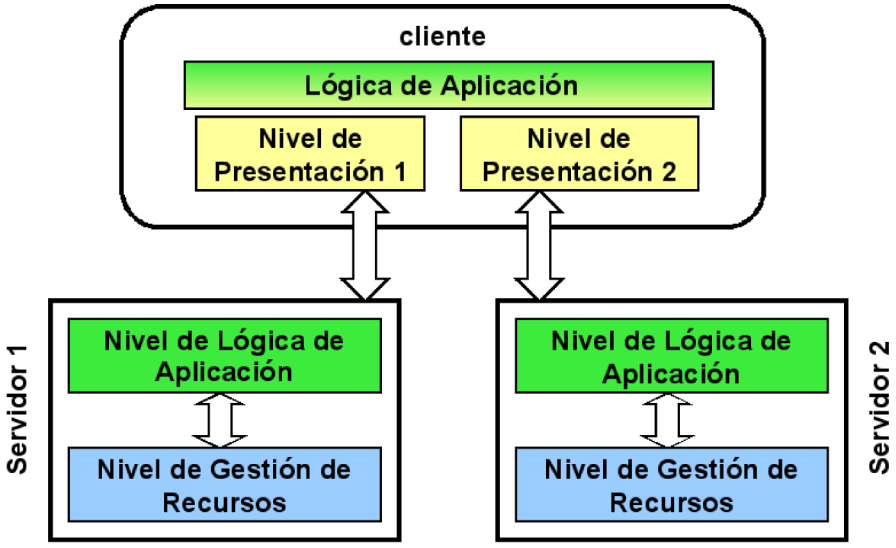

Figura 2.9: La integración en arquitecturas 2-capas se realiza en el cliente

las diferentes instancias de estas funciones; lo que genera un aumento de costos de mantenimiento en el sistema.

Otra desventaja, surge cuando un sistema 2-capas se convierte en una aplicación legada. Una vez que el cliente comienza a ser independiente del servidor (en el sentido que comienza a ser una pieza separada de código, posiblemente mantenido por otro grupo de desarrolladores diferente), puede ser desarrollado en forma propia y separada. En tal situación, era común usar a los clientes para conectar a diferentes servidores con el propósito de integrar sus servicios. Esto es, en principio, una buena idea, pero usa una arquitectura incorrecta (véase Fig. 2.9). Por ejemplo, si un cliente interactúa con dos servidores por ejemplo, es necesario que conozca sendas APIs. Un cliente de este tipo, debe estar pendiente de las modificaciones que sufran cualquiera de los servicios, aumentando su complejidad en el desarrollo y mantenimiento, restringiendo su propia vida útil. Se puede decir que el cliente es responsable de la integración de los servicios; es decir coordinar la información de ambos servidores, tratar con las excepciones y fallas en forma coherente, controlar el acceso a los mismos, etc. En otras palabras, surge un nuevo nivel de lógica de aplicación incrustado en el cliente. Tales configuraciones se vuelven con el tiempo inmanejables. Más aún, cada procedimiento de adaptación de un cliente a un nuevo servidor debe realizarse desde cero por cada posible combinación de servidores.

En resumen, a pesar de su éxito, las arquitecturas 2-capas tienen la reputación de ser poco escalables e inflexibles a la hora de integrar diferentes sistemas. Estos impedimentos no son propios de la arquitectura, pero delinearon la siguiente ola de evolución en los sistemas informáticos distribuidos.

2.2.3. Arquitecturas 3-capas

Las ***arquitecturas 3-capas*** surgen como la natural necesidad de integrar varios servicios provistos por varios servidores en un único cliente.

Debido a las mejoras logradas en la tecnología de redes y al hecho de contar con servidores con APIs públicas y estables, los sistemas informáticos dejaron de ser islas de información en donde los clientes sólo se podían comunicar con un solo servidor. Como se explicó anteriormente (2.2.2), el problema subyacente fue no contar con capacidades de integración de varios sistemas. En sí, las arquitecturas 3-capas son mejor entendidas si se las piensan como la solución a este problema arquitectural.

Las arquitecturas 3-capas solucionan el problema agregando una nueva capa entre el cliente

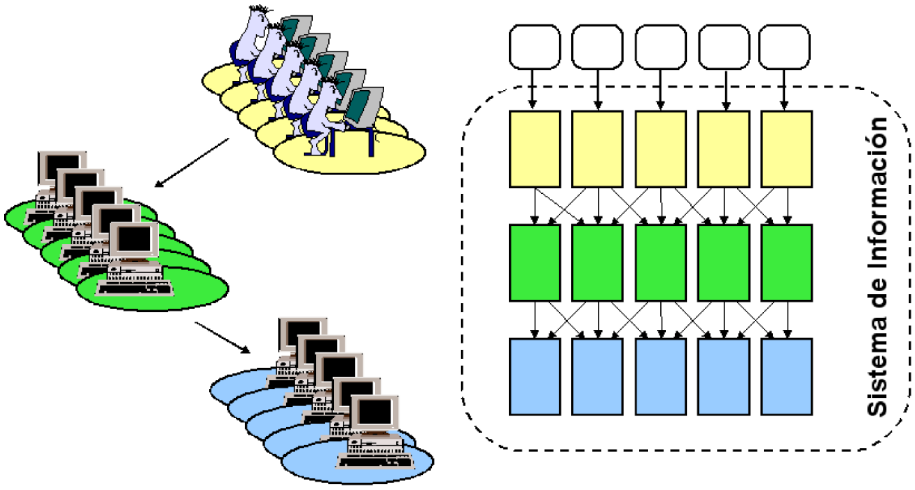

Figura 2.10: En una arquitectura 3-capas los niveles están completamente separados.

y el servidor. Esta nueva capa es la encargada de la integración de las aplicaciones subyacentes; es decir, aquí es dónde reside la lógica de aplicación de la integración. De hecho algunos autores [246] consideran que la arquitectura 3-capa extiende al modelo de 2-capas y lo llaman *arquitectura cliente-servidor de tres capas*.

Las arquitecturas 3-capas son mucho más complejas que los modelos C-S, por ende también son más difíciles de caracterizar. Sin embargo, en forma abstracta, las arquitecturas 3-capas están basadas en la clara separación de los 3 niveles conceptuales (véase Fig. 2.10). El Nivel de Presentación reside en el cliente, al igual que las arquitecturas 2-capas. El Nivel de Lógica de Aplicación se presenta implementado en una capa intermedia. Además de proveer los servicios de lógica de negocios, esta capa debe proveer los medios para lograr la integración de diferentes servidores o servicios, por esta razón esta capa es denominada ***middleware***. El Nivel de Gestión de Recursos esta compuesto por todos los servidores que el sistema trata de integrar. Estos servidores, pueden a su vez ser aplicaciones completas de diferentes arquitecturas (véase Fig. 2.11).

Una *plataforma middleware* o simplemente ***middleware*** es un tipo de software distribuido el cual conecta a diferentes tipos de aplicaciones y provee transparencia de distribución entre los componentes conectados. Es usado como puente entre las diferentes heterogeneidades presentes en un sistema [482]. En su sentido más básico, el middleware es un software de conectividad que consiste en un conjunto de servicios que permite que múltiples procesos ejecutándose en una o más máquinas interactúen a través de una red de computadoras. Las plataformas de middleware son esenciales para migrar aplicaciones 1-capa a aplicaciones cliente/servidor y proveer comunicaciones a través de diferentes plataformas [142].

A pesar que las arquitecturas 3-capas surgen principalmente para plataformas de integración, pueden ser usadas en los mismos escenarios que las arquitecturas 2-capas. Se puede cambiar un sistema 2-capas a su versión de 3-capas a través de la separación de la Lógica de Aplicación y la Gestión de Recursos. Al distribuir estos dos niveles en nodos diferentes, se logra balancear mejor la carga de trabajo de los servidores, por consiguiente se solucionaría, en principio, el problema de escalabilidad presente en 2-capas.

Los sistemas de 3-capas permiten implementar una Lógica de Aplicación menos acoplada con la Gestión de Recursos, logrando que ésta sea más portable y reusable. Sin embargo, se incrementa la complejidad y el costo de interconexión entre capas, generando una pérdida de rendimiento en ese caso.

Las arquitecturas 3-capas introducen importantes conceptos que completan y amplían los presentados en 2-capas. Los más importantes son:

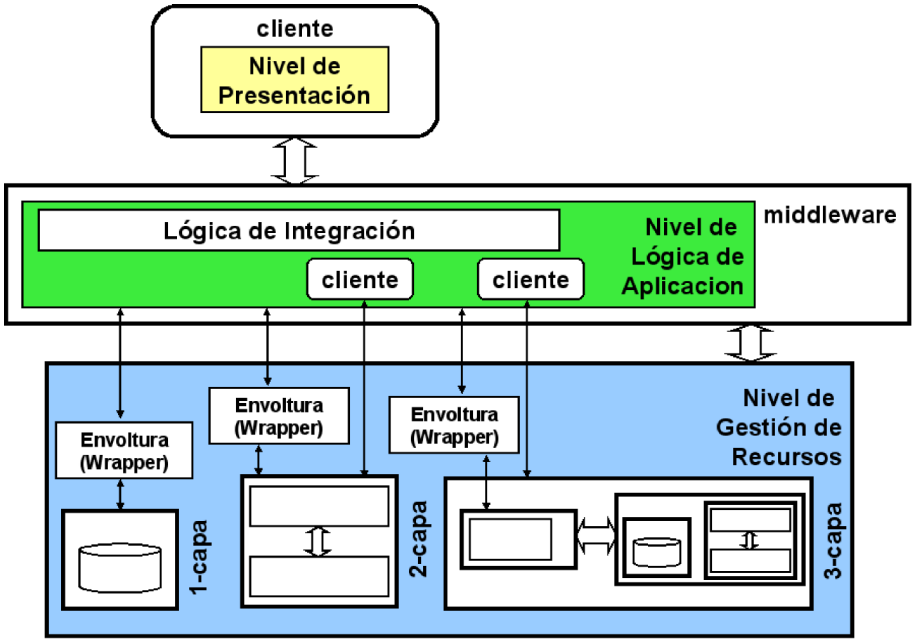

Figura 2.11: Integración de sistemas de usando middleware.

- ***Interfaces de Gestión de Recursos.*** El nivel de Gestión de Recursos debe implementarse para proveer interfaces, de forma tal que puedan ser usadas para acceder desde la Lógica de Aplicación ejecutándose en el middleware, que ahora está desacoplada. Estas interfaces deben ser estándares, de forma tal que el código de la Lógica de Aplicación pueda acceder en forma remota a varias fuentes de recursos de manera uniforme. Este principio fue el origen de las interfaces de conexión a bases de datos ODBC (open database conectivity)[300] y JDBC (Java database conectivity)[469].

- ***Integración de recursos a través de middleware.*** La arquitectura 3-capas tiene su fortaleza al tratar con la integración de diferentes recursos. Las infraestructuras de middleware modernas no sólo proveen el soporte a la Lógica de Integración, sino que también están dotadas con propiedades adicionales como ser la garantía en las transacciones, balanceo de carga, replicación, persistencia y otras.

Como se explicó, además de proveer los servicios de lógica de negocios, el middleware debe proveer determinados servicios fundamentales [246]:

- ***Servicio de Directorio:*** este servicio es requerido para localizar servidores o servicios de aplicación y de recursos, y de encaminar los mensajes a éstos.

- ***Servicio de Seguridad:*** un servicio integral de seguridad es necesario para brindar un mecanismo de seguridad inter-aplicaciones.

- ***Servicio de Tiempo:*** para establecer un formato universal de representación de tiempo entre las diferentes plataformas de servidores a integrar. Esta funcionalidad es crítica para mantener bitácoras de errores y estampillas de tiempo, y asegurar la sincronización entre las aplicaciones.

- ***Servicio de Transacciones:*** esencialmente para asegurar la semántica de transacciones soportando los mecanismos de *commit*, *rollback* y recuperación. La funcionalidad es vital para asegurar la integridad en los registros de operaciones del sistema.

El énfasis en el desarrollo de estos principios provistos por los middlewares ha generado otra ola de estandarización. Por ejemplo, se han desarrollado estándares para realizar transacciones

a través de diferentes sistemas, como por ejemplo DTP de X/Open (Distributed Transaction Processing) propuesto por el Open Group, anteriormente X/Open [489]. También se desarrollaron estándares para propiedades globales e interfaces entre plataformas de middleware usando orientación a objetos, como CORBA de OMG [352] y COM+ de Microsoft [296].

Ventajas de arquitecturas 3-capas. Comparada con la arquitectura 2-capas, el modelo 3-capas presenta dos ventajas importantes:

- *Mejor Transparencia*: los servidores dentro de la capa de aplicación permiten a una aplicación desacoplar la interfaz de usuario (Nivel de Presentación) de la Gestión de Recursos, proveyendo así una mejor transparencia de ubicación y migración. Esto significa que la ubicación y/o la implementación de la capa de recursos puede cambiar sin afectar los programas clientes.
- *Mejor Escalabilidad*: en las arquitecturas 3-capas, la presencia de los servidores en la capa de aplicación en una estructura 3-capas genera una nueva dimensión en la escalabilidad de grandes sistemas.

En sí, las arquitecturas 3-capas poseen como ventaja principal la capa de middleware en la cual reside el Nivel de Lógica de Aplicación, con funcionalidades de integración. A pesar de que esta nueva capa genera una disminución en el rendimiento, la arquitectura gana en flexibilidad. De la misma forma, la performance pedida en la comunicación entre la Lógica de Aplicación y Gestión de Recursos, es compensada con la posibilidad de distribuir la capa de middleware en varios nodos, obteniendo significantes mejoras en la reusabilidad, balanceo de carga y disponibilidad del sistema.

Desventajas de arquitecturas 3-capas. La principal desventaja en las arquitecturas 3-capas tiene relación con el problema de aplicaciones legadas. En las arquitecturas 2-capas, este problema surgía cuando un cliente se comunicaba con 2 o más servidores. En el caso de 3-capas, el problema deviene al integrar aplicaciones en Internet. Las arquitecturas 3-capas no fueron pensadas para este tipo de integración. Por otro lado, al tratar de integrar 2 aplicaciones 3-capas, el principal impedimento es la falta de estándares. Los Servicios Web son una alternativa viable para solucionar este problema.

2.2.4. Arquitecturas N-capas

Las ***arquitecturas N-capas*** son el producto de llevar el modelo de 3-capas a su máxima genericidad y de la importante relevancia de Internet como canal de acceso. De aquí surgen las dos concepciones de los modelos N-capas:

1. *Composición de diferentes sistemas.* Un sistema es N-capas o multicapa si posee una capa de Gestión de Recursos que no sólo contenga recursos simples, como bases de datos, sino que esté compuesta por varios sistemas completos 1-capa, 2-capas y 3-capas. Es decir son la estructura genérica presentada en la Figura 2.11.
2. *Agregando conectividad por internet.* Por ejemplo, se puede considerar a un sistema N-capas si utiliza un *Servidor Web* para implementar su Nivel de Presentación. El Servidor Web es tratado como una capa adicional por su complejidad. En tal configuración el cliente es un Navegador Web y el Nivel de Presentación esta distribuido entre el Servidor Web y el código para preparar las páginas HTML que se presentarán, comúnmente llamado *HTML filters*. (véase Fig. 2.12).

Las arquitecturas multicapas demuestran que las aplicaciones actuales poseen altos grados de complejidad y pueden contener varias capas como resultado de sucesivas integraciones realizadas

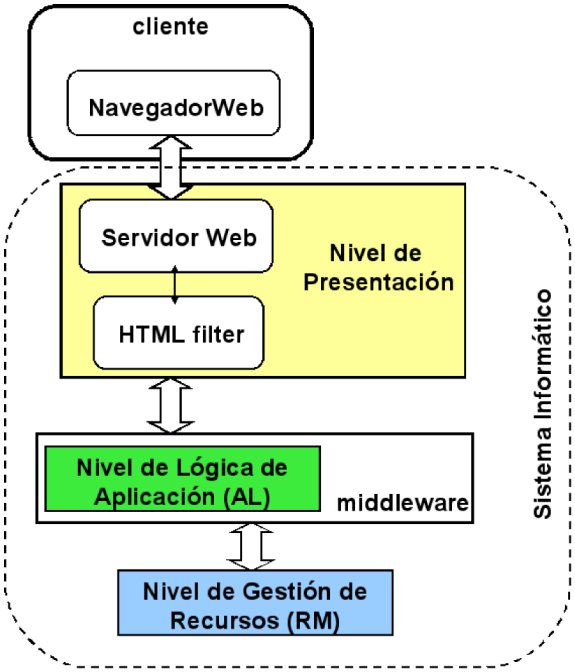

Figura 2.12: Arquitectura N-capa creada por extender un sistema 3-capas agregado un servidor Web.

para construir un sistema de información; el cual, en un futuro, será un ladrillo de construcción para otro sistema integrador.

Este hecho es en sí la principal desventaja del modelo N-capas. El poder integrar varios sistemas, lleva a la coexistencia de varias plataformas de middleware, usualmente con funcionalidad redundante [455]. Además se le suma el hecho que la dificultad y el costo de desarrollar, poner en marcha y mantener estos sistemas se incrementa exponencialmente a medida que se adicionan capas.

La mayoría de los sistemas N-capas actuales se componen por una colección de redes, servidores, clusters y conectividad entre varios sistemas. En un sistema N-capas puede ser difícil identificar los límites de cada subsistema. Por ejemplo puede tener un grupo de clientes que acceden vía Internet a un servidor web empresarial pasando por un *firewall*. Este servidor web puede estar implementado por un *clusters de computadoras*. Internamente, el sistema puede tener clientes que acceden por una LAN a una web corporativa residente en el servidor web o accedan directamente a la lógica de aplicación implementada en una plataforma middleware. Es común ver al Nivel de Lógica de Aplicación implementado en un cluster de máquinas. Por debajo de toda esta infraestructura se encuentra el Nivel de Gestión de Recursos, llamado en estas configuraciones *back-end* o *back-office*. El *back-end* puede estar constituido por una variedad de posibilidades, desde un simple servidor de archivos, un sistema de gestión de base de datos ejecutándose en una mainframe, hasta vínculos con sistemas completos 2, 3 o N-capas.

2.2.5. Distribución de Niveles y Capas

A modo de conclusión se puede ver que existe un patrón reiterado en la evolución de las arquitecturas de la monocapa hasta la multicapas: la sucesiva adición de capas. Con cada capa que se agrega, se busca desacoplamiento, por consiguiente se logra flexibilidad, funcionalidad y posibilidades de distribución.

La debilidad de este proceso de evolución reside en que con cada capa agregada se afecta directamente la performance global del sistema, generado por el costo de conectividad entre capas. Además, cada capa introduce más complejidad en términos de mantenimiento y gestión. Cuando un nuevo modelo de arquitectura aparece, indudablemente será criticado por la pérdida

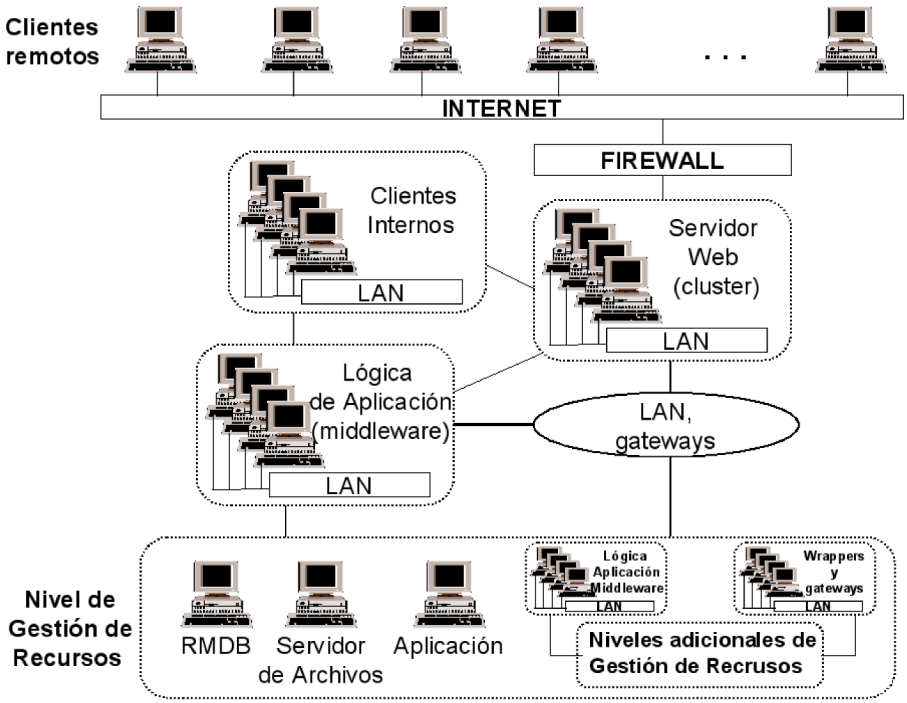

Figura 2.13: Sistemas N-capas y toda su heterogeneidad.

de performance. Pero se debe tener en cuenta el grado de flexibilidad que cada arquitectura ofrezca, será la variable que define si prevalece o no.

Conocer la evolución de las arquitecturas desde la 1-capa hasta la N-capas es un punto de partida para conocer los Servicios Web. Después de todo la arquitectura de Servicios Web es otro ejemplo de construir una capa nueva por encima de las ya existentes.

2.3. Comunicación en un Sistema Informático Distribuido

Al estar ejecutándose un sistema distribuido, varios procesos deben interactuar y coexistir. Los procesos son componentes activos con estado y comportamiento. El estado consiste en los datos manejados por el proceso. El comportamiento corresponde a la implementación de la lógica de aplicación.

Al entender que un sistema informático distribuido está constituido por varias capas, se entiende que debe existir una forma de comunicación entre estas. Los procesos cooperan unos con otros a través del intercambio del mensaje. Un mensaje consiste en una secuencia de bytes que son transportados entre dos procesos por un determinado medio de comunicación [400]. Hay dos principales características a tener en cuenta en la comunicación de pasaje de mensajes: el *mensaje* empleado en la comunicación y el *mecanismo* usado para enviar y recibir mensajes [246].

2.3.1. ¿Qué es un mensaje?

Un ***mensaje*** es una colección de objetos de datos que consiste en un ***encabezado*** *(header)*, generalmente de longitud fija; y un ***cuerpo*** *(body)*, generalmente de longitud variable. Un mensaje es generado por un proceso, gestionado su traslado y entregado a su destino [246]. En el intercambio de mensajes un proceso asume el rol de ***emisor*** o ***remitente*** el otro proceso es ***receptor*** o ***destinatario***.

Existe un *tipo* asociado con un mensaje que provee información sobre el diseño estructural del mismo, indicando cómo el mensaje debe ser identificado. Un mensaje puede ser de cualquier tamaño y puede contener datos o punteros a datos que están fuera de la porción continua del

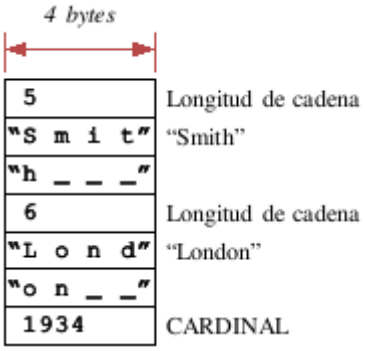

Figura 2.14: Mensaje XDR. Mensaje del registro ('Smith','London',1934).

mensaje. Los contenidos del mensaje son determinados por el proceso remitente. Por otro lado, el encabezamiento del mensaje tiene información relacionada con el sistema en sí, la cual puede ser suministrada por la plataforma que gestiona la entrega y el transporte de mensajes.

Los mensajes pueden ser completamente estructurados o *"tipados"*, o sin estructura. Los mensajes desestructurados tienen la flexibilidad de ser implementados de acuerdo a las necesidades de los puntos finales de comunicación, es decir del proceso remitente y del proceso receptor. Sin embargo, el principal problema de esta opción se presenta cuando algunas partes del mensaje, que contienen información del transporte del mismo (por ej: nombre de puerto) debe ser interpretadas por el software que provee la distribución o el protocolo de comunicación para ser correctamente transmitidos y entregados; si esta información no es tipada, se hace dificultosa o imposible su interpretación por parte de estas plataformas de comunicación. En algunas redes heterogéneas, sólo se puede transmitir información estructurada o tipada (enteros, reales, string, etc) para que sea posible la transferencia transparente de los datos.

Para lograr el intercambio de valores de datos entre computadoras diferentes, es necesario realizar un mapeo entre las estructuras de datos y los items de datos a mensajes. Los datos estructurados o agregados, como por ejemplo un registro, un objeto o un grafo, deben ser "aplanados" o *linealizados* como una secuencia de datos básicos para ser transmitidos. De la misma forma, el proceso debe revertirse, reconstruyendo las estructuras originales, una vez que el mensaje llegue a destino. Las operaciones de *linealización (marshalling/unmarshalling)* de datos son funciones básicas ofrecidos por las plataformas de distribución para transmitir datos complejos.

Ejemplo de un mensaje estructurado Como ejemplo de estructura de mensajes [109], se puede citar al estándar *Sun XDR (External Data Representation)* [143, 420] el cual define una representación los tipos de datos de uso comunes, simples y estructurados, como son los strings, arreglos, secuencias y registros. El estándar fue desarrollado por Sun Microsystems para usarlo en el intercambio de mensajes entre clientes y servidores en el protocolo NFS [427], que permite el acceso a archivos remotos sobre la red de forma transparente.

La figura 2.14 muestra un mensaje en el estándar Sun XDR. El mensaje entero consiste en secuencia de objetos 4 bytes, usando la convención de que un cardinal o entero ocupa un objeto y que los string de 4 bytes también ocupan un sólo objeto. Los arreglos, las estructuras, las cadenas de carácteres son representadas como secuencias de bytes con longitudes especificadas. Los carácteres se representan en ASCII. El uso de un tamaño fijo de objeto en el mensaje reduce la carga computacional a cambio de ancho de banda.

2.3.2. Mecanismos de Comunicación

La figura 2.15 es un diagrama de espacio tiempo que muestra el proceso de comunicación. El tiempo corre de izquierda a derecha. El proceso 1 es el emisor y el proceso 2 es el receptor. Un

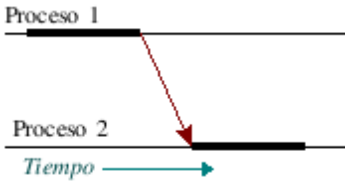

Figura 2.15: Intercambio de Mensajes entre dos procesos.

proceso puede estar en estado pasivo o activo. El estado activo en la figura se indica con la línea más gruesa. Un proceso puede realizar cómputos sólo cuando está en estado activo.

Interacciones con Orientación a Mensajes y con Orientación a Respuesta

Una simple clasificación del esquema del mecanismo de comunicación [324] es basada en el flujo del diálogo entre remitente y destinatario. Si el proceso remitente transmite un mensaje al receptor pero no espera una respuesta, el intercambio de mensajes es en una única dirección. En este caso se dice que el patrón de la comunicación es ***orientado a mensajes***. Si por otro lado, el destinatario contesta un mensaje de respuesta al emisor original, el cual lo espera para completar la interacción, se está en presencia de un patrón de comunicación ***orientado a respuesta***.

Interacciones Bloqueantes y No-Bloqueantes

Otra clasificación [10], ortogonal a la descripta en el párrafo anterior, es categorizar a la interacción entre procesos en *asincrónica* o *sincrónica* o, formalmente hablando, si es *bloqueante* o *no-bloqueante*[3]. En esquemas ***sincrónicos*** o ***bloqueantes*** el emisor queda en espera o en estado pasivo durante el envío del mensaje hasta que se verifique el arribo al destinatario. Por el contrario, durante un esquema ***asincrónico*** o ***no bloqueante***, el emisor permanece activo, posiblemente haciendo otros cómputos, inmediatamente después de enviar el mensaje.

Llamadas Sincrónicas o Bloqueantes En las interacciones bloqueantes, un hilo de ejecución que llama a otro hilo debe esperar hasta que sea devuelta la respuesta o verificación antes de continuar con su ejecución. Esperar la respuesta a una llamada tiene la ventaja de simplificar el diseño. Es más fácil de entender pues se sigue con la natural organización de las llamadas a métodos o procedimientos. También existe una gran correlación entre el código del programa que invoca y el código del programa invocado. Ambos componentes están fuertemente ligados en cada interacción, lo que simplifica el testeo y el análisis de perfomance.

Como resultado, la interacción sincrónica ha dominado la mayor parte de formas de middlewares. Cuando se separa el Nivel de Presentación en las arquitecturas 2-capas, la interacción con el resto de los niveles se realiza a través de llamadas sincrónicas remotas. De similar forma, se establece comunicación sincrónica entre la Lógica de Aplicación y la Gestión de Recursos en arquitecturas 3-capas.

Todas estas ventajas, pueden ser vistas como desventajas también. El hecho de esperar la respuesta, es un factor importante en términos de performance. Si la llamada espera en ser completada, se desperdician significativamente tiempo y recursos; en particular si el proceso fue intercambiado fuera de la memoria. Esta situación puede repetirse en cada capa, por lo tanto se agrava a medida que se agregan capas. Además, si cada llamada genera una nueva conexión, es posible que se desborde el límite de conexiones permitido, al tener muchas llamadas en espera de verificación/respuesta.

[3]La definición formal de sincronismo involucra la existencia de límites bien definidos de tiempo en un canal de comunicación para ser posible la transmisión de un mensaje. A los efectos de este capítulo, se prescinden los detalles de tiempo en sistemas distribuidos.

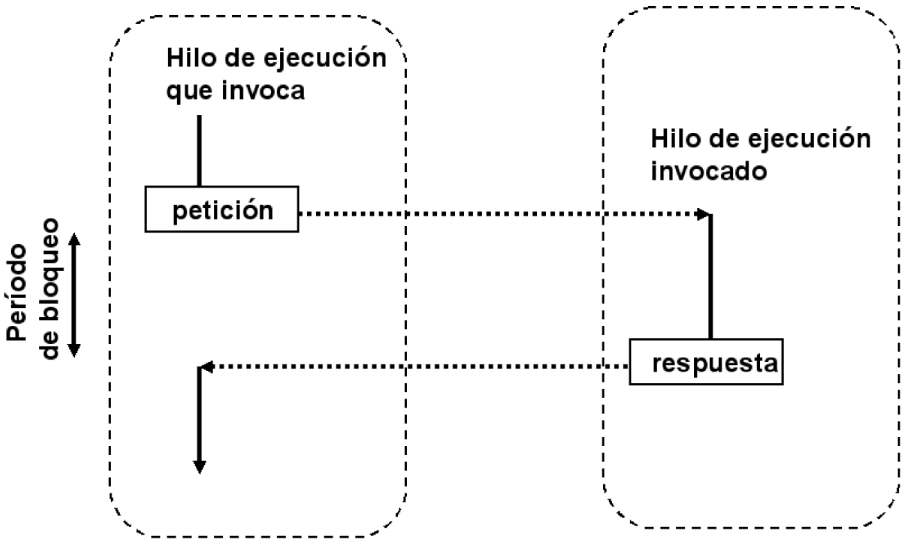

Figura 2.16: Una llamada asincrónica espera por su respuesta.

Por último, la fuerte integración entre componentes impuesta por la interacción sincrónica puede ser imposible de mantener en ambientes altamente distribuidos y heterogéneos. Esta interacción exige que el proceso emisor y el proceso destinatario del mensaje estén en línea al mismo tiempo y se mantengan operacionales mientras dure el diálogo. Esto tiene efectos importantes sobre la tolerancia a fallas y los procesos de actualización que exigen que ambos componentes queden fuera de línea. Nuevamente, esto se agrava a medida que se incrementa el número de capas.

Llamadas Asincrónicas o No-bloqueantes Si bien, en muchos casos, la necesidad de trabajar interactivamente hace que las limitaciones del sincronismo sean inevitables, existen escenarios en dónde no es necesario el trabajo con llamadas bloqueantes. Obviamente, la interacción asincrónica o no-bloqueante es la alternativa. Como ejemplo común a este tipo de interacciones, se puede citar al e-mail. Los mensajes de correo electrónico son enviados a un sevidor de e-mail el cual contiene una casilla de mensajes que almacena el correo hasta que un cliente de correo permite al destinatario leerlo y eventualmente contestarlo. Aquí el cliente del remitente no debe esperar respuesta, no existe correspondencia uno-a-uno entre mensaje y respuesta, y de hecho la respuesta puede nunca recibirse.

Los sistemas informáticos distribuidos soportados mediante comunicaciones asincrónicas, pueden ser vistos de manera similar. Se puede enviar un mensaje y, algún tiempo más tarde, se puede controlar si la respuesta ha sido remitida. Entre tanto, el programa llamador puede realizar otras tareas, eliminando la necesidad de coordinación en la interacción.

Históricamente, el modelo es similar a la noción de *procesamiento en batch*. En algunas formas primitivas de arquitecturas clientes/servidor, precedentes a las RPCs (véase Sec 3.2), se utilizaba interacción asincrónica. Luego, los Monitores de Transacciones (véase Sec. 3.3) incorporaron interacción no-bloqueada en la forma de colas para implementar trabajos en batch. Hoy en día, los sistemas asincrónicos más relevantes son los Middlewares Orientados a Mensajes (MOM) (véase Sec. 3.5.3) y los Brokers de Mensajes (véase Sec. 3.6), usados típicamente en arquitecturas n-capas e integración de aplicaciones interempresariales, para lograr una reducción en el acoplamiento entre capas.

La interacción asincrónica es utilizada en escenarios en línea. Es usada en múltiples aplicaciones para manejar un alto número de conexiones concurrentes. En tales casos, la petición se trata asincrónicamente, pero el proceso llamador espera activamente por la respuesta. En tales configuraciones, se puede reducir efectivamente los problemas del manejo de conexiones, tolerancia

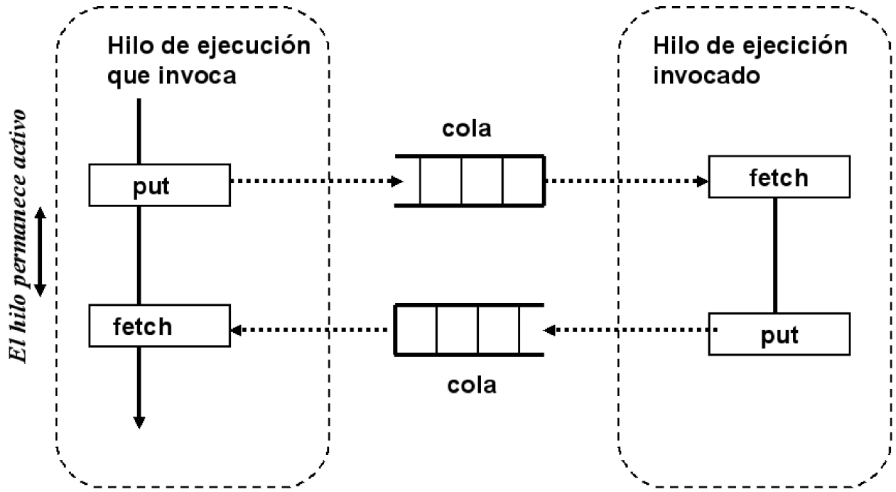

Figura 2.17: En una llamada asincrónica las colas permiten al proceso invocante seguir ejecutándose.

	No Bloqueante	***Bloqueante***
Orientado a Mensajes	Servicio de Datagramas	Rendezvous
Orientado a Respuesta	Invocación a Servicio Remoto (RSI)	Llamada a Procedimiento Remoto (RPC)

Cuadro 2.1: Clasificación de Mecanismos de Comunicación

de fallas, dependencia entre componentes y formato de representación.

El modelo asincrónico de comunicación, también es el más usado cuando la comunicación no respeta el patrón *petición-respuesta*. Ejemplos de tales comunicaciones incluyen diseminación de información, notificación de eventos, roles de productor/consumidor y sistemas de publicación/-suscripción (véase Sec 3.17).

La interacción asincrónica exige almacenar los mensajes en algún espacio intermedio hasta que el destinatario lo recupere, respetando el principio de *resguardo y remisión (store and forward)*, carácteristico de los sistemas de colas [157]. Tal estructura de almacenamiento puede tener funcionalidad asociada que ya no es necesario que la posean algunas de las aplicaciones interactuantes. A raíz de esto, muchos de los sistemas de colas del pasado sólo se limitaban repartir y recibir mensaje entre componentes. Ahora son usados como brokers que filtran y controlan el flujo de mensajes, para implementar estrategias complejas de distribución, manipular el formato de datos o su contenido a medida que los mensajes transitan las colas. Esto es particularmente usado en sistemas n-capas pues permite la separación de conceptos de diseño y permite alojar la lógica que atañe al intercambio de mensajes en las colas, antes que en las envolturas *(wrappers)* o componentes. Tal separación permite cambiar la forma de los mensajes, es decir se puede filtrar, transmitir, traducir o distribuir sin modificar los componentes que remiten y reciben los mensajes.

2.3.3. Clasificación de Mecanismos

Los dos patrones de comunicación y los dos esquemas de sincronismo producen cuatro categorías para clasificar los mecanismos de comunicación, descritas en el Cuad. 2.1.

Las iteracciones basadas en *Llamada a Procedimiento Remoto (RPC)* [63] es un ejemplo de comunicación sincrónica o bloqueante orientada a respuesta. El emisor envía una petición y queda pasivo esperando que el destinatario emita una respuesta con el resultado de la petición. Por otro lado, un ejemplo de comunicación asincrónica orientada a respuesta es la *invocación a servicio remoto RSI (remote service invocation)* [87] o las *RPCs Asincrónicas* [481]. Durante una RSI o una RPC Asincrónica, el emisor queda activo mientras el destinatario procesa la

respuesta. A pesar que las invocaciones asincrónicas hacen un mejor uso del pararelismo ofrecido en los sistemas distribuidos, las RPCs convencionales está basada en un paradigma popular de la programación (programación modular y llamada a procedimientos, *véase Sec. 3.2.1*) siendo más intuitiva su interpretación.

Los *servicios de datagramas* son un ejemplo de comunicación asincrónica orientada a mensajes. En este caso el emisor transmite un mensaje sin perder su estado de activo luego de enviarlo, y no espera ninguna respuesta. En general se considera *datagrama* a los paquetes enviados por un servicio no-confiable a través de una red de computadoras [262]. Un servicio no-confiable (unreliable) no notifica al remitente si un mensaje llegó a destino o no. Como ejemplo típico de estos servicios, se puede citar a aquellos que implementan el protocolo UDP [390] (User Datagram Protocol) de Internet.

Por el contrario, el *rendezvous* [27, 221], utilizado para la sincronización entre dos procesos, es un caso de comunicación sincrónica orientada a mensajes. En este ejemplo concreto, la comunicación sincrónica usada en el randezvous sirve para establecer una estampilla de tiempo (lógica) común para ambos procesos.

2.4. Resumen

La evolución de los sistemas informáticos distribuido está fuertemente atada a los avances en el hardware en la redes de computadoras. La caracterización de esta evolución puede ser mejor visualizada si entendemos a un sistema informático como una pila de tres niveles conceptuales abstractos con funcionalidades y características propias: la capa de presentación, la de lógica de aplicación y la de gestión de recursos.

En sus orígenes, los sistemas informáticos eran basados en *mainframes*. Arquitectura monolítica, los tres niveles conceptuales se fusionaban en una única capa residente en un servidor centralizado. Con la aparición de las redes de computadoras y el hecho que las computadoras personales fueron unidades de procesamiento más potentes, fue posible separar físicamente parte de la funcionalidad del sistema informático y alojarlo en las estaciones de trabajo. Surgen las arquitecturas *cliente-servidor* de dos capas: el nivel de presentación reside en la capa cliente, y el nivel de lógica de aplicación y gestión de recursos en el servidor. El esquema cliente/servidor fue el primer paso importante hacia los sistemas distribuidos modernos, planteando y desarrollando importante conceptos como las RPCs, las interfaces de servicio y las APIs.

El creciente desarrollo de aplicaciones 2-capas y la necesidad de integrar tales aplicaciones con sistemas legados y entre si, condujo al surgimiento de las arquitecturas 3-capas con una capa intermedia llamada *middleware* entre la capa cliente y la de servidor. Es en la capa de middleware en donde el esfuerzo de integración de funcionalidades de diferentes servidores tiene lugar. Si bien las arquitecturas 3-capas pueden considerarse como un paradigma para la construcción de nuevos sistemas, su principal escenario es la *integración de aplicaciones* siendo esto una nueva forma de distribución de sistemas.

Las arquitecturas N-capas surgen por llevar a los esquemas 3-capas, no sólo para solucionar problemas de integración de múltiples servidores, sino para lograr que las aplicaciones tengan alta disponibilidad en diferentes formas de presentación y una variedad heterogénea de recursos de datos. Por un lado una aplicación N-capa puede contener diferentes implementaciones de su nivel de presentación; pueden coexistir diferentes tipos de clientes que requieran una presentación vía Web y otra a través de una aplicación de escritorio. Por otro lado, la gestión de recursos puede contar con diferentes fuentes, un sistema de base de datos, un servidor LDAP, una aplicación legada monocapa, etc.

En cada uno de estos escalones de evolución siempre se ha sacrificado rendimiento por flexibilidad. En algunos casos resulta inevitable, como ser en los escenarios de integración o de reutilización de aplicaciones legadas. Las ideas planteadas en este capítulo son esenciales para entender a los Servicios Web y colocarlos a ellos en una correcta perspectiva. Los Servicios Web

son un paso más en este proceso evolutivo, el último y posiblemente el más significativo, pero un escalón al fin. Los Servicios Web como esquema solucionan problemas que las arquitecturas previas no pudieron solucionar de manera apropiada. También los Servicios Web pueden ser vistos como una nueva capa por encima de las plataformas de middleware y de integración de aplicaciones existentes. Esta nueva capa permite a los sistemas distribuidos interactuar a través de Internet y con el esfuerzo de estandarización que impulsan se trata de minimizar el costo de desarrollo asociado.

Capítulo 3

Middleware

Un ***Sistema de Middleware*** [10] facilita y gestiona la integración entre aplicaciones a través de plataformas de computación heterogéneas. Es una solución arquitectural para hacer frente al problema de integrar un conjunto de servidores y aplicaciones bajo interfaces de servicios comunes. Las formas propuestas de middleware son amplias y muy variadas, ya que no es lo mismo integrar dos bases de datos residentes dentro de la misma LAN que integrar dos aplicaciones 3-capas completas que residen en diferentes puntos geográficos. De igual modo, no es lo mismo integrar dos sistemas de una determinada empresa a través de la red local, que integrar dos sistemas de compañías diferentes a través de Internet.

3.1. Comprender el Concepto de Middleware

Las ***plataformas de middleware*** cumplen diferentes roles y se presentan en varias formas. Antes de comenzar a revisar cada una de sus formas, resulta útil entender los principios comunes a todas las plataformas de middleware.

3.1.1. Middleware como una Abstracción de Programación.

Los lenguajes de programación, y casi cualquier software para desarrollo, evoluciona para obtener mayores niveles de abstracción. Es decir, estos niveles más altos de abstracción buscan:

- ocultar los detalles del hardware y la plataforma,
- contar con primitivas e interfaces de alto poder expresivo,
- dejar las tareas más complejas a aplicaciones de soporte o aplicaciones intermedias (compiladores, optimizadores, sistemas de balanceo automático de carga, sistema de particionado de datos, sistemas de nombres y ubicación, etc.),
- reducir el número de errores de programas,
- reducir el costo de desarrollo y mantenimiento de las aplicaciones, promoviendo su reuso y portabilidad.

Los sistemas de middleware son, en principio, un conjunto de abstracciones de programación creadas para facilitar el desarrollo de sistemas distribuidos complejos. Para entender cualquier plataforma de middleware, se requiere entender este modelo de programación. Desde este modelo de programación se pueden determinar, en una primera aproximación, las limitaciones, el rendimiento general y la aplicabilidad de cualquier tipo de middleware. El modelo de programación subyacente determinará cómo evolucionará la plataforma cuando las nuevas tecnologías evolucionen.

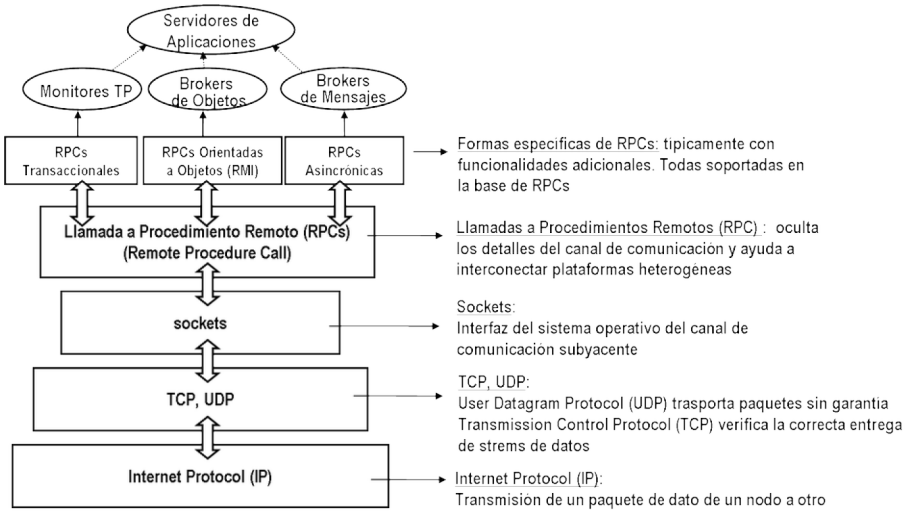

Figura 3.1: Genealogía de Middleware.

A modo de ejemplo, se puede considerar el desarrollar una aplicación que tiene dos partes de código que se ejecutan en sendas máquinas diferentes. A tales efectos, en un enfoque muy básico, se podría pensar en *sockets* para establecer canales de comunicación entre las dos partes de la aplicación y usar dichos canales para el intercambio de información. Para lograr esto, se debe considerar los siguientes puntos:

1. El *canal de comunicación* en sí. Es decir, se debe desarrollar todo el código que crea el canal y para el tratamiento de los posibles errores o fallas de comunicación que pueden ocurrir en el canal sobre la red.

2. Un *protocolo de comunicación*. Se necesita elegir y/o desarrollar un protocolo que especifique cómo debe realizarse el intercambio de información entre las partes en forma ordenada.

3. Un *formato de datos* para intercambio. Es requerido crear un formato de intercambio de la información entre las dos partes de la aplicación, de modo que pueda ser entendida correctamente por emisor y receptor.

4. La *aplicación* en sí. Luego de lograr cada uno de los puntos anteriores, se iniciará el desarrollo de la aplicación distribuida que usará el canal de comunicación. Esta aplicación deberá ser capaz de manejar mensajes entre extremos del canal, fallas en alguno de los extremos y las tareas de recuperación luego de una falla.

Se puede evitar mucho de este trabajo si se utilizan las herramientas y abstracciones de middleware. Para ejemplificar esta idea, se pueden citar las RPCs que se explicarán en detalle más delante. El esquema básico de RPCs oculta los detalles del canal de comunicación detrás de una interfaz de programación que se ve exactamente igual que una llamada a procedimiento normal. Lo único que hay que hacer es reformular la comunicación entre extremos del canal como llamadas a procedimientos.

El sistema de RPCs está construido sobre la interfaz de comunicación provistas por el sistema operativo (sockets), la cual a su vez esta construida sobre los protocolos de comunicación.

Si se desea que el middleware soporte también el manejo de errores y fallas, se podría proponer a los sistemas de RPCs transaccionales. Con *RPCs Transaccionales* se pueden abstraer de los detalles del estado de una invocación o iteraciones entre las partes, si un error ocurre durante las mismas. Se garantiza que cualquier efecto colateral será eliminado en el proceso de recuperación de la iteración.

Si bien el manejo de transacciones elimina todo efecto colateral de una transacción abortada no finalizada, no se asegura que la parte que falló recuerde en qué punto de ejecución lo hizo. Usando esquemas de *RPCs con persistencia*, es posible recuperar el estado de ejecución de una tarea previo a una falla.

De esta forma, los sistemas de middleware pueden volverse más complejos y abarcar cada vez más funcionalidades. Es entonces opción del desarrollador elegir entre codificar toda la funcionalidad desde cero o utilizar plataformas de middleware existentes. Desde el punto de vista de la programación, es mejor si se opta por un middleware que pueda encargarse de gran parte de la funcionalidad. Pero es importante destacar, que a medida que los middleware implementan más funcionalidades posibilitando más abstracciones, se vuelven más complejos y costosos en su infraestructura.

En la práctica, todas las aplicaciones distribuidas no triviales usan middleware. La clave de esta opción de diseño es utilizar estrictamente las funcionalidades de middleware necesarias, no más. Esto no siempre es una opción fácil ya que los sistemas middleware están ligados completamente a plataformas determinadas. Una solución a este problema de "altas dosis de middleware" es utilizar soportes menos sofisticados que puedan ser extendidos según los requerimientos. Si bien esta alternativa genera un mayor costo de desarrollo, es más adecuada frente al uso de una suite de middleware más completa, ligada a una plataforma determinada, la cual ofrece mucho más de lo que se necesita realmente. Es necesario notar que las ventajas que un sistema de middleware provee en términos de abstracción para programación, reclaman su contra parte en la complejidad y costo de la infraestructura que soporta tales abstracciones.

3.1.2. Middleware como Infraestructura.

Es obvio que si un sistema de middleware provee abstracciones de programación, necesitan de una infraestructura que implemente tales abstracciones. En la mayoría de los casos los sistemas de middleware requieren de gran cantidad de código para funcionar. La tendencia actual indica que aumenta la complejidad de la infraestructura del software a medida que el producto crece en la sofisticación en las abstracciones de programación que ofrece y en las capas que agrega (por ejemplo, algunas middlewares ofrecen un servidores web como parte de su infraestructura).

Una infraestructura tipo de un middleware está dividida en dos grandes grupos de software:

1. el *soporte al desarrollo de aplicaciones distribuidas* y, por otro lado,

2. el *soporte de ejecución* o run-time necesario para ejecutar tales aplicaciones.

Por ejemplo, las RPCs poseen infraestructura para implementar el *lenguaje de definición de interfaces (IDL Interface definition language)* y la interfaz con el compilador para el desarrollo, y además cuenta con librerías que hacen posible las llamadas remotas en tiempo de ejecución.

Cuando una abstracción de programación, como es el caso de las RPC, es usada y adoptada ampliamente, comienza a ser extendida y ampliada en diferentes formas. Cada extensión, orientada a la programación o a la ejecución de aplicaciones distribuidas, trata de ser lo más general para ser empleada en una amplia variedad de escenarios. Esto dota al middleware de abstracciones de programación más expresivas y más fácil de usar. Pero también, hace al sistema más complejo y su curva de aprendizaje más empinada.

Por ejemplo, si un cliente de RPC requiere autenticarse antes de realizar llamadas remotas, es necesaria toda una infraestructura de mecanismos de autenticación y validación de usuarios en el middleware.

Otro ejemplo sería contar con ligadura dinámica entre procesos en una llamada remota, esto exige que la ubicación física del código del proceso invocado se establezca en tiempo de ejecución. Para soportar tal funcionalidad es necesario contar con un *servidor de nombres* y un *servicio de directorio*, ambos deben estar en ejecución al momento de realizarce la llamada remota.

Usuarios más sofisticados pueden requerir soporte para ejecución multihilo *(multi-threading)*, autenticación automática, transacciones, RPC asincrónicas y otras funciones. Cada una de estas prestaciones exigen niveles de complejidad, tanto para la implementación de tales abstracciones de programación, como para la ejecución de los servicios requeridos en la infraestructura subyacente.

3.1.3. Tipos de Middlewares.

Brevemente se describen las diferentes formas en que evolucionaron las plataformas de middlewares:

Sistemas Basados en Llamadas a Procedimientos Remotos (RPCs): es la forma más básica de middleware. Su infraestructura provee los mecanismos esenciales para producir invocaciones a procesos remotos, como si fueran locales, en una forma uniforme y transparente. Hoy en día, los sistemas RPCs son la base de la mayoría de las otras plataformas de middleware.

Monitores de Procesamiento de Transacciones *(TP monitors)*: son la forma más antigua y mejor conocida de middleware. Es la tecnología más confiable, más testeada y más estable en los campos de integración de aplicaciones. En una forma muy simplificada, los Monitores de Transacciones pueden verse como sistemas de RPCs con manejo de transacciones.

Brokers de Objetos *(Object Brokers)*: tienen su origen como adaptación de los sistemas RPCs al paradigma de Orientación a Objetos, es decir, en vez de soportar llamadas a procedimientos remotos, los Brokers de Objetos soportan mensajes a objetos remotos. Si bien esta forma de middleware ofrece una especificación más avanzada, su implementación no difiere mucho de las plataformas RPCs.

Monitores de Objetos *(Objects Monitors)*: es la convergencia de las tecnologías de los Monitores de Transacciones y de Brokers de Objetos. Se puede ver a esta tecnología como la extensión de los Monitores de Transacciones hacia los lenguajes orientados a objetos proveyendo interfaces para tal caso.

Middleware Orientado a Mensajes MOM *(Message-oriented Middleware)*: esta tecnología surge al verse que no siempre son necesarias las interacciones sincrónicas entre partes remotas de una aplicación. Inicialmente se solucionó tal requerimiento con sistemas *RPCs asincrónicos*. Posteriormente, los Monitores de Transacciones extendieron su soporte a *Sistemas de Colas de Mensajes Persistentes (Persistents Message Queuing Sistems)*. A ciento punto, los diseñadores se dieron cuenta que los Sistemas de Colas de Mensajes resultaron muy útiles por sí sólos; de esta forma llegaron a ser una plataforma de middleware separada, llamada comúnmente *Middleware Orientado a Mensajes* o simplemente *MOM*. Tales sistemas permiten acceso transaccional a las colas de mensajes, persistencia de colas y primitivas para leer y escribir colas de mensajes locales y remotas.

Brokers de Mensajes *(Message Brokers)*: es una forma distinta de Middleware Orientado a Mensajes, que tiene la capacidad de transformar y filtrar mensajes que se mueven entre colas. Pueden seleccionar dinámicamente el receptor de cada mensajes según alguna política de filtrado que puede ser por tópico o contenido. En sí, los Brokers de Mensajes son sistemas de colas, con la diferencia que la Lógica de Aplicación puede vincularse con la gestión de las colas, por consiguiente permite implementar interacciones más sofisticadas en forma asincrónica; como ejemplo de esta funcionalidad, está la implementación del esquema de comunicacion publicar/suscribir.

Middleware basado en Agentes *(Agents Based Middleware)*: son las plataformas que permiten llevar las funcionalidades de middleware a entornos de computación ubicua. Los desafíos planteados por la computación móvil y ubicua exigen que interoperen una gran variedad de dispositivos, de diferentes capacidades de procesamiento y almacenamiento, en ambeintes cableados e inalámbricos. Los fundamentos de la computación ubicua promueven la interacción y cooperación entre los dispositivos, lo que exige que nuevos paradigmas de desarrollo sean el eje de diseño de las aplicaciones distribuidas. A tal efecto, la mayoría de estas plataformas adoptan el paradigma de agentes.

3.1.4. Convergencia de Tecnologías Middleware

A menudo se ha argumentado que coexisten muchos middleware con funcionalidades superpuestas e incompatibles [455]. Esto no solamente se refiere a las abstracciones de programación, sino también a la infraestructura subyacente. Con el ánimo de aprovechar las diferentes interfaces de programación, diferentes plataformas de middleware son usadas simultáneamente para la integración de sistemas; por consiguiente, diferentes infraestructuras se ejecutan concurrentemente. Lo irónico de esta situación, es que estas plataformas de middleware poseen un porcentaje significativo de sus infraestructuras que es igual para todas (basadas en RPC generalmente), pero son incompatibles entre sí por estar adaptadas y especializadas para cada producto comercial. Como resultado a este problema de redundancia en infraestructura de software, existen dos tendencias marcadas que se pueden observar en la evolución de tales plataformas. En un extremo está la consolidación de plataformas complementarias a un núcleo común; por otro lado, emergen las suites de productos que ofrecen para un único ambiente, variadas formas de middlewares.

Los Monitores de Objetos son un ejemplo de consolidación. Estas plataformas combinan las características y performance de los Monitores de Transacciones con interfaces orientadas a objetos de los Brokers de Objetos. Como se indicó en el capítulo anterior, es más eficiente agregarle al núcleo de los monitores de transacciones interfaces y adaptadores para el tratamiento de objetos remotos.

Por otro lado, existen muchos ejemplos de suites de productos que combinan todas las plataformas de middleware ofrecidas por una misma compañía de software. La idea es simplificar las arquitecturas multicapas recurriendo al uso de sistemas que están específicamente diseñados para trabajar juntos. Si bien éste es un proceso en pleno desarrollo, fue bien recibido por los desarrolladores para afrontar los problemas de integración que han encontrado en la práctica. El nivel de integración en este punto no es perfecto, pero es razonable esperar que se mejoren a medida que aparezcan tales plataformas multipropósito de middleware. (Ejemplos de esta integración son los Servidores de Aplicaciones, que se explicacarán en la Sec. 4.3)

3.2. Sistemas RPCs y Tecnologías Relacionadas

Las ***Llamadas a Procedimientos Remotos*** *o* ***RPCs*** *(Remote Procedure Calls)* son la plataforma de middleware más simple y constituye la infraestructura básica de la mayoría de las formas de middleware actuales.

3.2.1. Contexto Histórico

Las RPCs fueron introducidas en los inicios de los 80s por Andrew D. Birrell y Bruce J. Nelson [63] como parte de su trabajo en el entorno de programación Cedar. El artículo original presentaba a las RPCs como una forma transparente de realizar llamadas a procedimientos ubicados en otras máquinas.

La idea de las Llamadas a Procedimientos Remotos es bastante simple. Su paradigma esta basado en el concepto de llamada a procedimiento en un lenguaje de programación. La semántica de una RPC es casi igual a la semántica de una llamada a procedimiento tradicional. Una llamada a procedimiento normal tiene lugar entre dos procedimientos en un proceso único en el mismo espacio de memoria; en sí, las llamadas entre procedimientos son el mecanismo mejor conocido para transferir el control e intercambio de datos entre partes de un algoritmo que se ejecuta en una única máquina. Por otro lado, las RPCs tienen lugar entre un proceso cliente en una máquina y un proceso servidor en otra máquina, siendo ambas máquinas interconectadas por una red [34]. Es así que resulta natural extender este mecanismo para proveer transferencia de control y datos dentro de una red de computadoras.

Cuando un procedimiento remoto es invocado, el ambiente de donde se realiza la llamada (el *invocante*) es suspendido, los parámetros son pasados a través de la red al ambiente que contiene

al procedimiento, para luego ejecutarse dicho procedimiento en el ambiente invocado. Cuando el procedimiento concluye y produce resultados, estos son retornados al ambiente invocante, el cual reasume su ejecución, como si se tratara de una llamada a procedimiento local.

Hay varias ventajas detrás de esta simple idea:

- Para empezar, las RPCs tienen una semántica simple y clara lo que facilita el diseño y construcción conceptual de aplicaciones distribuidas
- Otra es la eficiencia. Las RPCs son lo suficientemente simples para ser implementeadas en una infraestructura liviana que garantice rapidez en la comunicación.
- Por último, las RPCs son genéricas, están basada en un concepto bien conocido de programación: el *procedimiento* [246]. Esto hace posible la construcción de aplicaciones distribuidas sin tener que cambiar de paradigma de programación o de lenguaje.

Luego de su presentación en los 80s, las RPCs llegaron a ser el fundamento de los sistemas 2-capas. En particular, se toma de esta especificación la idea de *cliente* y *servidor*. También introducen los conceptos de lenguajes de definición de interfaz IDL, servicio de directorio y de nombres, ligadura dinámica de procesos remotos, interfaz de servicio, y otros.

La fortaleza de las RPCs consiste en proponer una forma clara de tratar con la distribución en sistemas informáticos. Inicialmente las RPCs fueron implementadas como una colección de librerías; para luego, con el incremento en funcionalidad, desarrollarse como una plataforma de middleware. En síntesis, para que un programa se transforme en un componente distribuido, solamente se requiere que sea compilado y vinculado con el correcto conjunto de librerías de RPCs.

3.2.2. Esquema de Trabajo

El modelo de ejecución de una RPC es una interacción entre varios componentes de software. Por un lado está la ***aplicación cliente***, que es aquella que realiza la invocación remota; y como contraparte, la ***aplicación servidor***, es la que contiene la rutina invocada.

Entre estas dos aplicaciones existes varios componentes que son propios de la plataforma de RPCs. Por cada aplicación cliente, existe un ***stub cliente***, que vincula a la aplicación cliente con la plataforma RPC. Las ***librerías de servicios de runtime de RPCs***, tanto en la máquina que contiene el cliente como en aquella que aloja al servidor, son las responsables de interconectar ambos extremos de la comunicación y ocultar los detalles del canal de comunicación. De la misma forma, existe un ***stub de servidor*** que vincula a la aplicación que contiene la rutina a invocar, con los servicios de RPCs del servidor. Ambos stubs ocultan la complejidad de la distribución y la heterogeneidad de los sistemas que se vinculan por una llamada a procedimientos remota. Los stubs son los encargados de transformar la llamada y los argumentos en mensajes; y además, de convertir los datos de la representación local a una representación de datos intermedia comprendida por ambos stubs.

La figura 3.2 ilustra la operación básica de una llamada a procedimiento remoto. Una aplicación o proceso cliente realiza una llamada a un stub cliente. El stub se encarga de formatear[1] y serializar[2] apropiadamente los datos. La preparación del mensaje es similar a la descripta en la Sec. 2.3.1. El stub cliente convierte los argumentos de entradas de la representación de datos local a una representación de datos común (ver más adelante Sec. 3.2.5). Para lograr localizar el servidor y concetarse a éste, el stub cliente invoca al servicio de runtime de RPCs, que generalmente es una librería de rutinas que soportan la funcionalidad del stub cliente. El runtime

[1] El término original es *marshaling*, que significa ordenar o empaquetar un grupo de datos en un formato común de mensajes de forma tal que sean entendido por el remitente.

[2] La serialización consiste en transformar un mensaje con datos estructurados y/o complejos en una cadena de caracteres o bytes para ser enviada a través de un canal de comunicación.

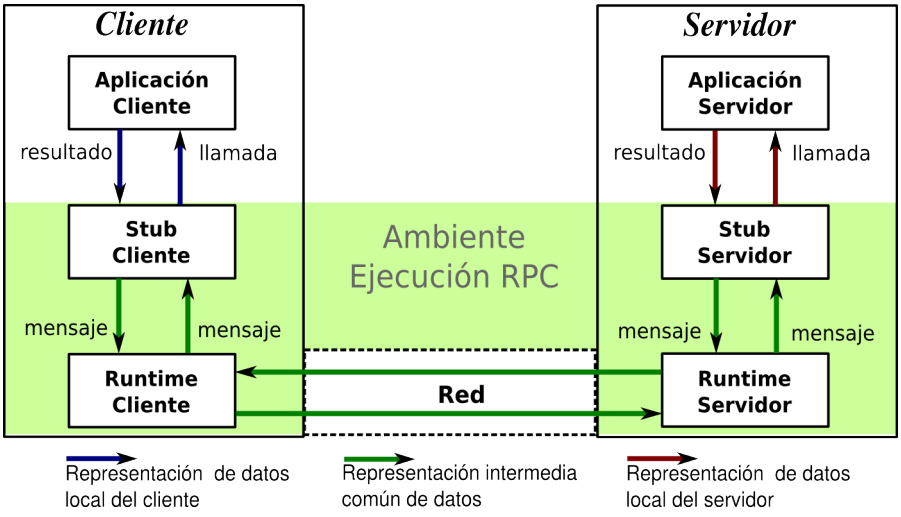

Figura 3.2: Modelo de Trabajo de RPCs.

del cliente transmite el mensaje con los argumentos al runtime del servidor, que es una librería que soporta la funcionalidad del stub servidor. En este punto, el servicio de runtime del servidor propaga la llamada al stub servidor el cual toma los argumentos de entrada y los convierte desde la representación intermedia a la representación que utiliza el servidor. Por último invoca a la aplicación servidor la cual realiza el procesamiento.

Cuando la aplicación servidor es completada, se retorna al stub servidor los resultado como argumentos de salida. El stub servidor convierte estos argumentos desde la representación local del servidor a la representación intermedia para transmitirlos por la red encapsulados en un mensaje el cual es pasado al runtime del servidor. El runtime del servidor transmite el mensaje al runtime del cliente, el cual lo pasa al stub cliente. Este extrae los argumentos de salida del mensaje y los retorna, traduciéndolos a la representación local, a la aplicación cliente.

3.2.3. Esquema de Desarrollo

El desarrollo de aplicaciones distribuidas basadas en RPCs tiene un método bien definido. Se considera que las RPCs conforman un métodología madura con varias especificaciones e implementaciones [34]. Para explicar la forma de trabajo del esquema o modelo subyacente de las RPCs, se pondrá como ejemplo el desarrollo de un servidor que implementa un procedimiento que va a invocarse remotamente por un único cliente (véase Fig. 3.3).

El primer paso del desarrollo es **definir la *interfaz del procedimiento***. Esto se realiza usando un ***Lenguaje de Definición de Interfaz*** o ***IDL*** [266] que provee una representación abstracta del procedimiento en términos de parámetros de entradas y salidas, puramente sintáctico, sin semántica. Es decir, es una especificación del servicio ofrecido por el servidor. Con las descripciones en IDL, se procede a desarrollar ambos componentes, cliente y servidor. Ejemplos de IDLs son: RPCL de Sun [470, 449], DCE IDL de X/Open [493] e IDL de CORBA [364].

El segundo paso es **compilar la descripción IDL**. Cualquier sistema de RPCs o middleware basado en RPCs provee un compilador de interfaces IDL. Típicamente, la compilación de las interfaces IDL produce lo siguiente:

- ***Stub Cliente***: cada signatura en la definición IDL genera un *stub cliente*.
- ***Stub Servidor***: es similar a la naturaleza de los stubs cliente, pero implementa la invocación del lado del servidor.
- ***Plantillas de código y referencias***: en muchos lenguajes de programación, es necesario definir, al momento de compilación, los procedimientos que van a ser usados. El compilador

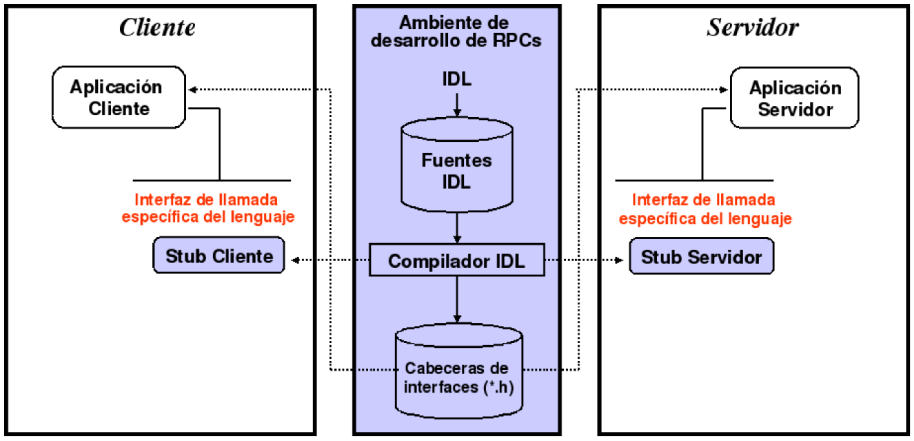

Figura 3.3: Esquema de desarrollo de aplicaciones distribuidas usando RPCs.

IDL asiste a esta tarea generando todos los archivos necesarios para el desarrollo. Por ejemplo, las primeras implementaciones de RPCs fueron para el lenguaje C. A la hora de compilar las interfaces IDL, además de generar los stubs, se generaban los archivos cabeceras de C (archivos *.h) necesarios para la compilación del cliente y servidor. Los compiladores actuales ofrecen además plantillas de los procedimientos, en las cuales sólo se define la signatura, relegando la implementación al desarrollador.

Los sistemas de RPCs pueden encargarse de todos los detalles de manejo del canal de comunicación. Si se desea más control sobre el estado de la interacción, los sistemas de RPCs ofrecen una amplia gama de interfaces, que van desde las más abstractas y transparentes, hasta aquellas que permiten configurar cómo debe hacerse la interacción en el canal de red.

La interfaces de programación pueden ofrecer una gama de primitivas, de acuerdo a la complejidad en interacción que se quiere soportar. La semántica de tales primitivas depende fuertemente del manejo de las fallas que pueden ocurrir en la llamada remota. Las posibles fallas pueden ser [400] (véase Fig. 3.4):

- *Pérdida del mensaje de petición* (1): si un mensaje fue perdido, el cliente debe retransmitir el mensaje despues de un tiempo de espera.
- *Pérdida del mensaje de respuesta* (2): el proceso fue ejecutado en el servidor, pero se perdió el mensaje con la respuesta al cliente.
- *Servidor caído* (3): si el servidor falla mientras estaba completando la ejecución de una llamada, debe ser capaz de quitar cualquier efecto colateral que se haya producido por la ejecución parcial no finalizada.
- *Cliente caído* (4): cuando un proceso cliente cae mientras espera la respuesta de una RPC, se la denomina *invocación huérfana*. En este caso debe definirse si el servidor debe enviar o no la respuesta.

A la luz de estas posibles fallas, una primera clasificación de las semánticas de primitivas para RPCs consiste en dividirlas en *primitivas confiables y no-confiables* [246]. Las primitivas que no tienen en cuenta las posibles fallas descritas, se dicen que son no-confiables. Una ***primitiva no-confiable*** (véase Fig 3.5(a)) simplemente coloca el mensaje en la red y no garantiza la entrega correcta del mismo y no provee retransmición automática si un mensaje se pierde. Para tratar con los problemas de fallas se requieren ***primitivas confiables*** (véase Fig 3.5(b)). En la comunicación interprocesos confiable, la primitiva de envio maneja la contingencia de pérdida de mensaje usando retransmisión y confirmaciones basándose en tiempos de espera. Esto implica que cuando se genera una RPC, el sistema se asegura que el mensaje fue recibido y confirmado.

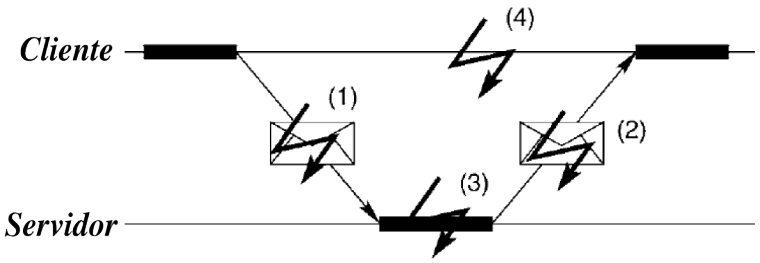

Figura 3.4: Posibles fallas en una llamada a procedimiento remoto.

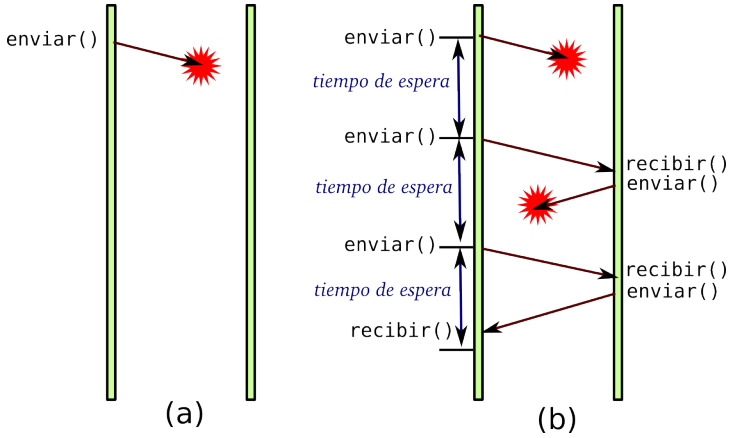

Figura 3.5: Semántica de pasaje de mensajes: (a) no-confiable, (b) confiable.

Para lograr confiabilidad en la comunicación, el uso de tiempo de espera exige que se retransmitan los mensajes si no se reciben los mensajes de confirmado. La semántica de una llamada a procedimiento remoto confiable difiere de acuerdo a lo que realiza el servidor una vez que recibe múltiples copias del mismo mensaje. Las diferentes semánticas de RPCs puede ser:

- ***"por-lo-menos-una-vez"*** *(At-least-once)*: en este tipo de RPCs el servidor repite el procesamiento del mensaje, aún si sólo se requiere una sola ejecución. En este caso, la primitiva confiable asegura que la petición sea ejecutada al menos una vez. El problema de este tipo de primitiva se presenta en los casos en los cuales cada ejecución cambia el estado interno del servidor, siendo lo deseable que se ejecute una sola vez. Son adecuadas para los casos en los cuales la invocación no deja efectos en el servidor. La primitiva "por-lo-menos-una-vez" es relativamente fácil de implementar.

- ***"exactamente-una-vez"*** *(Exactly-once)*: en muchos casos, la repetición de la ejecución de una petición puede destruir la consistencia de la información. De aquí la necesidad de una primitiva que asegura que la petición fuese procesada una y sólo una vez. Esta es la primitiva más deseada, pero la más dificil de implementar.

Además de esta clasificación de primitivas, de acuerdo a la confiabilidad en el tratamiento de la comunicacion entre procesos, es posible hacer otra distinción entre primitivas teniendo en cuenta el comportamiento deseado de la RPC. De esta forma se tiene las primitivas:

- ***Llamada a Procedimiento*** *(Procedure Call)*: invocación al procedimiento remoto con comportamiento petición/respuesta, como lo indica la Fig. 3.2.

- ***Llamada en Broadcast*** *(Procedure Call using Broadcast)*: una RPC en broadcast provee la capacidad para la aplicación de hacer llamadas a más de un servidor con una sola RPC.

Es decir, la llamada se transmite o se *difunde* de un nodo emisor a una multitud de nodos receptores de manera simultánea, sin necesidad de reproducir la misma llamada nodo por nodo. Ejemplo de este comprtamieto puede darse cuando un programa en un punto de venta, envia una RPC en Broadcast a distintos servidores indicando una venta.

- ***Llamada de no-respuesta*** *(No-response Procedure Call)*: una RPC de no-respuesta provee la capacidad para no exigir respuesta del servidor. Un ejemplo de tal comportamiento puede ser un proceso que monitorea una determinada actividad. El monitor hace una RPC de no-respuesta a un servidor que mantiene una bitácora de eventos de la actividad que se monitorea. El monitor no necesita respuesta del servidor.

Además de estas primitivas básicas, un sistema de RPC puede tener primitivas para registrar y publicar servidores, para consultar direcciones de servidores, y otras.

3.2.4. Ligaduras en las RPCs

Para lograr una llamada remota es necesario localizar y vincularse al servidor que aloja al procedimiento remoto. El ***proceso de ligadura*** *(Binding)* hace que el cliente establezca una asociación local con un determinado servidor remoto con el objetivo de producir la invocación remota. Este proceso puede establecer ligaduras *estáticas* o *dinámicas*.

Para entender la idea de ligadura entre distintos procesos se explicará algunas ideas fundamentales. Para que dos procesos, ejecutándose en máquinas separadas, puedan intercambiar datos, una *asociación* necesita ser formada. La asociación es una quíntupla = <*protocolo, dirección local, proceso local, dirección remota, proceso remoto*>.

Como se explicó, el *protocolo* es el mecanismo de transporte (usualmente se utiliza TCP o UDP) que sirve para mover información entre computadoras. Esto es la parte que necesita ser común entre ambos extremos. Los pares dirección/proceso define cada uno de los extremos de la comunicación.

El término *dirección* se refiere a una dirección de red asignada a una máquina. Generalmente es una dirección IP. El término *proceso* se refiere a un indentificador requerido para transportar los datos al proceso correcto una vez que los datos lleguen a la máquina. Este número no es un identificador real de proceso como puede ser un PID de un SO basado en UNIX; generalmente es un número estipulado y se lo conoce como ***puerto***. Como se verá más adelante (véase Sec. 4.1.2), existen asignaciones estándares de números de puertos a determinados servicios, como por ejemplo el puerto 23 para telnet, el 21 para FTP, etc.

La diferencia sustancial entre ligadura estática y ligadura dinámica, está fundada en el momento cuándo esta asociación se establece. En un esquema de ligadura estática, dicha asociación se efectúo previamente antes de la invocación RPC. Generalmente cuando se compila la RPC. Por el contrario, en un esquema dinámico, la asociación se establece en el mismo momento de la invocación.

En las ***ligaduras estáticas***, el stub del cliente tiene descripto en su código la ubicación del servidor dónde reside el proceso remoto. Cuando el cliente realiza la invocación, el stub simplemente redestina la llamada al servidor.

La principal ventaja del ligado estático es su simpleza y eficiencia. Además de los módulos de comunicación, y los servicios de runtime de RPCs, solamente se requiere una aplicación o servicio ***despachante***, que de acuerdo al puerto del proceso, asocia la invocación a un determinado stub servidor. En otras palabras, el despachante mapea puertos con procesos (véase Fig. 3.6)

La desventaja de la ligadura estática es que el cliente está altamente acoplado con el servidor; a tal punto que si el servidor falla, el cliente también. Si el servidor cambia de ubicación (por mantenimiento, actualizaciones, etc), el cliente debe ser recompilado con la nueva ubicación del servidor. Por otro lado, con esta configuración no es posible crear una estructura de servidores redundantes ya que los clientes están atados a servidores individualizados. Por consiguiente, el balanceo de carga no es posible en este contexto; si se tiene varios servidores que replican

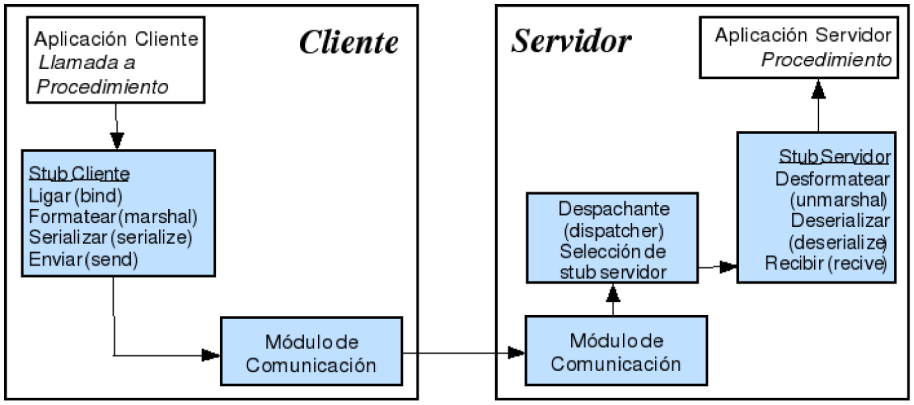

Figura 3.6: Funcionamiento de RPCs con binding estático.

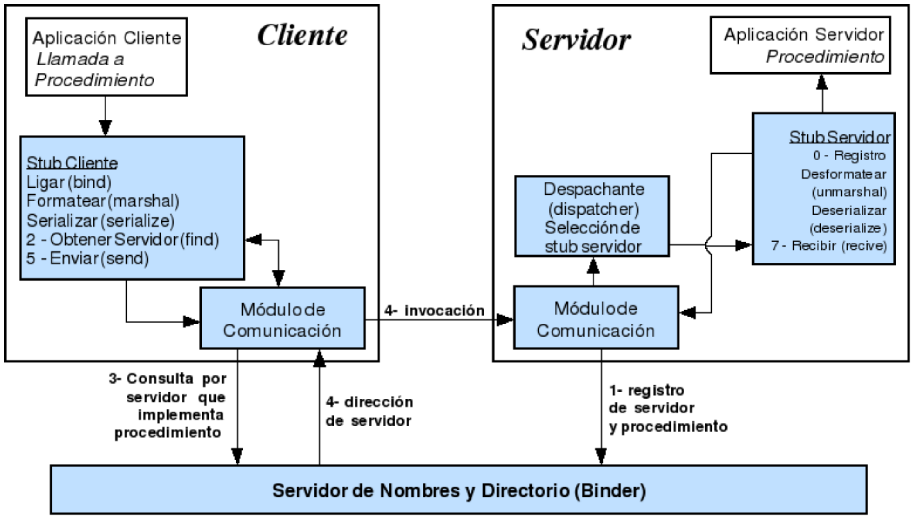

Figura 3.7: RPC con ligadura dinámica. Los pasos 0 y 1 registran el servidor y servicio en el servidor de nombres y directorio. A solicitud de una llamada remota, los pasos 2, 3 y 4 localizan al servidor y lo ligan con el stub cliente. En pasos 5, 6 y 7 se realiza y finaliza la llamada.

procedimientos, se generan situaciones en las cuales existen servidores atendiendo a demasiadas invocaciones, mientras otros permanecen inactivos.

Estas desventajas impulsaron el desarrollo de las ***ligaduras dinámicas***. En el ligado dinámico el cliente recurre a un servicio especializado en localizar al servidor apropiado. Este servicio es generalmente llamado ***Servidor de Nombres y Directorio*** *o* ***NDS*** *(Name and Directory Server)* también conocido como el *binder* en la primeras versiones de RPCs. El binder o el NDS conforman una nueva capa en la estructura del sistema informático; es decir, se agrega un nuevo nivel de indirección para ganar flexibilidad a costo de performance. El servidor de nombres y directorio es responsable de obtener la dirección física del ambiente destino basándose en la signatura del proceso invocado. De esta forma, el cliente realiza la invocación pasándole en control a su stub. El stub cliente solicita al NDS una dirección de un servidor adecuado en dónde ejecutar la llamada. El NDS responde con la dirección del servidor; con esta dirección el stub cliente se liga con el servidor y termina de realizar la invocación (véase Fig. 3.7)

El uso de ligaduras dinámicas trae consigo importantes mejoras a la interacción de RPCs. Es posible que los servidores NDS lleven un registro de las direcciones de servidores enviadas como respuesta, y con esta información realizar una correcta gestión del balanceo de carga entre servidores que contienen los mismos procedimientos. Por otro lado, si un servidor cambia de ubicación, solamente es necesario actualizar los registros del directorio del NDS para que las invocaciones sean redestinadas a la nueva dirección. Estas mejoras logran desacoplar el cliente

con el servidor, dando como resultado mayor flexibilidad cuando el sistema está operable. El costo de tal flexibilidad exige contar con la infraestructura necesaria para soportar el NDS, un protocolo para interactuar con éste, y primitivas para registrar y actualizar información de los servidores y procedimientos.

El servidor de nombres y directorio utiliza la especificación IDL de procedimientos para realizar el ligado dinámico. Es decir, la ligadura es establecida en base a la signatura del procedimiento. Sin embargo, pueden existir otros criterios, posiblemente no funcionales, para establecer ligaduras. Es común que a los servidores de directorios que proveen mecanismos más sofisticados de selección se los denomine ***traders***.

En principio la infraestructura del servidor de nombres y directorio está oculta al desarrollador, ya que el stub servidor registra el servidor cuando éste se inicia; por otro lado el stub cliente interactúa con el NDS cuando la llamada es producida. Tal enfoque es la base de la separación de los desarrollos de los clientes y servidores.

3.2.5. RPCs y la Heterogeneidad

La principal motivación del génesis de los sistemas RPCs era servir como mecanismos para interconectar sistemas heterogéneos (de diferentes plataformas). Como la interacción entre sistemas remotos es realizada por los stubs, es posible desarrollar stubs más complejos que posibiliten a los programadores construir clientes en plataformas diferentes a las usadas para los servidores. Es decir que los stubs no sólo ocultan la distribución, sino que también la heterogeneidad.

El problema de la heterogeneidad es que potencialmente existen varias plataformas en donde puede ejecutarse clientes y servidores, y existen varios lenguajes de programación en los que se puede programar tales clientes y servidores. En un enfoque muy básico, se podría pensar en usar una dupla de stub cliente y stub servidor por cada combinación posible de plataformas y lenguajes. Supongamos que tenemos n plataformas de clientes y m plataforma de servidor. Cada cliente de una determinada plataforma debe hacer m stubs, uno para cada stub servidor; en total $n \times m$ stubs. Por otro lado, cada servidor debe hacer n stubs para poder exportar sus servicios a cada una de las plataformas cliente; es decir $m \times n$ stubs servidor. Si sumamos ambas multiplicaciones resulta $2 \times n \times m$ stubs. Como este número de componentes de software es muy alto en relación a las plataformas que se usan, una alternativa mucho más adecuada es utilizar una representación intermedia, de forma que los stubs clientes y los stubs servidor sólo tienen que traducir sus peticiones y respuestas a esa representación intermedia (véase Sec. 2.3.1). Es decir, los stubs se deben desarrollar teniendo solamente en cuenta el formalismo de representación intermedia, desacoplándose de las posible plataformas que los puedan utilizar en cada llamada remota. Este enfoque logra que la heterogeneidad sea sólo una cuestión de desarrollar stubs en las plataformas requeridas, es decir se necesitan $n + m$ stubs.

Las RPCs usan las especificaciones IDLs, no sólo para definir la interfaz del proceso remoto, sino para mapear las representación de datos propia de un lenguaje de programación a la representación intermedia usada en el sistema de RPCs. Las representaciones intermedias de IDL permite que los clientes y servidores puedan ignorar las diferencias en términos de arquitectura de hardware, sistema operativos, lenguaje de programación y representación de tipos de datos usado en cada uno de ellos. Es decir, las representaciones intermedias también son un mecanismo para intercambio de datos entre clientes y servidores.

Ejemplos de representaciones intermedias son: NCS de la OSF [256], XDR de Sun [143], Courier de Xerox [543], MiG de Mach [139] y NDR de X/Open [491].

3.2.6. Extensiones de RPCs

No todas las formas de interacción posibles en sistemas distribuidos son adecuadamente soportadas por las RPCs. En particular, estas formas de interacción fueron las que motivaron la creación de estructuras 3-capas; es decir el incremento de interfaces publicas y estables disponibles

en los servidores y la posibilidad de uso de varias de estas en un mismo cliente, exigieron una estructura e infraestructura para soportar la integración de servidores.

Los desafíos planteados por las nuevas arquitecturas distribuidas, llevaron a mejorar el modelo original de las RPCs generando extensiones del mismo. Algunas de ellas, se han convertido en una nueva forma de middlevare.

Por ejemplo, en los sistemas originales de RPCs se guardan las características de invocaciones a procedimientos locales, son sincrónicas; es decir que el programa que invocó, debe esperar a que el proceso invocado termine para proseguir. Con RPCs, el cliente puede hacer sólo una llamada permaneciendo bloqueado, hasta que el servidor retorne el resultado. En el servidor, se pueden usar múltiples hilos de ejecución para atender a múltiples clientes concurrentemente, pero cada hilo es dedicado a un solo cliente. Esta es una limitación importante al modelo de las RPCs ya que restringe la implementación de las otras formas de interacción requeridas en un sistema distribuido.

Las ***RPCs asincrónicas*** fueron una de las primeras extensiones realizadas para soportar llamadas "no bloqueantes" (véase Sec. 2.3.2). El nuevo modelo permite a los clientes enviar una petición al servicio sin esperar bloqueado la respuesta. De esta forma se retorna el control de ejecución al cliente inmediatamente luego de enviar el mensaje de petición. Es así que el cliente puede tener varias invocaciones en espera mientras prosigue su ejecución.

Las RPCs asincrónicas son relativamente fáciles de implementar. Lo que se necesita es cambiar la forma en que el stub cliente opera. En lugar de tener un único punto de entrada para invocar el procedimiento, el stub provee dos puntos de entradas: uno para realizar la invocación y otro para obtener el resultado. Cuando el cliente realiza la llamada remota, el stub toma los parámetros de la invocación e inmediatamente retorna el control al cliente. Entretanto, el stub hace la llamada al servidor y espera la respuesta. Más tarde, el cliente realiza una segunda llamada a su stub, para obtener los resultados. Si ya obtuvo los resultados, el stub cliente los ubicó en una estructura de datos común para que el cliente pueda disponer de ellos. Si el stub no ha recibido los resultados todavía, el cliente recibirá un mensaje de espera y reintento. Como se ve, los stubs actúan sincrónicamente, dando una ilusión de asincronismo a los clientes y servidores que los usan.

3.2.7. Infraestructuras para RPCs

Aun sin considerar las extensiones de RPCs, el mecanismo básico requiere algún grado de infraestructura para el desarrollo y ejecución. Algunas de las implementaciones de esta infraestructura son mínimas, como el caso de *ONC RPC (Open Network Computing Remote Procedure Call)* propuesto originariamente por Sun Microsystems [473]. Y en otros casos son más extensivas, como el *Ambiente de Computación Distribuida* o *DCE (Distribuited Computing Enviroment)* propuesto por la Open Software Fundation (OSF) [490]. Brevemente se detallará cada una de estas implementaciones para finalizar con una comparativa entre ambas infraestructuras.

ONC RPC

ONC RPC *(Open Network Computing Remote Procedure Call)* [449], es el sistema de llamadas de procedimientos remotos más ampliamente difundido. ONC RPC fue originariamente desarrollado por Sun Microsystems como parte de su proyecto de sistema de archivos distribuidos NFS (Network File System) [474, 427], por lo tanto es común referirse a esta plataforma como Sun ONC o Sun RPC.

En una forma simplificada de verla [34], la plataforma ONC RPC está basada en las convenciones de invocación utilizado en Unix y el lenguaje de programación C. La serialización de datos se realiza utilizando ***XDR*** *(eXternal Data Representation)* [143] como lenguaje de representación intermedia. Existen funciones que permiten la codificación y decodificación de los datos en

```
Program: BINOP {
    version BINOP_VERS {
        long  BINOP_VERS (struct input_args) = 1;
    } = 1
} = 300030;

struct input_args {
        long  a;
        long  b;
};
```

Cuadro 3.1: Suma de dos números. Ejemplo expresado en RPCL.

archivos XDR para ser transportados y accedidos por diferentes plataformas. ONC RPC entrega los archivos XDR usando los protocolos de transporte UDP o TCP.

Una implementación más reciente de ONC RPC es ***TI RPC*** *(Transport Independent RPC)* [470, 291]. TI RPC está disponible como parte del sistema operativo Solaris [471], aunque no fue tan popular como su predecesora por fuera del entorno Solaris. La principal diferencia entre ONC RPC y TI RPC es la posibilidad que brinda ésta última para usar diferentes protocolos de capa de transporte además de UDP y TCP.

ONC RPC cuenta con una herramienta: ***rpcgen***, que es el programa que genera los módulos de interfaz a programas remotos, es decir, es un compilador IDL. `rpcgen` compila código escrito en el ***Lenguaje RPC*** *(RPCL)* [470, 449], el cual tiene una sintaxis similar a C y cumple el rol de IDL en esta implementación. De la misma forma que se requieren describir los tipos de datos que maneja la representación intermedia en el lenguaje formal XDR, también es necesario describir los procedimientos que operan con estos datos en lenguaje formal. RPCL es una extensión del lenguaje XDR, con el agregado de las declaraciones de *"Programa"*, *"Procedimiento"* y *"Versión"* [449].

La salida por defecto de un proceso de compilación con `rpcgen` es:

- El *stub del cliente.*
- El *stub del servidor.*
- Una serie de rutinas XDR, denominadas *Filtros XDR*, cuya función es traducir los valores de los tipos de datos definidos en el archivo de cabecera a la representación intermedia XDR.
- Un *archivo de cabecera (header)* necesario para cualquier Filtro XDR. Es decir, contiene las definiciones de datos comunes para el servidor y el cliente.

En el cuadro 3.1 se muestra la definición de un procedimiento y sus parámetros para una suma de dos números. Los parámetros de entradas `a` y `b` han sido agrupadas en una estructura registro `input_args` para proveen un único parámetro de entrada para la llamada RPC. La invocación en la aplicación cliente al stub cliente, podría ser: `c = binop_add(a,b);` .

El stub cliente prepara el parámetro `input_args` a partir de los parámetros `a` y `b` de la invocación en la aplicación.

En el cuadro también muestra cómo la aplicación servidor es identificado en la plataforma ONC RPC. El programa `BINOP` se declara para ser el número `300030` y la versión del mismo (`BINOP_VERS`) es declarada la `1`. El stub cliente invoca el procedimiento en el servidor a través del servicio de runtime del cliente pasando el nombre del servidor, el número de programa, el número de versión, el nombre del procedimiento y el parámetro único.

El acceso a los servicios RPC de una máquina servidor son provistos por un servicio denominado ***mapeador de puertos*** *(portmapper)* el cual vincula puertos de comunicaciones con aplicaciones servidor. Este mapeador de puerto cumple el rol de *despachante* (véase Fig. 3.6). El mapeador de puertos espera peticiones en determinados puertos y las deriva a los correspondientes aplicaciones servidor. Cuando una aplicación servidor es comenzada, se registra en el mapeador de puertos de su máquina local. El mapeador de puerto asigna el número de puerto TCP y/o UDP para que sean usados por esa aplicación; en el ejemplo anterior se le asigna el número `300030` a la aplicación `BINOP`. De allí en más, la aplicación servidor espera y acepta conexiones en ese puerto de comunicación. Previo a la llamada remota, el cliente contacta al mapeador para obtener el correspondiente número de puerto de la aplicación servidor.

ONC RPC soporta semántica de *"exactamente-una-vez"* y de *"al-menos-una-vez"* en sus primitivas. Por razones de eficiencia, el mecanismo de comunicación usado para las llamadas remotas en ONC RPC es el protocolo simple petición/respuesta. El cliente envía un mensaje de petición al servidor, el servidor procesa la petición y retorna un mensaje de respuesta al cliente. Por si solo, este mecanismo es inadecuado para soportar la semántica "exactamente-una-vez". Aún con el uso de identificadores únicos en los mensajes de petición y respuesta (denominados *ID de Transacción* en ONC RPC), el servidor es incapáz de conocer si el cliente ha recibido la respuesta, al menos que el cliente mande un confirmado. ONC RPC soluciona el problema delegándolo al protocolo de la capa de transporte para proveer la semántica de la llamada. Es decir, para llamadas "por-lo-menos-una-vez" se utiliza UDP siendo este un protocolo no-confiable, y para llamadas "exactamente-una-vez" usa TCP dado la confiabilidad del mismo.

Por otro lado, ONC RPC es capaz de realizar llamadas en broadcast y llamadas de no-respuesta para implementar el envío de secuencias o lotes de varias llamadas al servidor, denominando a esta prestación *loteo (batching)*.

Por su parte, ONC RPC soporta tres niveles de autenticación:

- *ninguno*: establecido por defecto,
- *ID de Usuario/Grupo de Unix*: basado en el sistema de permisos de Unix,
- *RPC Segura*: siendo éste el más robusto y basado en estampillas de tiempo encriptadas en DES.

ONC RPC es un sistema optimizado y eficiente, pero tiene limitaciones como un sistema generalizado que pueda aplicarse a redes amplias o a Internet. Las implementaciones de ONC RPC en su mayoría existen en el mundo de sistemas operativos basados en Unix. Sin embargo, Microsoft da soporte a las RPCs [302] el cual puede interoperar con cualquier producto compatible.

DCE: una infraestructura para RPCs

El ***Ambiente de Computación Distribuida*** o **DCE (Distribuited Computing Enviroment)** fue propuesto por la Open Software Fundation (OSF) [490]. DCE es muy popular todavía, porque representa el paso intermedio entre las arquitecturas cliente-servidor y los sistemas 3-capas.

El Ambiente de Computación Distribuida de la OSF es un estándar neutral que lo conforman un conjunto de tecnologías orientadas a la computación distribuida [493]. DCE provee un ambiente completo para el desarrollo y ejecución de aplicaciones distribuidas [230]. Provee servicios de seguridad para proteger y controlar el acceso a datos, el servicio de nombres que facilita la tarea de encontrar recursos distribuidos, y un modelo altamente escalable para organizar una amplia gama de usuarios, servicios y datos. DCE se ejecuta en la mayoría de las plataformas y está diseñado para soportar aplicaciones distribuidas en ambientes con hardware y software heterogéneo.

DCE es una plataforma completa que brinda una implementación estándar que las compañías de software pueden usar y extender según los requerimientos de sus productos. La idea fundamental, en coherencia con los principios de la OSF, es lograr que los diferentes productos que usan la implementación básica, sean compatibles.

DCE RPC *(DCE Remote Procedure Call)* es uno de los servicios que ofrece la plataforma DCE. DCE RPC fue un intento de estandarizar RPC. DCE RPC fue basado en su predecesor NCS RPC (Network Computing System RPC) de HP/Apollo [256] estandarizado luego por la OSF; además, contiene elementos de ONC RPC. Sin embargo, no provee compatibilidad plena con estos [34].

El lenguaje de definición de interfaces es el ***DCE IDL*** [517, 490, 230]. Como DCE RPC es descendiente de NCS RPC, provee herramientas de conversión para traducir definiciones hechas en NIDL (NCS IDL) a DCE IDL.

En la declaraciones de las interfaces de procedimientos, DCE IDL soporta declaraciones de múltiples parámetros, a diferencia de ONC RPC. Tales parámetros pueden ser de entrada, de salida y de entrada/salida. DCE IDL también soporta todos los tipos de datos del lenguaje C, el tipo de dato **`handle_t`** para datos no traducidos, y algunos otros.

Por ejemplo, IDL soporta el tipo de dato **`pipe`**. En algunos escenarios de computación distribuida, es necesario que cliente y servidor intercambien grandes cantidades de datos. La forma de hacer esto tradicionalmente es a través de la generación de canales entre cliente y servidor como resultado de una RPC. Como ONC RPC no posee soporte para tal tipo de aplicación y tales comunicaciones, la gestión del canal debería programarse completamente desde cero. DCE RPC provee las capacidades necesarias para tales aplicaciones gracias al tipo de dato de `pipe`. Una instancia de tipo `pipe`, puede ser un parámetro de entrada (un canal desde cliente hacia el servidor), de salida (un canal desde el servidor al cliente) o de entrada/salida (una canal de dos sentido entre clientes y servidor).

DCE RPC provee los mecanismos necesarios para la traducción entre la representación local del dato y la representación intermedia. Tales rutinas de traducción pueden ser producidas automáticamente, y en forma adicional a su funcionalidad básica, por el compilador IDL de DCE RPC: la aplicación ***rpcgen***.

El cuadro 3.2 muestra la especificación hecha en DCE IDL de la aplicación de adición entre dos números. La llamada al stub cliente desde la aplicación cliente puede ser : `c = binop_add(h, a, b, &);`.

El argumento `h` es de tipo `handle_t` el cual ubica información del estado y es mantenido por el cliente. La suma es retornada en un parámetro pasado por referencia al puntero `c`.

DCE RPC es una plataforma más rica en lo referente a las prestaciones de lenguaje y semántica respeto a ONC RPC. DCE RPC soporta la semántica "exactamente-una-vez" y "al-menos-una-vez" (denominada *"idempotencia" (idempotent)* en el contexto DCE) en sus primitivas de invocación. De la misma forma, tiene soporte para invocaciones en broadcast y peticiones de no-respuesta (las cuales se denominan *"puede-ser" (maybe)* en el contexto de DCE). En el ejemplo del cuadro 3.2, la llamada tiene semántica de "al-menos-una-vez", la cual se especifica con el modificador `[idempotent]` previo la definición del procedimiento `binop`.

Los atributos `uuid`, `version` y `endpoint` proveen una única identificación de la interfaz en el servidor. El uso normal de DCE RPC requiere además un servicio de nombre; sin embargo, a los efectos de desarrollo, la especificación del servicio de nombre puede omitirse. Un ***UUId*** *(Universal Unique Identifier)* [492] es un identificador que es único a través del tiempo y del espacio, con respecto al espacio de todos los UUIds. Un UUId puede ser utilizado con múltiples propósitos, ya sea para etiquetar a objetos con un tiempo de vida extremadamente corto; o bien, para identificar confiablemente a entidades persistentes en una red. Se considera ***punto de conexión***, o *punto final de conexión (endpoint)*, a cada extremo en una conexión a nivel de capa de transporte de red. Además de este significado, resulta adecuado definir al punto de conexión, o **punto final de servicio**, al punto de entrada a un servicio disponible en una red; es decir, al

```
/*
 * (c) Copyrigth 1990, 1991, 1992 OPEN SOFTWARE FUNDATION, INC
 * ALL RIGHTS RESERVED
 */
/*
 * OSF DCE Version 1.0, UPDATE 1.0.1
 */
[ uuid(f9f6be80-2ba7-11ce-89fd-08002b13b56d),
  version(0),
  endpoint(``ncadg_ip_udp:[6677]'', ``dds:[19]'') ]

 interface binop
 {
     [idempotent] void binop_add
     (
         [in] handle_t h;
         [in] long a;
         [in] long b;
         [out] long *c;
     );
 }
```

Cuadro 3.2: Suma de dos números. Ejemplo expresado en DCE IDL.

punto de acceso a un programa o recurso [230].

En cuanto al soporte de runtime, DCE RPC por defecto provee ejecución concurrente en el servidor de diferentes llamadas. Cada vez que una invocación llega al servidor, el servidor crea un nuevo hilo de ejecución para que maneje la llamada. Una petición de llamada será puesta en cola de espera, sólo si no existen recursos necesarios para que el sistema del servidor produzca un nuevo hilo. Esta propiedad diferencia a DCE RPC de ONC RPC, ya que el uso de hilos de ejecución es integral al ambiente, en tanto que en ONC RPC su utilización estaba supeditada al diseño de la aplicación servidor y a la posibilidad que tuviera la plataforma del servidor para gestionar hilos.

DCE RPC cuenta con un servicio de plataforma que hace las veces de despachante (véase Fig. 3.6). El ***servicio de mapeo de puntos de conexión*** *(endpoint mapper service)* mantiene una base de datos local en dónde asocia puntos de conexión con direcciones que localizan aplicaciones servidoras en una máquina con servidores, interfaces y otros objetos. Cada interacción basada en RPCs utiliza este servicio a través del runtime de DCE RPC para resolver ligaduras entre clientes y servidores [230].

El mecanismo de comunicación utilizado para invocaciones remotas en DCE RPC depende de la semántica que fue establecida en la descripción IDL del procedimiento. Para llamadas "al-menos-una-vez" (o "idempotencia"), se utiliza un mecanismo simple de *petición/respuesta*. Para el caso de las semánticas "exactamente-una-vez" (la opción por defecto en DCE RPC), el protocolo es el siguiente:

1. El cliente manda un paquete de petición al servidor.
2. El servidor manda un paquete de respuesta al cliente en el cual acusa el recibo del paquete de petición.
3. El cliente contesta con otro acuse de recibo, indicando la recepción del paquete de respuesta.

Este mecanismo de comunicación, en conjunto con el uso de UUIDs con los paquetes de petición y respuesta, aseguran el soporte de la semántica "exactamente-una-vez" en DCE RPC; aún utilizando protocolos no orientados a la conexión como UDP.

Al igual que ONC RPC, DCE RPC resuelve le problema de representación en diferentes plataformas traduciendo las representaciones locales a una representación intermedia común, denominada ***NDR*** *(Network Data Representation)* [491].

Para establecer la conexión entre cliente y servidor, DCE RPC soporta varios mecanismos de autenticación [34]. Los principales son:

- **Kerberos Versión 5**[3]: la opción por defecto, también denominado *DCE Compatido-Secreto (DCE shared-secret).*;
- **No Autenticación**: este modo debe ser explicitado en la petición de cliente.

El DCE provee, además de DCE RPC, un número adicional de servicios que son útiles a la hora de desarrollar aplicaciones distribuidas. Estos servicios incluyen:

- ***Servicio de Directorio de Celda*** *(Cell Directory Service)*: Es una forma sofisticada de un servidor de nombres y directorio, el cual permite que varios dominios de RPC puedan coexistir sobre la misma red, sin interferencias.
- ***Servicio de Reloj*** *(Time Services)*: provee mecanismos para sincronización entre nodos.
- ***Servicio de Hilos*** *(Threads Services)*: coordina la atención a cada petición cliente generando un hilo de ejecución. El servicio de hilos tiene también soporte para múltiples procesadores.
- ***Servicio de Archivos Distribuidos*** *(Distributed File Service)*: permite compartir archivos a través del ambiente DCE.
- ***Servicio de Seguridad*** *(Security Service)*: provee mecanismos para autenticación y seguridad en el ambiente DCE.

3.2.8. Importancia de las RPCs en la evolución del middleware

Las RPCs fueron concebidas para hacer más fácil la programación de aplicaciones distribuidas ocultando los detalles inherentes al canal de comunicación. En sí, éste fue un fuerte debate en los orígenes de las RPCs: si las RPCs debían ser transparentes al programador o no. Por un lado, se entienden que debían permanecer transparentes porque se ocultaba el detalle de la comunicación a bajo nivel de las partes de una aplicación distribuida, bajo el concepto simple del procedimiento. Por otro lado, se consideraban que no debían ser transparente, porque si un programa tenia una llamada a procedimiento remoto cambiaba la naturaleza del programa. El hecho de utilizar constructores especiales para las RPCs, forzaba a los programadores a entender las implicaciones funcionales (invocación a funciones remotas) y no funcionales (disponibilidad, rendimiento, etc) de la distribución, por ende a reducir las posibilidades de error.

En su forma más básica, los sistemas RPCs son el punto de partida para el desarrollo de las subsiguientes plataformas de middleware. La mayoría de las formas de middleware que evolucionaron después son construidas a partir de RPCs o de sus extensiones. En la actualidad, las RPCs son la parte fundamental de muchos sistemas informáticos distribuidos. Por ejemplo, las *invocaciones a métodos remotos* o *RMI (remote method invocation)* [465] soportan el paradigma

[3] ***Kerberos*** es un protocolo de autenticación de redes de ordenador que permite a dos computadores en una red insegura demostrar su identidad mutuamente de manera segura. Sus diseñadores se concentraron primeramente en un modelo de cliente-servidor para brindar autenticación mutua: tanto cliente como servidor verifican la identidad uno del otro. Kerberos se basa en criptografía de clave simétrica y requiere un tercero de confianza. [452, 255, 311]

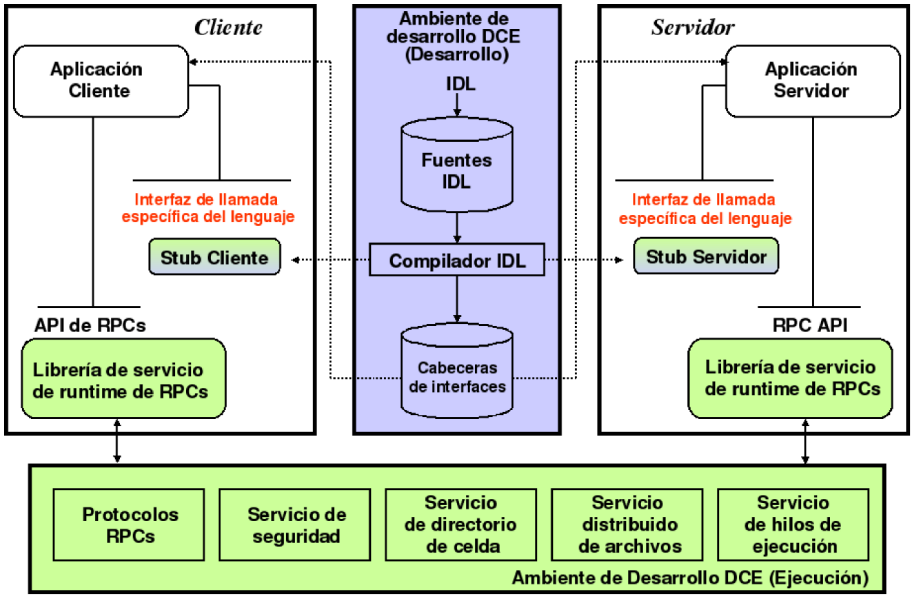

Figura 3.8: La Arquitectura de DCE. Coloreadas se ven las partes de esta arquitectura que conforman los mecanismos necesario para el desarrollo y ejecución de RPCs.

de orientación a objetos aplicado a sistemas distribuidos, pero su arquitectura subyacente extiende los mecanismos de las RPCs a la invocación de métodos de objetos remotos. Otro ejemplo son los *procedimientos almacenados (stores procedures)* de algunos sistemas de administración de bases de datos. Un procedimiento almacenado encierra un conjunto de tareas para aplicar a una o varias bases de datos. Tales procedimientos puede ser invocado en forma remota por clientes que tiene una conexión activa al motor de bases de datos que contiene a la o las bases de datos.

Como herramienta de desarrollo, las RPCs pueden ser usadas en la construcción de una nueva plataforma de middleware como primitivas de bajo nivel para implementar interfaces con funcionalidades más completas. De esta manera, algunos programadores optan usarlas directamente, por su simpleza, para programar aplicaciones distribuidas.

Por lo expresado, es indudable que hay que reconocer el lugar que tiene las RPCs en la historia de los sistemas distribuidos y en la plataformas de middleware. Cualquiera sea la forma en que las RPCs sean usadas, el mecanismo subyacente y las implementación de las RPCs han llegado a ser una parte intrínseca de cualquier plataforma de middleware, de la integración de aplicaciones y de los Servicios Web.

3.3. Monitores de Procesamiento de Transacciones

Los ***Monitores de Procesamiento de Transacciones*** *(Transaction processing monitors)* ***Monitores TP***, o simplemente ***Monitores de Transacciones***, son una de las formas más antiguas de middleware. Debido a que están presentes en la industria hace tiempo, son las plataformas de middleware más conocidas y estudiadas en detalle [60, 192, 533]. Además, los productos basados en monitores de transacciones son los más eficientes debido a que su infraestructura y mecanismos internos son optimizados desde hace años. En la actualidad, los Monitores TP son la piedra angular de muchos sistemas n-capas y su arquitectura y funcionalidad son de constante referencia para las nuevas formas de middleware.

3.3.1. Contexto Histórico

En sí, los monitores TP son previos a las arquitecturas cliente/servidor y a las 3-capas. Uno de los monitores de transacciones más antiguos es el ***CICS*** *(Customer Information and Control*

System) [228] de IBM, desarrollado en los 60's y usado hasta la actualidad. Inicialmente, este monitor TP fue diseñado para dotar a las mainframes de soporte eficiente a la multiplexación de recursos entre sus aplicaciones clientes que accedían en forma concurrente. Como parte de esta tarea, era necesario que CICS contara con mecanismos para manejar múltiples hilos de ejecución (multitheading) y consistencia de datos; es por eso que extendió su funcionalidad abarcando el concepto de transacción. CICS fue el primer producto que ofreció un entorno para realizar transacciones protegidas en un ambiente distribuido [60].

Es importante entender que la génesis de los Monitores de Procesamiento de Transacciones fue influenciada por las capacidades de los sistemas operativos de la época. En muchas formas, los Monitores TP fueron considerados como sistemas operativos alternativos que ofrecen capacidades que completan a las soportadas por los sistemas operativos tradicionales. Por ejemplo, CICS era conocido por crear y despachar hilos de ejecución más rápido que el SO que lo contenía. CICS, como otros que lo sucedieron, era no sólo un ambiente de ejecución sino que también era una completa herramienta de desarrollo. En coherencia con la arquitectura monocapa prevaleciente en la época, CICS era enteramente monolítico y corría completamente como un único proceso en el SO.

Con la migración a arquitecturas cliente/servidor y a 3-capas, los Monitores de Transacciones se hicieron más modulares pero no necesariamente más simples. Esto permitió que ofrezcan mayores funcionalidades. Así como CICS fue inicialmente un sistema monocapa diseñado para transacciones en línea; ***Tuxedo*** *(Transactions for Unix, Extended for Distributed Operations)* [37], de los comienzos de los 80s, fue un sistema de 2-capas basado en colas.

Eventualmente, todos los productos comerciales basados en monitores TP se convirtieron en sistemas 3-capas que soportaban funcionalidades para transacciones en lotes, transacciones en-línea y hasta transacciones interactivas.

Obviamente, tales mejoras en las funcionalidades requirieron un mejoramiento en las interfaces de programación. Como resultado, los monitores de transacciones comenzaron a proveer sus propios lenguajes o versiones de lenguajes de programación. Por ejemplo, en los principios de los 90s, Encima [147, 502] introduce el *Transactional-C* [229], un dialecto de C en el cual la transacción era el constructor de primera clase; lo que hacia considerablemente más fácil programar en ambientes de Monitores de Transacciones.

Como excepción a este incremento de funcionalidad y complejidad en la evolución de Monitores TP, están los *Monitores TP-lite*, o *Motores de Proceso de Transacciones Livianos*. La idea era proveer sólo el manejo de transacciones de RPCs de los Monitores TP, como capa adicional embebida en los sistemas de administración de base de datos DBMS. Para implementar *procedimientos almacenados*. Un procedimiento almacenado permite escribir lógica de aplicación dentro del motor de bases de datos, en vez de hacerlo en una capa intermedia independiente como en los Monitores de Transacciones convencionales. El resultado de esta configuración, es un sistema 2-capas más simple y liviano que los Monitores de Transacciones, pero lo suficientemente poderoso para aquellas aplicaciones que sólo cuentan con una base de datos como nivel de gestión de recursos. Por esta razón, hay controversia en considerar a los Monitores TP-lites verdaderas herramientas de integración y verdaderos Monitores de Transacciones; sin embargo son ampliamente usados en una vasta cantidad de soluciones.

Por muchas décadas, los Monitores de Procesamiento de Transacciones fueron la forma dominante de middleware. De hecho, una de las formas más exitosa en rendimiento y disponibilidad. En la actualidad, los Monitores TP pueden ser encontrados en la mayoría de los servidores de aplicaciones [228, 225], MTS de Microsoft [301] y Tuxedo de BEA [37]. De la misma forma, los Monitores de TP están en el corazón de las suites modernas de integración de aplicaciones empresariales. En síntesis, los Monitores de Procesamiento de Transacciones, han jugado un rol gravitante en el desarrollo de las plataformas de middleware actuales.

3.3.2. RPCs Transaccionales y Monitores TP.

El principal logro de los Monitores de Procesamiento de Transacciones es soportar la ejecución de *transacciones distribuidas*. Con este propósito, los Monitores TP implementan una abstracción llamada ***RPC Transaccional*** *(TRPC)*.

Los sistemas de RPCs convencionales fueron originariamente diseñados para soportar invocaciones simples y únicas de un cliente a un servidor. Sin embargo, este esquema de trabajo falla cuando las llamadas se encadenan. Es decir, una aplicación cliente A hace una llamada remota a una aplicación servidor B, el cual hace una segunda llamada remota a otra aplicación servidor C. Si alguna de las dos invocaciones falla, el cliente A no sabrá que fue hecho y que no; en dónde se produjo el error, si en B o C. En el peor de los casos, si los cambios producidos en B deben ser consistentes con los que se espera producir en C y si estos no se llevaron a cabo, existirán inconsistencias en el estado de los servidores; inconsistencias que el cliente A no podrá manejar.

Un escenario similar se produce si un cliente intenta interactuar con dos o más servidores, como parte de una única tarea, realizando dos llamadas remotas en secuencia a sendos servidores. No existe forma de mantener la consistencia entre los servidores, si alguna de las invocaciones falla, al menos que se programe la aplicación cliente para manejar tal eventualidad.

La semántica de las RPCs transaccionales indica que si un grupo de invocaciones dentro de una *transacción* son llevadas a cabo exitosamente, se puede garantizar que todas éstas fueron ejecutadas. Si algunas de las invocaciones fueron abortadas o fallidas, ninguna de éstas se ejecuta; o mejor dicho, no quedan efectos colaterales de las invocaciones parciales.

Las llamadas a procedimientos que engloban una transacción se encierran en delimitadores específicos de comienzo de transacción *BOT (beginning of transaction)* y de fin de transacción *EOT (end of transaction)*. Esto garantiza que la infraestructura de las RPCs mantenga la *atomicidad*. La transacción en sí, es llevada a cabo por el ***gestor de transacciones*** *(transaction management)* el cual coordina interacciones entre clientes y servidores. La funcionalidad de un gestor de transacciones puede ser explicada como una extensión del mecanismo de RPCs.

Supongamos un ejemplo en dónde la transacción reside en la aplicación cliente, la cual contiene dos llamadas remotas a dos aplicaciones servidor diferentes A y B (encerradas entre BOT y EOT) (véase Fig. 3.9). Cuando la instrucción BOT es encontrada, el stub cliente contacta al gestor de transacciones para crear el ***contexto de la transacción*** y darle un ***identificador***. El contexto de transacción será utilizado en forma compartida a lo largo de la ejecución de las llamadas. Desde este momento, todas las invocaciones llevan el contexto consigo. Cuando el stub cliente realiza la llamada al primer servidor A, el stub servidor A extrae el contexto de transacción, notifica al gestor de transacciones que está participando de la transacción y prosigue con la llamada en forma normal. Lo mismo ocurre con la invocación al segundo servidor B.

Cuando se alcanza la instrucción EOT, el stub cliente notifica al gestor de transacciones. El gestor de transacciones comienza un ***protocolo commit de dos fases*** o ***2PC*** *(two-phases commit)* entre los dos servidores involucrados para determinar el resultado de las invocaciones respectivas. Una vez que el protocolo termina, el gestor de transacciones le informa al stub cliente, el cual retorna un código en respuesta al EOT del cliente indicando si la transacción pudo realizarse correctamente o no.

El protocolo 2PC [191, 267] es usado en las TRPCs desde sus orígenes en el producto comercial CICS de IBM. Luego fue estandarizado por el Open Group dentro de la especificación X/Open [489]. En la actualidad, 2PC es el estándar para garantizar atomicidad en los sistemas distribuidos [533, 192]. En el protocolo 2PC, el gestor de transacciones realiza el commit de la transacción en dos fases [318, 432]:

- En la primera fase, el *gestor de transacciones o* ***coordinador*** (como se lo denomina en el contexto de 2PC [191]), envía un mensaje *"prepare-to-commit"* a cada uno de los servidores involucrados, preguntando si están listos para ejecutar el commit de la transacción. A esto, cada *servidor o* ***cohorte*** contesta *"ready to commit"* si el procedimiento invocado por la

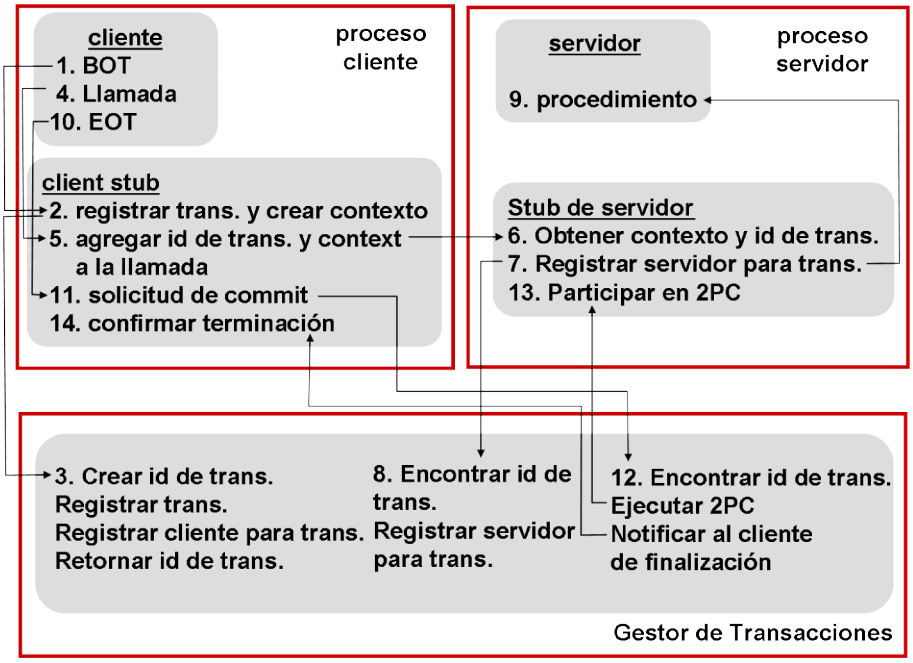

Figura 3.9: Esquema de Invocación de RPCs transaccionales.

llamada fue completado con éxito. Se genera un *registro de evento* con información para deshacer las operaciones involucradas en el procedimiento invocado, y otro registro para establecerlas definitivamente. Por otro lado, el servidor responderá *"abort"* en el caso que no haya podido completar el procedimiento invocado.

- En la segunda fase, el gestor de transacciones revisa cada una de las respuestas de los servidores. Si estas respuestas son todas *"ready-to-commit"*, el instruye a cada uno de los servidores para que realicen el commit definitivo de la transacción. Si al menos uno de los servidores respondió con *"abort"*, el gestor pide a todos los servidores que deshagan los efectos de cada procedimiento ejecutado.

La tolerancia a fallas del 2PC es logrado por *registro de eventos (logging)*; es decir, por registrar en almacenamientos persistentes el estado del protocolo en cada llamada remota transaccional. De esta forma, consultando a la bitácora de eventos, es posible consultar y reconstruir la situación previa a una falla y recuperar el sistema. Lo que se registra en la bitácora de transacciones depende de la configuración del gestor de transacciones. Sin embargo, es importante destacar que el registro de eventos de transacciones tiene directa implicación en el rendimiento de todo el sistema.

Como se ve, el protocolo 2PC realiza una serie de suposiciones para su concreción [432]. Una de esas suposiciones es la existencia de una almacenamiento permanente y estable en cada participante de la transacciones, obligado por el registro de eventos.

Otra suposición, asume que ningún participante o nodo estará caído por siempre, y eventualmente cada uno de las cohortes se comunicarán con el coordinador. La última suposición no genera grandes complicaciones, porque el enrutado del tráfico de red generalmente puede ser reconfigurado. Sin embargo, la primera suposición es mucho más fuerte y conflictiva, ya que exige la existencia de todos los puntos de una transacción; no dando lugar para que un nodo desaparezca. Esto es el origen de la principal desventaja del protocolo 2PC: la cualidad de protocolo es bloqueante. Un nodo permanecerá bloqueado hasta que reciba los mensajes de confirmación. El coordinador de la transacción puede permanecer bloqueado indefinidamente si una de sus aplicaciones cohortes no contesta el mensaje "prepare-to-commit". Y todas las aplicaciones cohortes, que completaron su parte de la transacción pueden permanecer bloqueadas indefinidamente en

espera del "ready-to-commit" del coordinador, si éste falla.

3.3.3. Funcionalidades de un Monitor de Procesamiento de Transacciones

En sí, los Monitores TP son extremadamente complejos. En una forma muy simple, se puede definir su funcionalidad como toda la necesaria para desarrollar, ejecutar, gestionar y mantener sistemas informáticos distribuidos transaccionales. Esta funcionalidad usualmente incluye:

- **Soporte para RPCs**. Es decir, contar con IDLs, servidor de directorios y nombres, seguridad y autenticación, compiladores de stubs, etc.
- **Abstracciones de programación para tratar con RPCs transaccionales**. Esto incluye RPCs, BOT, EOT, y mecanismos para manejar reintentos (callbacks) y soporte para estructuras de flujo de ejecución (como ser *on commit*, *on abort*, etc).
- Un **gestor de transacciones** para implementar las TRPCs. Este gestor debe tener completo soporte para el protocolos 2PC. Además de funcionalidades para recuperación, bloqueo y bitácora de transacciones.
- Un **monitor de sistema**, encargado de sincronizar los hilos de ejecución, asignar prioridades, realizar balanceo de carga, replicación, inicio y detención de componentes, y otras. El monitor de sistema provee la flexibilidad y el rendimiento al monitor TP.
- Un **ambiente de ejecución** que actúa como soporte computacional para todas las aplicaciones que usan el monitor TP. Este ambiente de run-time provee los recursos y servicios que pueden ser necesitados, como por ejemplo seguridad, sistemas de archivos transaccionales, servicios transaccionales, etc.
- **Componentes adicionales**, especialmente adecuados para escenarios particulares. La variedad de estos componentes puede abarcar desde protocolos propietarios para interactuar con mainframes hasta sistemas basados en colas persistentes para interacciones asincrónicas.
- **Otras herramientas** para instalación, gestión, control y auditoría del rendimiento de cada uno de los componentes previos.

Los monitores TP proveen interfaces de programación que pueden usarse en diferentes niveles de complejidad. Están aquellas de forma más simple, que se limitan a transformar las RPCs en TRPCs; y también las más complejas que permiten configurar los niveles más bajos de la interacción.

3.3.4. Arquitectura de un Monitor de Procesamiento de Transacciones

En una forma simplificada, la arquitectura de un monitor TP esta dada por:

- **Interfaz**: es la vinculación directa con el cliente. La interfaz provee las API necesarias para la programación de clientes, como así también acceso directo por terminal y mecanismos de autenticación.
- **Flujo de programa**: el componente de flujo de programa almacena, carga y ejecuta procedimientos, posiblemente escritos en un lenguaje provisto por el monitor TP. Los programas almacenados envuelven invocaciones de operaciones a los recursos lógicos, identificados por su nombre.

- **Router o enrutador**: mapea cada operación con una invocación. La invocación puede involucrar a un recurso en la capa de gestión de recurso subyacente (como por ejemplo una base de datos) o un servicio local provisto por el mismo monitor TP. El router incluye una base de datos propia en la que almacena la definición de relaciones entre los nombres de recursos lógicos y los dispositivos físicos. En el caso de cambiar la configuración de sistema, el administrador del monitor TP sólo deberá actualizar este mapeo; el cliente no necesita ser modificado ya que se accede a los recursos por su nombre lógico.

- **Gestor de comunicaciones**: es el que permite la comunicación con los recursos distribuidos del sistema. Puede ser un sistema basado en mensajes, seguramente dotado de semántica de transacciones para poder revertir fallas o garantizar peticiones.

- **Envolturas** (Wrappers): sirven para ocultar la heterogeneidad de los diferentes recursos distribuidos. Simplifican el desarrollo de los módulos de comunicación, ya que permiten implementarlo en forma independiente de las características de un determinado recurso.

- **Gestor de Transacciones**: es el que administra las transacciones distribuidas. Mediante la ejecución del protocolo 2PC garantiza la propiedad ACID (atomicidad, consistencia, unicidad y durabilidad) de los procedimientos ejecutados a través del monitor de TP.

- **Otros Servicios**: una amplia gama de servicios proveen al monitor TP de rendimiento eficiente, alta disponibilidad, tolerancia a fallas, robustez, replicación, etc.

Los Monitores de Transacciones están hechos para soportar cientos o miles de clientes accediendo concurrentemente. Esta carga de trabajo no puede ser alcanzada por arquitecturas monocapas o 2-capas. Tal capacidad de carga es posible porque los Monitores TP utilizan hilos de ejecución para atender a cada uno de sus clientes, en lugar de disparar procesos (como lo hacen sistemas monocapas o 2-capas). Los procesos son unidades pesadas autocontenidas en memoria; cada una contiene en forma oculta el código a ejecutar y datos de sesión. Requieren ser administrados por el SO. A diferencia de los procesos, los hilos *(threads)* son entidades livianas, que comparten en memoria el código que los originó con otros hilos de la misma clase, como también todos los datos visibles a la hora de su creación. Una vez creados, cada hilo tiene su propio identificador, como también su propio registro y pila de variables, lo que le permite ejecutarse independientemente.

Además de los mecanismos de gestión de hilos, los monitores TP proveen componentes para el balance de carga para mejorar el rendimiento si se dispone de varias máquinas. Una parte importante de la arquitectura de los monitores TP está destinada a la implementación eficiente de estos mecanismos. En particular, la performance de los monitores de procesamiento de transacciones son su característica distintiva.

3.4. Brokers de Objetos

Un ***Broker de Objetos***, o ***Broker de Peticiones de Objetos*** *(object request broker)* es una tecnología de middleware que gestiona la comunicación e intercambio de datos entre objetos. Un Broker de Objetos brinda a los desarrolladores de sistemas orientados a objetos la posibilidad de construir sistemas distribuidos utilizando objetos ya desarrollados (implementados en diferentes plataformas y lenguajes) residentes en máquinas distintas que se comunicarán entre sí mediante el Broker de Peticiones de Objetos [521].

Un Broker de Objetos debe soportar variadas y numerosas funcionalidades para poder operar consistentemente y eficientemente; dichas funcionalidades están ocultas a las aplicaciones que utilizan el Broker de Objetos para interactuar. Como toda plataforma de middleware, los Brokers de Objetos tratan de ocultar la distribución. Es responsabilidad del Broker de Objetos proveer la ilusión de localía; es decir, hacer que parezca que un objeto es local a la aplicación cliente, aunque

dicho objeto resida en otro proceso y/o máquina diferente [405]. De esta manera, el Broker de Objeto provee un marco de trabajo para comunicación entre objetos inter-sistemas; siendo este un pilar para la interopertividad y reutilización de sistemas de objetos.

El siguiente paso técnico, hacia la interoperatividad de sistemas, es la comunicación entre objetos a través de plataformas heterogéneas. Un Broker de Objetos permite que los objetos oculten el detalle de implementación de las aplicaciones clientes. Estos "detalles" pueden incluir el lenguaje de programación, el sistema operativos, el hardware de la máquina y la ubicación del objeto. Cada uno de ellos pueden ser "transparentes" para la interoperatividad y el reuso. Diferentes sistemas de Brokers de Objetos pueden soportar diferentes niveles de transparencia, extendiendo los beneficios de la orientación de objetos, traspasando los límites de las plataformas y canales de comunicación.

Los detalles de implementación de la infraestructura del broker de objetos no son importantes para los desarrolladores de aplicaciones distribuidas; en sí, ellos solamente se concentran en los detalles de las interfaces de los objetos distribuidos. Al igual que las plataformas de middleware basadas en RPCs, esta forma de ocultamiento de información mejora el mantenimiento del sistema debido a que los detalles de comunicación entre objetos distribuidos son ocultos de los desarrolladores y aislados en el broker de objetos [103].

Los Brokers de Objetos extienden el paradigma de las RPCs al universo de la orientación a objetos [10]. Además proveen una serie de servicios que simplifican el desarrollo de aplicaciones orientadas a objetos distribuidas. En pocas palabras, los Brokers de Objetos son middlewares que soportan la interoperatividad entre objetos distribuidos.

3.4.1. Contexto Histórico

Los Brokers de Objetos aparecieron en los principios de los 90s como una natural evolución de las RPCs hacia el ámbito de la orientación a objetos; paradigma de programación que fue desde entonces ganando aceptación. El propósito de los Brokers de Objetos es el mismo que las RPCs: ocultar la complejidad de llamadas remotas haciendo que se vean como llamadas locales desde la perspectiva del programador. En los Brokers de Objetos, en vez de tratar con llamadas a procedimientos, se trata con invocaciones a métodos de objetos remotos. Si una aplicación cliente invoca un método de un objeto remoto, la función que se ejecutará en respuesta dependerá de la clase de ese objeto remoto en el servidor. En un mismo programa cliente, puede tener diferentes invocaciones a diferentes métodos de un mismo objeto remoto, las cuales deben ser coherentes de acuerdo a su clase. Es por eso que un middleware debe ligar los clientes con objetos específicos corriendo en el servidor, y manejar las iteracciones en el tiempo que perdure la ejecución del cliente. Esta es la funcionalidad básica de un broker de objetos.

Con el tiempo, los brokers de objetos han agregado funcionalidades que van mas allá de la interoperatividad, como ser transparencia de locación, sofisticadas técnicas de ligaduras dinámicas, gestor de ciclo de vida de los objetos y persistencia.

Existen varias formas de implementar los conceptos básicos de un Broker de Objetos. Por ejemplo, las funcionalidades pueden estar compiladas en la aplicación cliente, pueden ser un proceso o servicio independiente, o pueden ser parte del kernel del sistema operativo. Estas decisiones de diseño pueden ser fijadas en un único producto, o puede haber una variedad de opciones definidas por el implementador del Broker de Objeto.

Probablemente el mejor ejemplo de broker de objetos es la especificación ***CORBA*** *(Common Object Request Broker Architecture)* [348, 388, 374]. Creada por OMG (Object management Group) en los inicios de los 90s. CORBA fue concebido como una arquitectura de referencia para sistemas basados en componentes. Desde el punto de vista de los middlewares, CORBA ofrece una especificación completa de un broker de objetos. A diferencia del DCE RPC, no provee ninguna implementación específica como estándar [349]. La especificación trata de ser abstracta al lenguaje de programación empleado para el desarrollo, como así también al sistema operativo en dónde correrá la aplicación.

Si bien CORBA es la especificación más popular y la que cuenta con más productos que la implementan, no debe tratarse a los Brokers de Objetos y CORBA como sinónimos. De hecho existe brokers de objetos que no son basados en CORBA, como por ejemplo el *Modelo de Objetos de Componentes Distribuidos DCOM (Distributed Component Object Model)* [296, 486] y sus descendientes como por ejemplo COM+ [387], los cuales son específicos de los sistemas operativos de Microsoft.

Más recientemente, el interés en CORBA ha decrecido conforme nuevas tecnologías surgen. Como por ejemplo las plataformas de servidores de aplicaciones de .Net de Microsoft y J2EE de Java (véase Sec. 4.4). La OMG ha contestado a este desinterés permitiendo interoperatividad entre Java y CORBA [366], definiendo un modelo de componentes CORBA más rico basado en J2EE [360].

Sin embargo, CORBA no deja de ser un estándar híbrido que cubre no sólo la orientación de objetos, sino también muchas de los servicios ofrecidos por otras formas de middleware. Desde ese punto de vista, es natural que CORBA sea eventualmente abarcado por otros estándares.

3.4.2. Arquitectura de Sistema CORBA

Todo sistema compatible con CORBA esta compuesto de tres partes principales (véase Fig. 3.10):

1. El ***ORB*** *(Object Request Broker)*: es el "negociador" de la plataforma, el cual permite a los objetos hacer y recibir peticiones y respuestas en una forma transparente en un ambiente distribuido. Es la base para construir aplicaciones de objetos distribuidos y para la interoperatividad de aplicaciones en ambientes homogéneos y heterogéneos.

2. ***Servicios de Objetos***: son un conjunto de servicios (interfaces y objetos) que soportan las funciones básicas para usar e implementar objetos distribuidos. Los servicios son necesarios para construir cualquier aplicación distribuida y son siempre independientes del dominio de aplicación. Los servicios son accesibles por APIs de programación. Por ejemplo, el Servicio de Ciclo de Vida (life Cycle Services) [354] define las convenciones para crear, eliminar, copiar y mover objetos; no especifica cómo el objeto es implementado en una aplicación. Las especificaciones de los Servicios de Objetos están contenidas en la especificación *CORBAservices (Common Object Services Specification)* [349].

3. ***Facilidades CORBA***: son un conjunto de servicios o prestaciones que varias aplicaciones pueden compartir, pero las cuales no son fundamentales como los Servicios de Objetos. Generalmente proveen servicios de alto nivel necesarios para las aplicaciones antes que para objetos individuales. Ejemplos de estas prestaciones son: gestión de documentación, internacionalización, soporte para agentes móviles, gestión de sistema y correo electrónico. Las facilidades están estrechamente ligadas con el dominio de aplicación. La especificación que las describe es CORBAfacilities: Common Facilities Architecture [347].

4. ***Aplicaciones de Objetos*** (Objetos definidos por el Usuario): son productos adquiridos o desarrollados en forma propietaria. Se corresponden con la noción tradicional de aplicaciones, por consiguiente no están estandarizados por la OMG. Las aplicaciones de objetos son la capa superior en el Modelo de Referencia CORBA.

Es así que el Object Request Brokers (ORB) es el corazón del Modelo de Referencia. Es como una central telefónica, la cual provee los mecanismos básicos para realizar y recibir llamadas remotas [453]. Combinado con los Servicios CORBA, se asegura niveles significativos de comunicación entre aplicaciones compatibles con CORBA

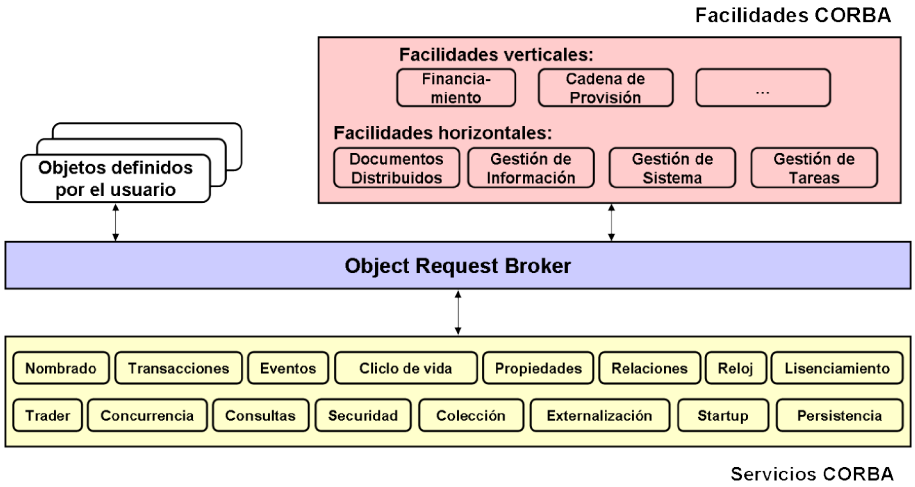

Figura 3.10: Arquitectura básica de CORBA.

3.4.3. Fundamentos y esquema de trabajo de CORBA

Fundamentos

El esquema de trabajo para el desarrollo en CORBA [363] es muy similar al planteado en RPCs (véase Fig. 3.11):

- Primeramente, se definen los **tipos de objetos** o *clases* por medio de interfaces descritas en lenguaje de descripción de interfaces, el ***OMG IDL*** [364]. Este lenguaje soporta los principios de orientación a objetos como la herencia y el polimorfismo. Una *interface* es un conjunto de nombres de operaciones y los parámetros que se requieren para invocar a tales operaciones. Hay que tener en cuenta que, si bien IDL provee un marco de trabajo para describir objetos que serán manipulados por el ORB, no es necesario que el código fuente IDL esté disponible para que el ORB funcione. Mientras que la información equivalente esté disponible en la forma de rutinas de stub o interfaces run-time del repositorio, un ORB debe ser capaz de funcionar adecuadamente.

- Luego, se compila la definición IDL usando un ***compilador IDL***. Éste produce el ***stub cliente***, también llamado *proxy*, *servidor proxy* o *objeto proxy*; y el *stub servidor* o ***skeleton de implementación***. La signatura del método (servicio provisto) es almacenada en el *Repositorio de Interfaces*. El ***Repositorio de Interfaces*** es un servicio de run-time que brinda la descripción IDL disponible en tiempo de ejecución. La información contenida en el Repositorio de Interfaces puede ser utilizada por el ORB para completar invocaciones, cuya ligadura puede ser estática o dinámica. Además de este rol central, el repositorio es un lugar común para almacenar información asociada con las interfaces de los objetos. Por ejemplo, librerías de stubs o skeletons.

- Por último, el ***programa cliente*** es compilado y vinculado con su stub cliente; de la misma forma que la ***implementación del objeto***, el objeto proveedor de servicio, es compilado y vinculado con su skeleton de implementación.

Ligadura Estática

En su modo básico, para desarrollar clientes, es necesario conocer la interface IDL del objeto proveedor de servicio con el que se interactuará. La programación de esta interacción debe respetar la signatura y la semántica de los métodos descriptos en la interfaz. Este esquema requiere que los clientes estén ligados estáticamente en tiempo de compilación con la interfaz del objeto

servidor. De hecho, la compilación IDL genera un stub específico para una determinada interfaz de servicio de un objeto remoto.

Servicio de Selección Dinámica e Invocación

Como alternativa a este esquema estático, CORBA permite a las aplicaciones clientes descubrir dinámicamente los objetos servidor, recuperar sus interfaces y construir invocaciones dinámicas de estos objetos. Todo esto en tiempo de ejecución y sin requerir un stub previamente generado y linkeado con el cliente. Esta capacidad está basada en dos componentes: el ***repositorio de interfaces*** *(interface repository)* [362] y la ***interfaz de invocación dinámica*** *(dynamic invocation interface)* [361] (véase Fig. 3.11). El repositorio de interfaces almacena todas las definiciones IDL conocidas por el ORB. La interfaz de invocación dinámica provee métodos como *get_ interface* y *create_ request* que pueden ser usados por los clientes para consultar el repositorio y construir dinámicamente invocaciones basadas en interfaces descubiertas.

Sin embargo, al recuperar una determinada interfaz del repositorio de invocación dinámica, el cliente ya conoce cuál interfaz solicitar; es decir, el cliente ya sabe qué servicio necesita. Existe una etapa previa en la que, precisamente, se decide cuál servicio es el adecuado a invocar. En CORBA [351] se ofrecen el ***Servicio de Nombres*** *(Naming Service)* [358] y el ***Servicio de Negociación entre Objetos*** *(Trading Object Service)* o simplemente ***Trader*** [353] para obtener las referencias a objetos remotos. La relación entre el Servicio de Nombre y el Trader es análoga a las páginas blancas y amarillas de teléfonos. El Servicio de Nombres permite recuperar una referencia de un objeto basado en el nombre del servicio que se necesita. El Trader, por otro lado, posibilita que los clientes puedan buscar un determinado servicio en base a sus propiedades. De hecho, los servicios pueden publicar sus propiedades a través del Trader. Diferentes servicios pueden tener diferentes propiedades que definen las características no-funcionales del servicios. Con el Servicio de Nombre, los clientes buscan y recuperan objetos que implementan determinada interfaz. Con el Trader, los clientes pueden obtener objetos cuyas propiedades tiene valores específicos.

A pesar que la capacidad de CORBA de realizar selección e invocación dinámicas de servicios es fascinante, es raramente usada en la práctica. Construir invocaciones dinámicas es una tarea difícil, no por el problema computacional, sino por el diseño semántico. Por empezar, para que los clientes puedan buscar y localizar servicios, deben entender el significado de las propiedades de los mismos; y ésto requiere de una ontología compartida entre cliente y proveedores de servicios. Además, si el cliente no tiene específicamente implementada la interacción con un servicio determinado, es difícil poder dotarlo de la capacidad de figurarse cuáles operaciones puede hacer el objeto proveedor de servicios descubierto dinámicamente, cuál es la semántica de cada uno de sus parámetros y en qué orden deben invocarse las operaciones para lograr la funcionalidad deseada.

3.4.4. Encapsulamiento en CORBA

Una de las características significativas en los Brokers de Objetos, y de CORBA en particular, es el *encapsulamiento*. El fundamento del encapsulamiento es ocultar los detalles internos de implementación de un objeto a sus clientes. Esto tiene su importancia en permitir a los proveedores de servicios cambiar su implementación sin que los clientes modifiquen sus códigos. El encapsulamiento está presente en sistemas RPCs y Monitores TP, ya que éstos trabajan con interfaces bien definidas descritas en IDL. Sin embargo, las interfaces son más naturales en un modelo orientado a objeto como el que sustenta OMG IDL de CORBA.

Si bien las *interfaces para programación de aplicaciones APIs* son el ingrediente principal del encapsulamiento, existen otras formas de manifestación relevantes. Un ejemplo de esto es la independencia de CORBA hacia los lenguajes de programación y sistemas operativos. A diferencia de las RPCs y los Monitores TP, los clientes y proveedores de servicios no necesitan ser implementados en el mismo lenguajes ni en la misma plataforma. En sí, los clientes y proveedores

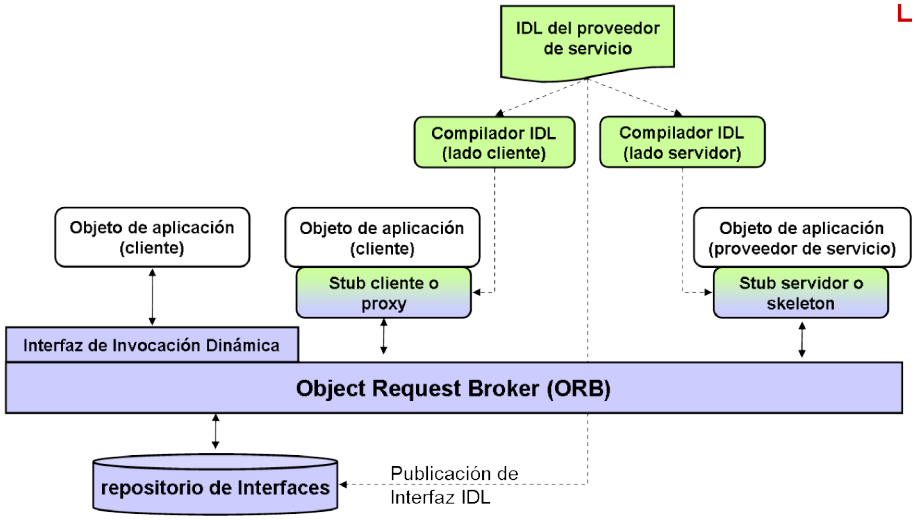

Figura 3.11: Funcionamiento básico de CORBA. (En verde se indican los componentes para soporte del desarrollo y, en azul, el soporte para ejecución)

de servicios no necesitan conocer en qué fueron implementados y sobre qué plataforma corren. Lo único necesario para el cliente es conocer la descripción IDL del proveedor de servicio. El flujo de invocaciones se realiza a través del ORB, los parámetros de invocación son traducidos a una representación independiente del lenguaje de programación y del sistema operativo por el stub cliente; en contraparte, el skeleton convierte desde la representación independiente a la específica del proveedor de servicios. Este esquema de encapsulamiento da libertad de cambiar, no solamente la lógica de programación, sino también la plataforma de desarrollo de cliente y la del proveedor de servicios, entre tanto se respete la especificación IDL.

Una gran ventaja de CORBA, sobre otros usos previos del concepto de IDL, es que el mapeo de IDL a diferentes lenguajes de programación está estandarizado [350]. Esto asegura que la implementación de un objeto, pueda ser portable a través de implementaciones de ORBs de diferentes vendedores, independientemente del lenguaje de programación usado para construir el objeto. Este esfuerzo de estandarización es mucho más efectivo en CORBA que en sistemas basados en RPCs o Monitores TP.

Otra forma de encapsulamiento provista por los Brokers de Objetos es la independencia de ubicación. Cuando los clientes necesitan invocar un servicio que soporta determinada interfaz, acceden al ORB para recuperar la referencia al objeto proveedor de servicio. Una ***referencia*** es un identificador lógico al objeto proveedor, asignado cuando el objeto se crea. Conceptualmente una referencia no tiene relación con la ubicación física del objeto; desde el punto de vista del cliente es un identificador opaco. Es función del ORB mantener la correspondencia entre la referencia y la ubicación real del objeto. La referencia se sostiene, hasta que el objeto sea destruido, o si el objeto cambió su ubicación física durante su vida.

En sí, los objetos no solamente pueden cambiar su ubicación física, sino que también pueden correr en diferentes ORB. De hecho, CORBA no sólo propone especificaciones para interoperatividad entre objetos, sino que también soporta interoperatividad entre ORBs. Para lograr esto, CORBA define que cada implementación de ORBs deba adoptar el ***Protocolo General Inter-ORBs GIOP*** *(General Inter-ORB Protocol)* [365]. Este protocolo de integración de ORBs puede ser implementado sobre otros protocolos de transporte. CORBA exige que al menos se impremente una versión de GIOP sobre TCP/IP, la cual se denomina como ***Protocolo Inter-ORB de Internet IIOP*** *(Internet Inter-ORB Protocol)* (véase Fig 3.12). Gracias a este protocolo, un ORB puede redestinar una invocación de un cliente a otro ORB en dónde reside el proveedor de servicio, siempre y cuando ambos ORB sepan de la existencia mutua. Si bien la especificación de la interoperatividad entre ORBs remotos hacen posible que CORBA sea un verdadero

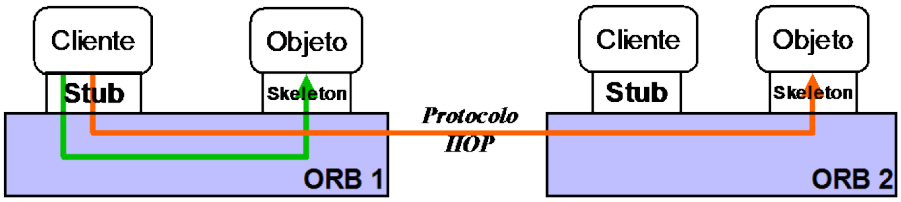

Figura 3.12: Interoperatividad entre ORBs usando IIOP. [351]

sistema universal de componentes, ha llegado demasiado tarde, ya que tal funcionalidad ha sido sobrepasada por los Servicos Web [10].

3.4.5. Monitores de Objetos = Monitores TP + Brokers de Objetos

Los ***Monitores de Objetos*** o ***Monitores de Transacciones de Objetos*** *(Object Transaction Monitors)* o ***OTMs*** [76] son un claro ejemplo de la forma en que evolucionaron las plataformas de middleware. A menudo, nuevas plataformas aparecen como una suite con muy pocas modificaciones respecto a un producto existente. Los Monitores de Objetos surgieron de esta forma. Si bien se los presenta como una fusión entre los Monitores TP y los Brokers de Objetos, en realidad fue el paso obvio en ese tiempo y sólo una forma adecuada de obtener productos de competitividad comercial.

Parte de los problemas encontrados en los Brokers de Objetos, y en particular en CORBA, es que ofrecían como novedad tecnológica solamente la orientación de objetos como forma para estandarizar las interfaces a través de diferentes sistemas y lenguajes de programación. De hecho, CORBA fue pensado para ser implementado por encima de plataformas de middleware convencionales, como DCE, Monitores TP y formas primitivas de Middleware Orientados a Mensajes (véase Sec. 3.5). Desafortunadamente los lenguajes de programación son sólo una parte pequeña en el escenario de los sistemas distribuidos.

Mucho de los servicios especificados por CORBA tardaron tiempo en implementarse. Un claro ejemplo de esto es el *Servicio Transaccional de Objetos OTS (Object Transactional Service)* [356]. El OTS describe esencialmente el funcionamiento que los Monitores TP mostraban hace ya varios años. Como los primeros productos que implementaron OTS fueron desarrollados desde cero, parecían pobres en rendimiento y funcionalidad respecto a los ya desarrollados y robustos Monitores TP. Este hecho fue gravitante y limitó la adopción de los Brokers de Objetos.

La verdadera solución de este dilema fue utilizar Motores TP con una capa adicional que permitiera la orientación a objetos, generalmente compatible con CORBA. Y ese es el surgimiento de los Monitores de Objetos.

En términos de funcionalidad, los OTMs no ofrecían grandes diferencias respecto a los Monitores TP. Este fue el primer paso hacia la asimilación de las ideas de los Brokers de Objetos por otras formas más completas de middleware. Java, C# y otros ambientes dieron culminación a este proceso de convergencia.

La compania BEA Systems (adquirida por Oracle) lanzó en 1998 el primer OTM del mercado: BEA M3 [41], basado en su Monitor TP Tuxedo [37]. En la actualidad, el producto evolucionó en la en la plataforma BEA WebLogic Enterprise [39]. Otros productos de OTMs son Component-Broker de IBM [193, 286], VisiBroker ITS [73] y su sucesor VisiTransact/VisiBroker de Borland [72] y OrbixOTM de IONA [234].

Por su parte Sun provee una implementación de un servicio de transacciones basada en Java: Sun JTS [467]. JTS soporta la especificación la API Transaccional (Java Transaction API) [472] a alto nivel, e implementa un mapeo a la especificación del Servicio Transaccional de Objetos OTS (Object Transactional Service) de OMG/CORBA [356] a bajo nivel. JTS usa las interfaces estándares de CORBA ORB/TS y el protocolo IIOP para la propagación del contexto de una transacción entre lo gestores JTS de transacciones.

3.5. Middleware Orientado a Mensajes

Un sistema de *Middleware Orientado a Mensaje (Menssage Oriented Middleware)* provee un mecanismo para integrar aplicaciones en una manera flexible y con bajo grado de acoplamiento. Este tipo de middleware brinda entrega asincrónica de datos entre aplicaciones gracias a un sistema de colas de mensajes. Los *sistemas de colas* son básicamente un almacenamiento temporal que permite el guardado y reenvío de mensajes. De esta forma, la aplicación cliente y la aplicación servidor no se comunican directamente entre sí, sino a través de la plataforma MOM, la cual funciona como intermediaria. A los principio de la década del 2000, las plataformas MOM eran el método más popular y aceptado para integrar y conectar aplicaciones empresariales y sistemas legados en ambientes heterogéneos. Las plataformas MOM les permiten a los usuarios y desarrolladores interconectar programas e intercambiar datos entre sistemas o procesos usando interfaces consistentemente definidas [371].

Las formas de middleware presentadas hasta ahora han mostrado conceptos y técnicas de interoperatividad basadas en la invocación sincrónica y bloqueante de métodos o procedimientos. Es importante explorar los MOMs, como variante conceptual a estos protocolos de interacción [10].

3.5.1. Contexto histórico

Es usual presentar a los ***Middlewares Orientados a Mensajes MOM*** *(Menssage Oriented Middleware)* como una tecnología que revolucionó la forma de construcción de los sistemas informáticos distribuidos. Pero la idea no es nueva; estaba en los modelos tempranos para implementar sistemas en batch. De hecho, existían sistemas que ofrecían una versión asincrónica de RPCs. También se contaba con monitores TP que implementaban sistemas basados en cola para lograr interacción orientada a mensajes; por ejemplo, el monitor TP de Tuxedo [37] estaba basado en sistemas de colas. Es más, la noción de *colas persistentes (persistent queue)* eran completamente entendida a comienzo de los 90s [61].

En sí, los Middleware Orientados a Mensajes modernos son descendientes de los ***sistemas de colas*** introducidos en los Monitores de Transacciones. Si bien originariamente, los Monitores TP usaban sistemas de colas para implementar sistemas basados en batch, con la expansión del uso de los Monitores de Transacciones en la intergación de sistemas de gran escala, se vio que la interacción asincrónica era mejor opción, que la ofrecida por RPCs, para esta tarea. Fue así que los sistemas de colas de los Monitores TP comenzaron a jugar un rol fundamental en el diseño de sistemas informáticos distribuidos; ésto originó que lleguen a ser plataformas independientes. Hoy en día, los mayores proyectos de integración son hechos usando Middlewares Orientados a Mensajes basados en sistemas de colas.

Como productos comerciales de plataformas MOM, se puede citar al WebSphere MQ de IBM [232], MSMQ de Microsoft [299] y WebMethod Enterprise de WebMethod (adquirido por Software AG) [444].

Por su parte, CORBA con el propósito de desacoplar la comunicación entre objetos en determinados escenarios, también especifica su propio servicio de mensajes: el *Servicio de Eventos (CORBA Event Services)* [357], cuya filosofía sirvió de base para las futuras especificaciones de sistemas basados en mensajes.

3.5.2. Interoperatividad basada en mensajes.

El término ***interoperatividad basada en mensajes*** se refiere a un paradigma de interacción dónde los clientes y proveedores de servicios pueden comunicarse mediante el intercambio de mensajes. Como se explicó en la Sec. 2.3.1, un ***mensaje*** es una estructura de datos, típicamente caracterizada por un tipo y un conjunto de ***parámetros***. Si bien los tipos usados dependerían de la plataforma en donde se implementa el sistema de mensajes, en la actualidad la mayoría de

```
Mensaje : solicitudPresupuesto {
    PresupuestoNro: Integer
    Cliente: String
    Artículo: String
    Cantidad: Integer
    FechaReqDeEntrega: TimeStamp
    DirecciónDeEntrega: String
}
```

Cuadro 3.3: Mensaje de pedido de presupuesto a un proveedor. El mensaje incluye parámetros para identificar el solicitante del presupuesto (cliente), el artículo y cantidad requerida, una fecha y dirección exigida de entrega.

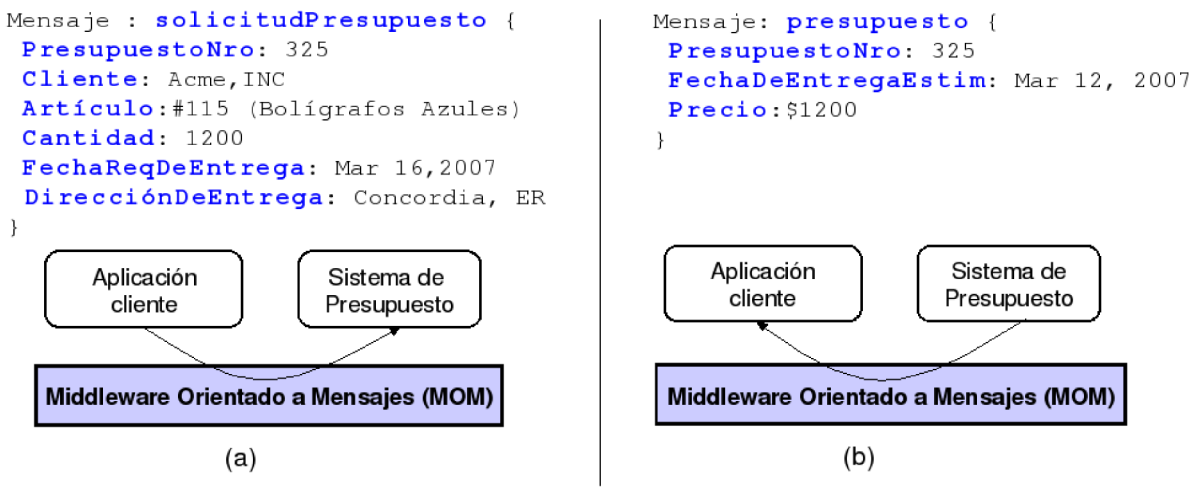

Figura 3.13: Ejemplo de interoperatibidad basada en mensajes. Una aplicación cliente envía una petición a un proveedor de servicios (a). El proveedor de servicio retorna una respuesta (b).

los casos presentan tipos basados en XML [276]. Como ejemplo, se muestra en el Cuadro 3.3 un mensaje que consiste en un pedido de presupuesto.

Con la interoperatividad entre cliente y proveedor de servicios, una vez que éstos se ponen de acuerdo en el tipo de mensajes que van a usar, se está en condiciones que interactúen mediante el intercambio de mensajes. Para el ejemplo, se puede suponer que existe una aplicación que recibe las solicitudes de presupuestos (aplicación cliente) y la transfiere a un sistema de cálculo de presupuesto (proveedor de servicio). Este sistema confecciona el presupuesto y notifica a la aplicación cliente (véase Fig. 3.13).

Si bien, en las explicaciones relacionadas con las plataformas MOM, se hace referencia a "cliente" y "proveedor de servicio", esta distinción puramente conceptual en una interacción orientada a mensajes; ya que ambas partes meramente intercambian mensajes. La dualidad cliente-servidor sólo sirve para explicar la semántica de los mensajes y de la secuencia de diálogos entre componentes. Esta es una diferencia sustantiva con respecto a las otras formas de interacción discutidas previamente, ya que en esos casos existían objetos o componentes que actuaban como clientes invocando a otros objetos o funciones que actuaban como servidores.

3.5.3. Colas de Mensajes

Uno de los principales aportes de las plataformas MOMs es la utilización de los sistemas de ***colas de mensajes***. En un modelo de colas de mensajes, los mensajes enviados por una

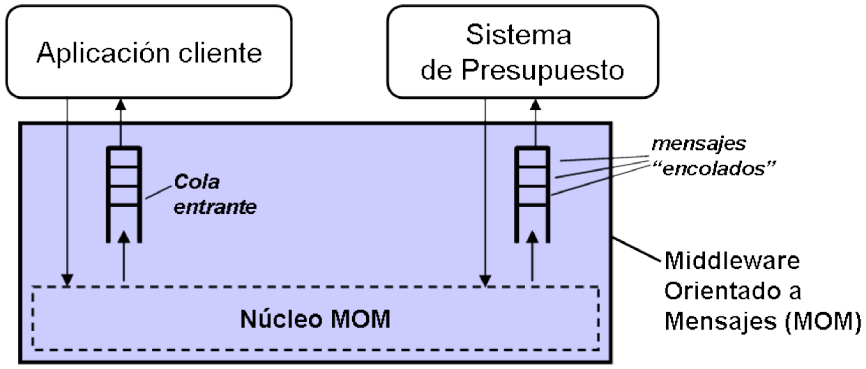

Figura 3.14: Modelo de colas de mensajes

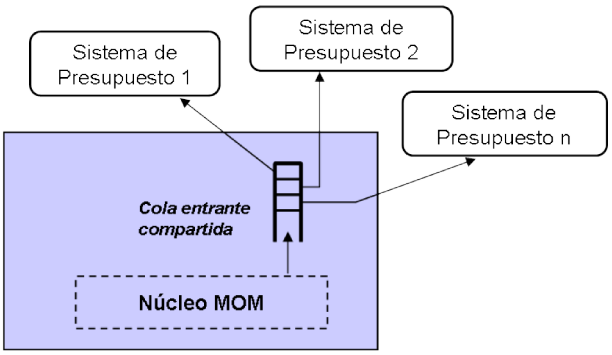

Figura 3.15: Modelo de colas de mensajes compartida

aplicación cliente son "encolados"[4] o ubicados en estructuras de datos colas, identificadas con un nombre y asociadas a una determinada aplicación destinataria (proveedora de servicio). Cuando el destinatario lo determine, invocará la función adecuada del MOM disponible por su API, para que recupere los mensajes en su cola respectiva.

Las colas de mensajes brindas varios beneficios. En primer lugar, le da el control a los destinatarios de cuándo tratar los mensajes. Los destinatarios pueden establecer políticas para gestionar la atención y procesamiento de mensajes. No es necesario que se esté continuamente verificando la cola de mensajes.

Los sistemas basados en colas son más robustos y tolerantes a fallas que los sistemas basados en RPCs o los Brokers de Objetos. Por ejemplo, el proceso destinatario puede no estar corriendo o imposibilitado de recibir mensajes. En este caso, los mensajes serán retenidos en la cola específica (mantenida por el MOM), y serán entregados una vez que el destinatario vuelva a estar en servicio. Esto evita el bloqueo de las aplicaciones en espera de respuesta. Esta característica logra una importante flexibilidad en la integración de sistemas. Como contraparte, exige que las aplicaciones sean diseñadas de manera diferente a la natural forma de interrelación entre módulos, al contrario de los promovido por las RPCs (véase Sec 3.2) [157].

Otro importante punto a destacar, es la posibilidad de asociar a una misma cola de mensajes varias aplicaciones destinatarias que proveen el mismo servicio, es decir a compartir la cola de mensajes (véase Fig. 3.15). El sistema MOM controla el acceso a la cola, asegurándose que cada mensaje sea entregado a una sola aplicación destinataria. Esto posibilita una mejor distribución del balanceo de carga aumentando el rendimiento.

Por último, los mensajes pueden ser rotulados con determinados atributos que influyen directamente la gestión de la cola. Por ejemplo pueden tener un tiempo de expiración, lo que

[4] "encolado" es una traducción literal ad-hoc para el término en inglés *queued*. Que es el participio del verbo *queue* que significa *encolar* o *poner en una cola*.

permitiría desechar mensajes caducos si no se entregan en tiempo. Otro ejemplo sería que los mensajes cuenten con una prioridad, de forma tal que los mensajes de más alta prioridad sean entregados primero.

3.5.4. Formas de interacción con el sistema de cola de mensajes

Los sistemas de colas tiene una API que puede ser invocada para enviar o recibir mensajes. El envío de mensaje es típicamente una operación no bloqueante; es decir, cuando un clienten genera un mensaje y lo remite, continua su ejecución sin esperar bloqueado la confirmación de recepción o respuesta.

Por el contrario, la recepción de un mensaje es una operación bloqueante. La idea es que el componente receptor espere los mensajes y los procese a medida que los recibe. Para esto existe un hilo de ejecución principal en el receptor que activa un nuevo hilo por cada mensaje recibido para atenderlo, mientras este hilo principal vuelve a esperar el siguiente mensaje. Sin embargo, los destinatarios pueden recibir mensajes en forma no-bloqueante proveyendo una función callback que será invocada por el MOM cada vez que un mensaje arribe.

Como ejemplo de API para interactuar con sistemas MOM se puede citar al ***Servicio de Mensaje de Java JMS*** *(Java Message Service)* [463]. Al igual que la definición dada en la Sec. 2.3.1, un mensaje en JMS es caracterizado por un *encabezado (header)*, el cual incluye metadatos como por ejemplo el tipo de mensaje, fecha de expiración y prioridad; y un *cuerpo (body)* que contiene los datos específicos de la aplicación. En JMS, como en la mayoría de los MOM, la ubicación y locación de destinatarios y remitentes es logrado a través de las colas. Un componente (sea remitente o destinatario) se liga con una cola, posiblemente usando ligadura dinámica por nombre, y comienza a enviar o recibir mensajes de la cola.

JMS es sólo una API, no es una implementación de una plataforma middleware orientada a mensajes. De hecho, muchos MOM son compatibles con JMS. Son ejemplos de plataformas: Java Open Reliable Asynchronous Messaging JORAM [344] y JBossMQ [241] provenientes del mundo del código abierto. Por otro lado, FioranoMQ [168] es un ejemplo de producto comercial.

3.5.5. Sistemas de colas transaccionales

Otra de las características importantes de los MOM, orientada a proveer robustez para afrontar errores y fallas, es la posibilidad de hacer ***sistemas de colas transaccionales*** *(transactional queuing)*. Con la abstracción para colas transaccionales, el sistema MOM asegura la entrega fiable de mensajes. Es decir, el MOM se asegura que un mensaje enviado sea recibido por su servicio destinatario sólo una vez. Los mensajes tienen soporte en ***almacenamiento persistente*** si eventualmente el MOM sale de funcionamiento en el intervalo entre que se notifica de la llegada de un mensaje y su entrega; para poderse recuperar en la etapa previa de la falla del sistema.

Además de proveer garantía en la entrega de los mensajes, los sistemas transaccionales de colas pueden tratar adecuadamente las fallas. De hecho, se puede agrupar una cantidad de recepciones de mensajes y notificaciones dentro de una unidad atómica de ejecución. Es decir, las operaciones del grupo deben realizarse o todas o ninguna. Si no se han podido ejecutar todas las operaciones, debe hacerse un *rollback*; es decir, deshacer todos los efectos de la parte de las operaciones parciales que se realizaron en la transacción no completada.

En un sistema transaccional de colas, hacer un ***rollback*** en la recuperación de mensajes consiste en colocar todo el grupo de mensajes de la transacción nuevamente en la cola; de forma tal que pueda ser consumido nuevamente por la aplicación destinataria u otra aplicación en el caso de colas compartidas. Por otro lado, desde la perspectiva del remitente, hacer un rollback consistirá en borrar de la cola receptora el grupo incompleto de mensajes y solicitar el reenvío. El grupo completo de mensajes sólo será visible por el destinatario una vez que todos los mensajes fueron guardados correctamente en la cola.

3.6. Brokers de Mensajes

Los sistemas basados en RPC tradicionales y los middlewares orientados a mensajes (MOM) crean vínculos punto-a-punto entre aplicaciones; por ende son más bien estáticos e inflexibles a la hora de elegir el destinatario o la cola a la cual destinar su mensaje [273]. Los ***Brokers de Mensajes***, también conocidos como *Sistema de Mensaje Empresarial (Enterprise Messaging System)* [157], afrontan esta limitación actuando como *negociadores (brokers)* entre las entidades de sistemas, mediante la creación de una infraestructura de comunicación (lógica o física) centralizada para la integración de aplicaciones. Este modelo centralizado de comunicación tiene la forma de *"centro y rayos" ("hub and spoke")* [222] -en forma análoga a una rueda de bicicleta-, en el cual el broker de mensajes es el "centro" y los "rayos" son los vínculos con las aplicaciones que este integra. Los Brokers de Mensajes proveen flexibilidad para el enrutamiento de mensajes y otras características que dan soporte a la integración de aplicaciones empresariales.

Las ideas de Integración de Aplicaciones Empresariales *o* EAI *(Enterprise Application Integration)* y las tecnologías que derivan de ellas, son un paso más adelante en la evolución de los sistemas distribuidos. EAI extiende las capacidades de los middleware orientándose a la integración de aplicaciones, y no al diseño de una nueva lógica de aplicación (como se propone en el esquema 3-capas). Tales extensiones propuestas por EAI involucran cambios significativos en la forma en que los middleware son utilizados, desde la forma de programación (orientada a la integración y reuso) hasta la manera de interacción entre componentes (marcadamente asincrónica) [10].

Los Brokers de Mensajes son la tecnología de middleware que permite madurar estos conceptos de EAI. Las capacidades de middleware descritas de los Brokers de Mensajes, junto con el manejo asincrónico de mensajes que ellos soportan, es lo requerido para cualquier escenario demandado por configuraciones de EAI. Es por esta razón que los Brokers de Mensajes se consideran como la herramienta de EAI dominante en estos días. Los desafíos que la EAI plantea en su filosofía, reciben a los Brokers de Mensajes como primera respuesta tecnológica [273].

3.6.1. Contexto Histórico

El contexto histórico de los Brokers de Mensajes está intimamente ligado a las necesidades de EAI. La EAI, como paradigma tecnológico, plantea una serie de exigencias que no son del todo cumplidas por las plataformas de middleware convencional. La diferencia entre el middleware convencional y EAI es algo sutil. Para entenderlas en términos de requerimientos y sistemas reales, ayuda poder diferenciar entre desarrollo de aplicaciones e integración de aplicaciones.

Las arquitecturas cliente/servidor surgieron a medida que se aumentó las capacidades de ancho de banda en las redes y por un mejoramiento en la capacidades de cómputo de las PC y estaciones de trabajo. La funcionalidad, que antes residía solamente en una única ubicación física, comenzó a distribuirse en un conjunto de servidores. Al mismo tiempo, las empresas comenzaron a ser más descentralizadas y geográficamente dispersas y también dependientes de los sistemas informáticos. Como consecuencia, los servidores establecieron su presencia en cualquier dominio dentro de la organización. Debido a las limitaciones propias de la arquitectura cliente/servidor, estos servidores fueron islas de información; los clientes se podían comunicar con estos, pero no estos entre sí.

Cuando las arquitecturas 3-capas emergieron buscaban solucionar dos problemas. Primero, gracias a la separación de la lógica de aplicación del nivel de recursos se logró más flexibilidad en el sistema. Esto fue de mucha ayuda cuando las funcionalidades de un sistema se comenzaron a implementar en clusters de computadoras en lugar de servidores de alto porte. Segundo, los esquemas 3-capas sirven como mecanismos de integración de diferentes servidores. En este contexto, los middlewares pueden verse como la capa intermedia de una arquitectura 3-capas, siendo la natural ubicación para vincular varios servidores de recursos. Un ejemplo de esto es la utilización de Monitores de Transacciones para integrar diferentes motores de bases de datos.

El uso de los middlewares facilitaron la proliferación de servicios. De hecho, las arquitecturas 3-capas facilitan la integración de diferentes Gestiones de Recursos y, en general, de diferentes servicios. El resultado de una integración puede verse como un servicio en sí, el cual puede combinarse con otro y generar un nuevo servicio, y así indefinidamente generando la proliferación de servicios. La gran ventaja de este enfoque reside en que cada nueva capa de servicios provee un más alto nivel de abstracción que oculta la complejidad de aplicación y la lógica de integración. La desventaja es que en este escenario, la integración no se limita solamente a gestores de recursos o servidores, sino se trata con integración de servicios.

Desafortunadamente, mientras que se han obtenido significantes logros de estandarización para las interfaces de determinado tipo de servidores (por ejemplo los motores de base de datos), no se puede afirmar lo mismo para el caso de servicios. Mientras la integración de servicios sea dentro de una misma plataforma de middleware, no se presentan mayores inconvenientes, más allá de la complejidad propia de cada sistema. Pero los problemas de integración surgen cuando los servicios a vincular provienen de diferentes plataformas de middleware.

Así, las arquitecturas 3-capas brindan medios para unir las islas de información creadas por la proliferación de sistemas C-S, pero no cuentan con una manera general para integrar aplicaciones 3-capas; no presentan una infraestructura viable que pueda reducir la heterogeneidad y estandarizar las interfaces como así también las interacciones entre sistemas. La EAI surgen en respuestas a esta limitaciones. En resumen los middlewares convencionales sirven para integrar servidores residentes en el nivel de gestión de recursos; en tanto la EAI es una generalización de esta idea que incluye como bloques de construcción a las lógicas de aplicación de diferentes sistemas de middlewares.

Los Brokers de Mensajes son directos descendientes de los middleware orientados a mensajes o plataformas MOM. Ellos derivan del nuevo requerimiento expuesto por la EAI, relacionado a la necesidad de integración de aplicaciones empresariales heterogéneas y de alta granularidad como ser aquellas que intervienen en una cadena de suministros.

Una cadena de suministros de una empresa X puede contener los siguientes pasos: cotización, procesamiento de orden de compra y procedimiento de entrega. En el proceso de cotización involucra las solicitudes de los clientes para cotización de precios, fecha y condiciones esperadas de entrega de determinados bienes que vende la empresa X. Luego de esto, el cliente puede requerir la compra de los bienes y comenzará el procesamiento de esta orden de compra. En este paso, se verifica que la orden se corresponda con una cotización previa y si se puede cumplir con ésta en los tiempos que solicita el cliente. También la orden debe ser tratada en los sistemas de planificación de manofactura o de compras a proveedores para obtener los componentes necesarios para fabricar los bienes a vender. Por último, en el procedimiento de entrega se vinculan las funciones de envío a domicilio y de aspectos financieros relacionados con el cobro.

Cada uno de estos pasos está soportado por uno o varios sistemas informáticos y varios soportes de datos que posiblemente, también estén geográficamente separados en distintos departamentos o áreas organizativas, e incluso diferentes localidades. Estos sistemas informáticos pueden ser de desarrollo propio o paquetes "enlatados" adquiridos y adaptados. Algunas de las aplicaciones a integrar pueden ser el resultado de procesos previos de integración; pueden correr en sistemas operativos diferentes. Generalmente tienen interfaces de diferente tipo (algunas publicadas con IDL y otras con sintaxis propietarias) y distinto tipo de funcionalidades (algunas pueden ser transaccionales y otras no). Es posible que usen diferentes formatos de datos y produzcan información que no puede ser fácilmente adaptada para pasaje remoto de parámetros (como documentos multimediales). Pueden usar mecanismos de seguridad disímiles (algunas usan certificados X.509 en tanto que otras validacion UNIX usuario/clave). Y es probable que cada sistema tenga su propia infraestructura como así también su propio protocolo y modelo de interacción (una instalación DCE, un monitor de transacciones, un sistema basado en CORBA).

Como si estas heterogeneidades técnicas fueran pocas, se debe tener en cuenta los desafíos no técnicos: los sistemas a integrar son gobernados y operados por distintas áreas organizativas. Cada

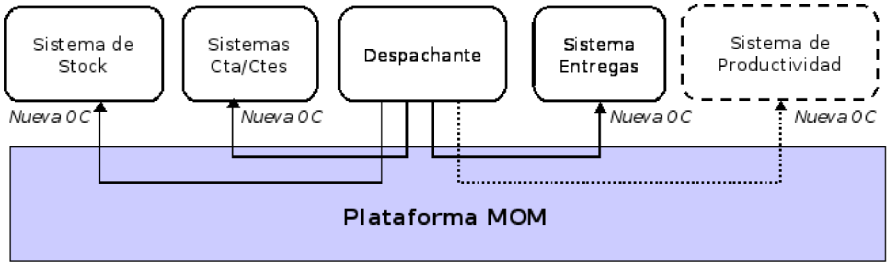

Figura 3.16: Con la interoperatividad propuestas por las plataformas RPCs y MOM, una aplicación Despachante debe cambiar en el caso de interactuar con una nueva aplicación (en líneas de puntos).

área es autónoma y su sistema también soporta tareas internas específicas cuyas funcionalidades no están alineadas con los objetivos de la aplicación integradora.

A pesar de todas estas dificultades, es primordial automatizar la cadena de suministro. Cuando una cadena de suministro no está automatizada, cada paso de un sistema a otro debe ser llevado manualmente con el respaldo de un documento de papel. Cuando esto ocurre, todo el proceso involucra una gran cantidad de tareas humanas, aumentando la posibilidad de errores e ineficiencias. Se torna imposible obtener un diagnóstico global del estado de una orden de compra. No es posible hacer trazabilidad ni monitoreo del proceso de suministro. Este contexto, daña la calidad del servicio al cliente; imposibilitando por consiguiente, el crecimiento sustentable de la empresa.

En este escenario las limitaciones de usar sistemas MOM para implementar EAI se ponen de manifiesto. Los sistemas MOM no proveen mecanismos para soportar la definición de lógicas sofisticadas para el ruteo de mensajes a través de diferentes sistemas y no ayuda al desarrollo para lidiar con la heterogeneidad de sistemas.

En respuesta a estas necesidades los Brokers de Mensajes extienden las capacidades de los sistemas MOM. En los sistemas MOM, la tarea de middleware sólo era mover mensajes de un punto a otro con ciertas garantías. Un broker de mensajes no solamente transporta mensajes, sino que también puede agregar lógica a los mensajes y tiene a cargo rutear, filtrar y procesar los mensajes a medida que estos se mueven a través del sistema. Como se explicará más adelante en la Sec. 3.6.3, una forma simplificada de ver a los Brokers de Mensajes [298], es como una extensión de las plataformas MOM con capacidades para soportar el esquema de interacción *publicar/suscribir*. Además, la mayoría de los brokers de mensajes proveen adaptadores (adapters) que enmascaran la heterogeneidad y hacen posible acceder a todo el sistema con el mismo modelo de programación y formato de intercambio de datos. Estas características son claves para soportar la EAI.

Los primeros productos que implementaban los brokers de mensajes aparecieron a los inicios de los 1990's, al mismo tiempo que la necesidad de EAI fue reconocida. En general las empresas de software aumentaron sus versiones de productos MOM. Algunos ejemplos de productos comerciales son Tibco ActiveEnterprise [499, 217], BEA WebLogic integration [39, 38, 40], WebMethods Enterprise [444, 445] y WebSphere MQ [232, 120].

3.6.2. Fundamentos

Para entender el impacto que tienen los Brokers de Mensajes es necesario entender las limitaciones de los sistemas basados en RPC y en MOM. Si se considera la situación de recibir una orden de compra en la cadena de suministro planteada antes, muchos sistemas deben procesar la compra recibida, además del propio sistema de gestión de orden de compras. Por ejemplo: el sistema de stock para verificar disponibilidad del producto, el sistema de cuentas corrientes para verificar el historial de pago del cliente y el sistema de entrega a domicilio para ver la factibilidad de realizar la entrega en el destino propuesto por el cliente (véase Fig. 3.16).

Con RPCs y MOMs, la aplicación que toma y despacha la orden de compra necesita tener una referencia (estática o dinámica) a cada una de las tres aplicaciones e invocar un método

determinado en cada una de estas (para el caso de RPCs) o enviar un mensaje a las respectivas colas (caso de MOMs). Este esquema de interconexión es aplicación-a-aplicación (o punto-a-punto); mientras más conexiones existan, más interfaces se deberán conocer, aumentando la complejidad en las aplicaciones que cumplen el rol de Despachantes [298].

Además se puede dar, y es una situación común en los ambientes organizacionales, que surja una nueva aplicación que requiera que se le notifique cada orden de compra que se genere, como puede ser un sistema de productividad de vendedores. En este caso, el sistema de gestión debe ser modificado para incluir el código necesario relacionado con la notificación adicional al sistema de productividad. También se puede dar, que el sistema que genera ordenes de compra no sea único, como es el caso de mantener ventas en salón y por Internet. En esta situación, el sistema de gestión de ordenes de compras y la aplicación de compras on-line deben notificar a los mismos sistemas. Cada nuevo sistema adicional que consuma información, origina que se modifiquen cada uno de los sistemas despachantes; por otro lado, al desarrollar un nuevo sistema productor de información, se deben tener en cuenta todas las aplicaciones que se deben notificar.

3.6.3. Extendiendo al MOM básico

La principal limitación de los ejemplos anteriores, es que en las plataformas MOM básicas, la responsabilidad de definir los receptores de los mensajes reside en los remitentes. Como se ve, este limitado esquema punto-a-punto comienza a ser complejo en escenarios dinámicos y en aquellos en los que el número de despachantes y receptores crece.

El aporte más importante que han brindado los Brokers de Mensajes como arquitectura de middleware es poder desacoplar a los extremos de una interacción gracias al soporte de nuevas formas de interacción. Una de las formas de interacción de aplicaciones o modelos más difundidos y ampliamente adoptado, es el paradigma ***publicar/suscribir*** *(publish/subscribe)*. La adopción de esta forma de interacción propicia a la formación de *Sistemas Basados en Eventos* [322], como una de las arquitecturas fundamentales para la integración de aplicaciones. Un ***Sistema Basado en Eventos*** consiste en eventos y notificaciones como medios de comunicación, productores y consumidores (de las notificaciones) como componentes interactuantes, suscripciones que significan el interés de un consumidor en determinadas notificaciones, el ***servicio de notificación de eventos*** que es el responsable de transmitir las notificaciones entre productores y consumidores. El Broker de Mensajes cumple la función de servicio de notificación.

Cualquier cosa de interés que pueda observarse dentro de una computadora es considerada un ***evento***. Una ***notificación*** es un conjunto de datos que reifican ese evento; es decir, lo describen. La notificación es creada por un observador del evento; éste, además de la información específica del evento, agrega información relacionada con las circunstancias en las que se produjo el evento. En general, varias notificaciones pueden describir el mismo evento individual pero desde puntos de vistas diferentes. Esto se puede dar por razones de seguridad, de fiabilidad del sistema, etc. O simplemente por diferentes formas de estructurar la notificación en modelos de datos diferentes. Los modelos de datos más comunes son pares nombre/valor [94], objetos [26, 122, 155], y datos semiestructurados [11, 321], como por ejemplo XML.

Las notificaciones son transportadas mediante mensajes, que son contenedores de datos a nivel de red que sirven para llevar datos entre dos puntos finales o extremos de una conexión. La interacción entre publicadores y consumidores es una interacción basada en intercambio de mensajes [157].

Los componentes de software de un sistema basado en eventos actúan como *productores* y *consumidores* de notificaciones de eventos. Los ***productores*** *(producers)* son los componentes que observan los eventos y tiene la decisión de publican las notificaciones. Las notificaciones publicadas no están dirigidas a ningún conjunto de destinatarios específicos; se envían al servicio de notificación de eventos, para su posterior distribución. Por ende, los productores no pueden anticipar ninguna reacción del lado del receptor.

Los ***consumidores*** *(consumers)* reaccionan a las notificaciones entregadas por el servicio de

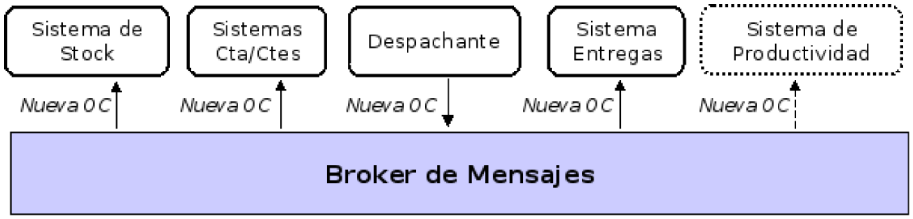

Figura 3.17: El modelo **publicar/suscribir** de los Brokers de Mensajes hace que la interoperatividad sea más flexible y robusta.

notificación. Estos componentes, también desconocen a su contraparte productora de notificaciones. Los consumidores establecen, en forma de suscripciones, las notificaciones que les interesan recibir (véase Fig. 3.17).

Una ***suscripción*** *(subscription)* describe el interés de un consumidor en un conjunto de notificaciones. Los consumidores registran su interés remitiendo las suscripciones al servicio de notificación. El servicio evalúa las notificaciones por cuenta y nombre del consumidor y entrega aquellas notificaciones que concuerdan con las suscripciones de los mismos. Las suscripciones son implementadas en forma de ***filtros*** *(filter)*, los cuales básicamente retorna un valor lógico booleano, para comprobar la idoneidad o no de una notificación. Adicionalmente, una suscripción puede contener parámetros para establecer las políticas generales de notificación, como por ejemplo credenciales de seguridad exigidas para acceder a determinado tipo de notificaciones [44], o tiempo de caducidad de notificaciones [100]. Las suscripciones pueden verse como las interfaces de entrada de los consumidores, las cuales describen los datos que están preparados para procesar.

Los ***anuncios*** *(advertisements)* son publicados por los productores para declarar las notificaciones que enviarán. También describen un conjunto de notificaciones y pueden preservar la forma de las suscripciones. Desde el punto de vista de la distribución, los anuncios sirven para establecer decisiones de ruteo, ya que el servicio de notificación puede establecer el origen de cada tipo de notificación. Los anuncios constituyen la interfaz de salida de los componentes productores.

La expresividad de las suscripciones en términos de capacidades de filtrado dependen del modelo de filtro y del modelo de dato. La combinación de ambos conforman el *esquema de clasificación de notificaciones*. Los modelos de datos más usados son los pares nombre/valor y datos semiestructurados. En cuanto a los modelos de filtro existen cuatro tipos que se distinguen: *canales*, *tópico o tema*, *tipo* y *basado en contenido*. A continuación se describen cada una de ellas:

- ***canales*** *(channels)*: es la forma más simple de suscripciones identificar un conjunto de notificaciones; los productores seleccionan, por el nombre, un canal en el cual publicar sus notificaciones. Los consumidores, por su parte, seleccionan un canal del cual obtener notificaciones. Un ejemplo de este enfoque el Servicio de Eventos de CORBA [357]. Por su parte, el Servicio de Notificación de CORBA [359], también ofrece manejo de canales, junto con otras formas de filtros de notificación.

- ***tópico o tema*** *(topic/subject)*: los filtros basados en tópicos o temas usan concordancia de cadenas de caracteres para la selección de notificaciones [346]. Los publicadores envían cada notificación con un string que representa la temática, denotando un nombre e un espacio de nombres conformado por un sistema jerárquico de tópicos. Por ejemplo `Documentos/Internos/CadenaSuministro/NuevaOrdenDeCompra`. En estos casos una aplicación puede suscribirse a un conjunto de subtipos de un tipo base; por ejemplo, usar la codificación `Documentos/Internos/CadenaSuministro/.*` serviría para suscribirse a todos los mensajes cuyo tipo tenga un nombre de espacio que comience con la forma dada. La simplicidad de este modelo de filtro es su principal desventaja. Exige que solamente

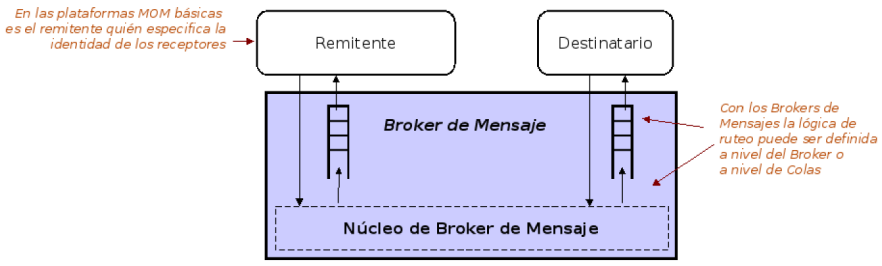

Figura 3.18: Los Brokers de Mensajes permiten a los usuarios definir lógica de enrutamiento de mensajes.

se respete una única jerarquía de espacios de nombres para clasificar una notificación. Al ser el espacio de nombre un camino en una estructura jerárquica, no se admiten múltiples tópicos para un mensaje, y deben clasificarse desde un único punto de vista.
Por ejemplo, una clasificación alternativa al mensaje sería `Ventas/Documentos/Afectaciones/NuevaOrdenDeCompra`. Para tener en cuenta la asociación entre notificación y tópico, esta debe regenerarse con el tópico diferente; esta situación genera un crecimiento exponencial en la generación de notificaciones y en los árboles de espacios de nombres.

- ***tipo*** *(type)*: el filtro basado en tipos, usa las expresiones del camino en el espacio de nombres y pruebas de inclusión de subtipos para seleccionar notificaciones [155, 35]. Al permitir herencia múltiple, el espacio de nombres se convierte en una malla, que admite diferentes caminos para el mismo nodo.

- ***basado en contenido*** *(content-based)*: el filtro basado en contenido es el esquema de selección de notificaciones más ampliamente utilizado [93, 320]. Los filtros son evaluados en la integridad del contenido de la notificación. La expresividad del filtro está dada por el modelo de datos de la notificación y los predicados aplicados. Por ejemplo, se pueden establecer filtro a partir de simples comparaciones o buscando correspondencia con plantillas de expresiones regulares. Por ejemplo, una aplicación se suscribe para recibir mensajes a través de aserciones relacionadas a los parámetros del mensaje. Por ejemplo: `tipo:“NuevaOrdenDeCompra” AND cliente=“ACME co” AND Cantidad >1200`.

Los brokers de mensajes pueden soportar varios sistemas de filtros, brindando la posibilidad de definir distintas reglas de ruteo específicas de aplicación. Los brokers de mensajes mejoran los sistemas MOM y solucionan esta limitación factorizando la *lógica de ruteo* de los mensajes fuera de las aplicaciones remitentes y colocándola en el middleware [10]. En un broker de mensajes se puede establecer una regla de lógica de ruteo que identifica a través de filtros, para cada mensaje, cuáles serán sus aplicaciones receptoras o componentes consumidores (véase Fig. 3.18).

La ventaja fundamental de este enfoque es que, sin interesar cuantas aplicaciones despachantes de un mensaje existan, hay un único lugar en donde se gestiona la lógica de integración; el mismo broker de mensajes que cumple la función de servicio de notificación de eventos. La lógica de ruteo puede estar definida en el broker de mensajes o en las mismas colas. Si está en el broker de mensaje, se aplicará a todos los mensajes para destinarlos adecuadamente. Si está en las colas, se define qué clase de mensajes son de interés para la cola.

De por sí, los sistemas MOM tienen una limitada forma de desacople, ya que las aplicaciones envían mensajes a las colas, antes que a las aplicaciones en sí. Puede que la aplicación receptora no se encuentre en línea en el momento que se recibe el mensaje, permaneciendo en la cola hasta que la aplicación se active. Con la posibilidad de implementación de un sistema basado en eventos y su esquema de interacción publicar/suscribir, los Brokers de Mensaje extienden las capacidades de desacople de las plataformas MOM, agregando transparencia de identidad entre los múltiples extremos de una interacción [157].

De la misma forma, el uso de colas compartidas provee de alta disponibilidad al sistema. Las aplicaciones que consumen mensajes de una cola compartida son generalmente del mismo tipo o son diferentes hilos de ejecución de la misma aplicación. Con este esquema de colas compartidas se logra obtener balanceo de carga en el procesamiento de mensajes. Si se tiene en cuenta la posibilidad de desacoplamiento, brindado por los Brokers de Mensajes, y el balanceo de carga, ofrecido por las colas compartidas, se puden afirmar que los middleware de este tipo ofrecen gran flexibilidad y adaptabilidad.

En la actualidad, todos los brokers de mensajes soportan el paradigma suscribir/publicar. Han existido intentos de estandarizar las interfaces y APIs entre las aplicaciones y el middleware. La especificación de Sun, Java Massage Service (JMS) [463], citada como ejemplo de interacción de colas en MOM, tambien posee especificaciones para manejo de broker de mensajes. La especificación JMS incluye la API para los modelos de interacción punto-a-punto y publicar/suscribir. Respecto a este último, las publicaciones y suscripciones se hacen basándose en filtros de tópicos.

Yendo más allá de la lógica de ruteo que se puede establecer, también es posible implementar en el middleware funcionalidades más específicas a la aplicación, por medio de ***reglas de transformación de contenidos***. Por ejemplo, se pueden incorporar reglas de este tipo a las colas, en los casos en dónde se requieran que determinados datos de mensajes de un determinado tipo, sean transformados en otro tipo de dato requerido como parámetro en la aplicación receptora. El caso típico se da cuando un mensaje tiene una magnitud de medida expresada en el sistema métrico, por ejemplo. La aplicación receptora de este mensaje trabaja con el sistema inglés de medida. Una regla de transformación consistiría indicar el algoritmo de transformación de un tipo a otro, de forma tal que ni el emisor ni el receptor cambien su código.

En general no hay límite en cuanto a la "cantidad" de lógica de aplicación que se puede incorporar en el broker o en las colas. La decisión de ubicar reglas en un lugar u otro es una decisión de diseño. No es buena idea colocar toda la lógica en el broker; si bien esto logra aplicaciones más genéricas y robustas, el hecho de procesar varias reglas específicas de aplicación por cada mensaje enviado en el sistema, degrada la latencia y el rendimiento general. Si distribuimos la lógica hacia las colas de aplicaciones, se logra mejor tiempo de latencia y disponibilidad, pero se hace difícil la depuración de fallas y el mantenimiento.

3.6.4. Administración Distribuida de un Broker de Mensajes

Los brokers de mensajes incluyen soporte para un ***administrador***; es decir, un usuario especial que tiene las capacidades para definir los tipos de mensajes que pueden ser enviados y recibidos, e indicar cuáles usuarios tiene privilegios para enviar y recibir mensajes pudiendo establecer reglas de ruteo específicas. Si bien el administrador está presente en los sistemas MOM básicos, es en los brokers de mensajes en dónde tiene real significación ya que es el responsable del alto grado de desacoplamiento presente entre las aplicaciones.

Los brokers de mensajes son plataformas naturalmente extensibles para sustentar a aplicaciones con necesidades de comunicación intensivas, que pueden abarcar diferentes dominios organizacionales (posiblemente diferentes departamentos, áreas o empresas). De hecho es posible componer varios brokers de mensajes. Por ejemplo, se puede pensar en una aplicación XA, cliente de un broker BA, que desea suscribirse a determinados tipos de mensajes que genera una aplicación YB, cliente de otro broker BB. En este caso, XA debe suscribirse primero en su broker BA, BA a su vez se suscribirá al broker BB. Cuando YB genere un mensaje del tipo deseado, BB lo distribuirá a todos los suscriptores, el broker BA entre ellos. Cuando el mensaje llega a BA, pasando la frontera entre brokers, BA lo distribuye a los suscriptores, entre ellos XA.

En realidad, los brokers BA y BB son clientes entre sí; la única diferencia es que tienen diferente dominio administrativo, por consiguiente distinto usuario administrador. Para el ejemplo, BB deberá establecer permisos de suscripción para BA, en tanto que BA deberá establecer permisos de publicación para BB. El sistema de administración y seguridad se mantiene simple y modular con este enfoque, ya que solamente cada administrador de broker debe configurar los

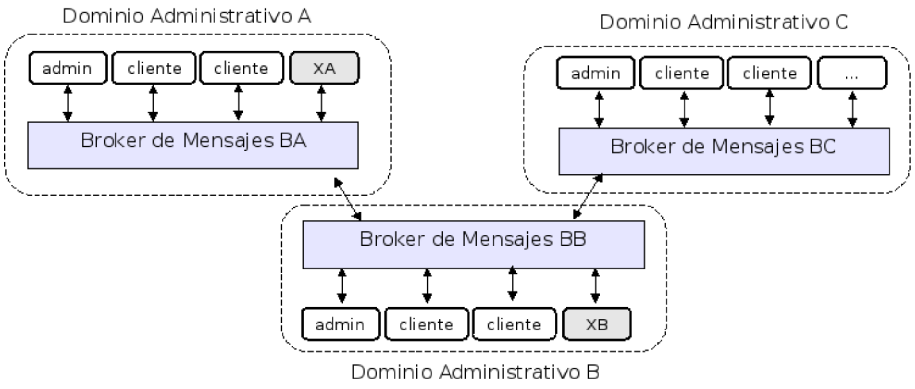

Figura 3.19: Los clientes pueden interoperar a través de múltiples brokers de mensajes de diferentes dominios administrativos.

permisos adecuados para publicación y suscripción (véase Fig. 3.19).

3.6.5. EAI con Brokers de Mensajes

Por las características de los Brokers de Mensajes, es posible pensar en ellos como una solución a la problemática planteada en la EAI. En la Fig. 3.20 se ve los componentes fundamentales para un escenario de EAI. Los componentes esenciales son:

- Los **adaptadores**, encargados de realizar el mapeo de los formatos de datos, interfaces y protocolos hacia un modelo y formato común. El propósito de los adaptadores es ocultar toda la heterogeneidad y presentar una vista unificada de los diferentes sistemas subyacentes de la integración. Cada aplicación a ser integrada necesita contar con un adaptador. Los adaptadores deben configurarse para suscribirse a los mensajes adecuados, de forma tal que cuando uno de estos mensajes se reciba, se ejecute la acción apropiada en la aplicación subyacente.

- El **broker de mensajes**, encargado de facilitar las interacciones entre adaptadores; en definitiva, la interacción entre los sistemas back-end que requieren integrarse.

- Una **aplicación original** que implementa la lógica de integración. Esta aplicación es la que interactúa con el broker de mensajes y, consecuentemente, con las aplicaciones detrás de los adaptadores. Esta aplicación fue desarrollada teniendo en cuenta la plataforma de Broker de Mensajes elegida, y su lógica de procesamiento se basa en la publicación y recepción de mensajes.

Todo middleware está orientado a la integración de aplicaciones. Sin embargo el término plataforma de EAI es usado para referirse a la plataforma de software orientada a integrar aplicaciones disimiles, heterogéneas y de alta granularidad.

Los Brokers de Mensajes no son la única forma de solucionar los problemas planteados por la EAI. De hecho los Monitores de Transacciones (véase Sec. 3.3.4) proveen una forma limitada de funcionalidades de integración. Sin embargo, los Brokers de Mensajes fueron creados con ese propósito y proveen más recursos al efecto; como ser en términos del número y variedad de tipos de adaptadores ofrecidos, mecanismos de personalización y extensibilidad, y herramientas para el desarrollo de nuevos adaptadores.

Independientemente de lo que plantea cada marco particular de integración, se puede decir que el uso de Brokers de Mensajes se caracteriza por las siguientes ventajas:

- *Bajo costo de desarrollo.* La integración es un proceso simple y puede ser llevado a cabo prontamente. La simplicidad viene del bajo grado de acoplamiento que propone la arquitectura en general (en contraste con los sistemas altamente acoplados que proponen los

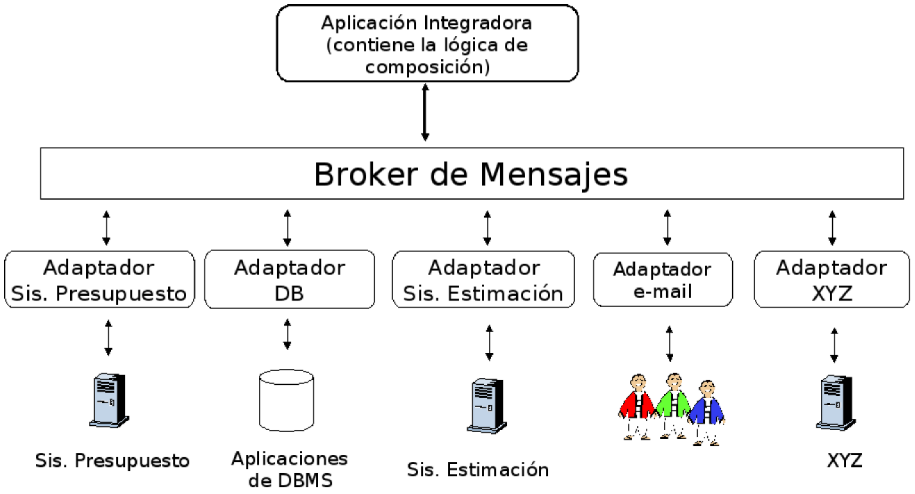

Figura 3.20: Componentes de un escenario de EAI. Los clientes pueden interoperar a través de múltiples brokers de mensajes de diferentes dominios administrativos.

Monitores TP y los Brokers de Objetos). El mayor esfuerzo en el desarrollo consiste en la construcción y/o configuración de los adaptadores.

- *Bajo costo de oportunidad*: debido a que la integración se puede efectuar en forma rápida, la automatización (y su correspondiente ahorro en recursos) también es lograda prontamente. Esto afecta directamente la capacidad de la organización para desarrollar nuevos servicios y expandirse. Los Brokers de Servicios, es una plataforma tecnológica que facilita la rápida reacción a nuevas demandas de servicios empresariales, siendo una mejor opción que soluciones alternativas (middlewares basados en RPCs o Motores de Transacciones).

- *Bajo esfuerzo de mantenimiento*. Las funcionalidades que hacen que cada aplicación backend esté disponible para integración residen en un único módulo: el adaptador. Si esta aplicación es modificada, solamente debe ser modificado el adaptador. No es necesario cambiar las aplicaciones que implementan la lógica de integración.

A pesar de estas ventajas, los Brokers de Mensajes no son la panacea. De hecho, optar por esta tecnología para realizar integración de aplicaciones tiene sus desventajas. La principal es el costo del licenciamiento. Generalmente la licencia de middleware es costosa, como así también las licencias de los adaptadores. Además del costo de licenciamiento, se debe contar con profesionales calificados que reciban el entrenamiento y equipamiento adecuado para la instalación y operación de la herramienta.

Esta situación desalienta a las pequeñas y medianas empresas para adoptar plataformas de EAI. Sin embargo, en las grandes organizaciones la situación es diferente. Por empezar, las grandes empresas tiene problemas y necesidades más complejas para integración. Es común que los sistemas a integrar sean de diferentes áreas organizativas; y es común que en las grandes empresas, estas áreas sean autónomas para elegir plataformas de desarrollos y equipos de desarrollo. Por último, las grandes empresas pueden afrontar los costos económicos de la adquisición de licencias, del entrenamiento de profesionales y de la adquisición de nuevo equipamiento para instaurar plataformas de integración. Los grandes proyectos de integración presentes en tales compañías, justifican la inversión.

La condición de tecnología onerosa de las herramientas de EAI, se tiende a revertir en los años actuales. La adopción de los estándares propuestos por la tecnología de Servicios Web alivian los esfuerzos económicos y facilitan el diseño y uso de middlewares de integración.

3.6.6. Bajo Acoplamiento y Reusabilidad

El verdadero poder de las plataformas MOM y de los Brokers de Mensajes está en lograr bajo acoplamiento entre las aplicaciones que integra. En un esquema básico, como el mostrado en la figura 3.13, la aplicación cliente manda sus mensajes de `solicitudPresupuesto` a traves de un MOM a una cola asociada a la aplicación específica, el `"Sistema de Presupuesto"`. Esta aplicación es la que provee las capacidades para contestar el mensaje. De igual forma, bajo un Broker de Mensajes, el cliente publica sus peticiones y el servidor las consume y, en concordancia, el servidor publica las respuestas y que el cliente consume.

Pero, más adelante, se podría reemplazar al `"Sistema de Presupuesto"`, por otro sistema más moderno, con mayores funcionalidades, desarrollado en otra plataforma diferente al sistema original. Sin embargo, la aplicación cliente no notará el cambio; seguirá enviando el mensaje original `solicitudPresupuesto` y éste seguirá siendo contestado. Las únicas exigencias lógicas a esta flexibilidad en el sistema, es que el nuevo proveedor de servicios siga proveyendo el servicio de cálculo de presupuesto bajo la misma interfaz original y que se asocie a la cola respectiva o realice las correspondientes publicaciones.

De la misma forma, la aplicación cliente puede ser reemplazada, o mejor aún, puede existir otra aplicación diferente totalmente diferente a la primera. En ambos casos, la exigencia es que las nuevas aplicaciones clientes respeten la API de la aplicación servidor [157].

Desde ya que el bajo grado de acoplamiento ofrecido por las plataformas MOM y Brokers de Mensajes, facilitan y promueven el reuso de aplicaciones. En el ejemplo, es importante contar con un único sistema de cálculo de presupuesto para evitar redundancia de servicios, evitando las posibilidades de diferentes cálculos a iguales parámetros. Un único módulo de cálculo de presupuesto es más fácil de mantener y ajustar, que varios. Los MOMs favorecen la integración de estos componentes reutilizables proveyendo interfaces y gestionando la recepción y reenvío de las peticiones o mensajes destinadas a estas aplicaciones a través de sistemas de colas. Los Broker de Mensajes mejoran este desacoplamiento y posibilidad de reuso, instaurando la figura de un negociador en el cual descansa completamente la lógica de ruteo, permitiendo que el incremento de partes interconectadas crezca sin problemas, bajo un mismo sistema de interfaces.

Por otro lado, el marco de trabajo para el desarrollo de aplicaciones a ser integradas bajo plataformas MOM y Brokers de Mensajes, exige que se desarrollen las mismas respetando una interfaz de servicio, sin necesidad de conocer totalmente las características de las aplicaciones clientes que usarán el servicio. Las interfeces de servicio son la esencia de los diseños orientado a la integración de aplicaciones. Combinadas con el uso de estándares, las interfaces son el ingrediente básico para lograr bajo acoplamiento en la comunicación entre consumidor y proveedor de servicio, en forma transparente al lenguaje y la plataforma particular de cada uno de ellos [413].

El desarrollo de componentes en base a interfaces de servicio, con independencia de las características propias de las aplicaciones consumidoras del servicio, es un marco propicio para lograr políticas de reutilización. En particular las aplicaciones expuestas a través de una interfaz de servicio operan como *"cajas negras"*; el reuso sistémico alienta el reuso basado en "cajas negras" antes de promover el cambio y la adaptación de código de las aplicaciones existentes (reuso de "cajas blancas") [393].

Por lo expuesto, las plataformas MOM y los Brokers de Mensajes no sólo promueven la integración de aplicaciones a partir del reuso, sino el desarrollo de aplicaciones y componentes para reuso [283].

3.7. Middleware Basado en Agentes

Una de las motivaciones más importantes para relacionar los principios de middleware con los sistemas de *Middleware basados en Agentes* es lograr una plataforma tecnológica que brinde posibilidades de realización y desarrollo para la *computación móvil* y *ubicua*.

Cuando se hace referencia a ***computación móvil*** o ***computación ubicua*** [534], se las presenta como evolución de la computación distribuida. Se ha comenzado a utilizar el término computación móvil a principios de los 90's cuando los dispositivos personales y redes inalámbricas permitieron la movilidad de los usuarios de los sistemas distribuidos; este avance tecnológico trajo consigo las lógicas complicaciones de direccionamiento, de calidad y capacidad cambiante de las redes por utilizar otros medios de interconexión (como las ondas de radio), de adecuación a nuevos protocolos de red, de transporte y de presentación para una nueva gama de dispositivos móviles. Una gran cantidad de soluciones se presentaron a estos nuevos planteos; como perfil común a todas ellas, existía el desafío de lograr interconectar a las aplicaciones y ofrecer los servicios al usuario final, tratando de tener los mismos resultados como si éste estuviera conectado a una red fija [514].

En la actualidad, gracias a la evolución de la tecnología de hardware, existen un número creciente de dispositivos: teléfonos móviles, PDAs, organizadores, que son portables debido a su tamaño. Estos dispositivos tienen capacidades de cómputo y además pueden comunicarse con otros elementos sin necesidad de conexiones físicas, gracias al desarrollo y evolución de protocolos inalámbricos, tanto en redes celulares (GPRS [16] y UMTS [503, 247]), como en redes locales (WLAN [123] y Bluetooth [67, 80]). En la actualidad, es más o menos habitual, que estos dispositivos proporcionen nuevos servicios: por ejemplo, desde un teléfono celular es posible, además de sostener una conversación telefónica, mandar mensajes de texto, consultar la ubicación de la farmacia más cercana y eventualmente programar un lavarropas.

La computación móvil dio paso a lo que se denomina como ***computación ubicua***, cuando los dispositivos con capacidades de computación y comunicación no eran solamente dispositivos personales, sino casi cualquier tipo de dispositivo físico que rodea al usuario: sensores que capturan la información del entorno o del propio usuario, actuadores que permitan ejecutar acciones de forma remota sobre elementos físicos (puertas, interruptores, etc), en electrodemésticos (lavarropas, impresoras, máquinas de fax, etc.) y sistemas de transporte (coches, autobuses, etc) [208]. Además es necesario que estos dispositivos tengan la capacidad de operar sin infraestructura fija, lo que implica no sólo la necesidad de utilizar protocolos inalámbricos, sino que será necesario proporcionales autonomía; es decir, poder comunicarse con otros dispositivos sin necesidad de dispositivos intermedios, típicamente servidores, para construir nuevos servicios.

Las nuevas restricciones y características que imponen estos entornos, plantean una evolución de los conceptos, ya maduros, para entornos de redes fijas en las que los nodos eran computadoras. No basta con migrar los sistemas y aplicaciones existentes a estos nuevos entornos [32]; debido a que existen una mayor diversidad de dispositivos, con diferentes limitaciones hardware para ejecutar aplicaciones y en los que hay que tener en cuenta que las comunicaciones pueden ser costosas y las distancias mucho más significativas que en las redes tradicionales.

Este nuevo paradigma exige soluciones de las herramientas de desarrollo de software para soportar la construcción de aplicaciones para entornos de computación móvil y ubicua. De la misma forma, las tecnologías de plataformas de redes generó la definición de nuevos protocolos de red inalámbrica para la comunicación directa entre dispositivos móviles y con redes fijas, brindando la adecuada transparencia.

Además, la mayoría de estos nuevos dispositivos no son multipropósito, sino que proporcionan una serie de servicios concretos; pero gracias a la interacción entre estos, es posible lograr cooperación para componer nuevos servicios, de forma inteligente, que satisfagan las expectativas del usuario. Para lograr este nivel de adecuación, es necesario que los dispositivos descubran dinámicamente las capacidades de otros dispositivos, establezcan sus intencionalidades, se comuniquen con protocolos establecidos y planifiquen y reestructuren sus capacidades para ofrecer servicios más adecuados a las necesidades del usuario.

Es indudable que en el escenario descripto hasta aquí, subyace la ineludible responsabilidad que debe afrontar la tecnología de middleware que soporte tales desafíos. Las plataformas de middleware no solamente deben adaptarse a las nuevas configuraciones de transporte de datos

transparente sobre redes heterogéneas inalámbricas y/o fijas; sino que además deben asimilar a una variedad, cada vez más creciente, de dispositivos con diferentes capacidades de procesamiento y almacenamiento, que demandan la ejecución de aplicaciones distribuidas adaptadas a sus interfaces de usuarios particulares.

Sumado a los problemas de interoperatividad y disponibilidad de aplicaciones distribuidas en estos entornos, la exigencia de comunicación y cooperación de dispositivos para componer y adaptar servicios, demanda que la lógica de aplicación de tales sistemas se diseñe bajo un nuevo paradigma más adecuado al nivel de autonomía, de escalabilidad y movilidad que exigen tales aplicaciones. De esta forma, surge la vinculación entre los sistemas distribuidos y el *paradigma de Agentes Inteligentes* [243].

Las plataformas de ***Middleware basado en Agentes*** son un paso más en la evolución de los sistemas informáticos distribuidos. Sustentan los sistemas informáticos capaces de operar bajo las reglas de la computación ubicua, en la forma de *sistemas multiagente*.

Existen varios esfuerzos para materializar tales ideas, algunos con más desarrollo que otros. Algunas plataformas y sistemas que vale la pena destacar son: OPAL (The Otago Agent Platform) [401, 329], UbiMAS (Ubiquitous Mobile Agent System) [28, 29], EMAA (Extensible Mobile Agent Architecture) [428], DIET (Decentralised Information Ecosystem Technologies Agents Platform) [279, 188], Cougaar (Cognitive Agent Architecture) [108], CAN (Content Addressable Network) [403] y SWAN (Small World Adaptative Networks) [70].

En este trabajo se explicará la plataforma *JADE (Java Agent DEvelopment Framework)* como modelo de middleware orientado a sistemas multiagentes. ***JADE*** *(Java Agent DEvelopment Framework)* [485] es una plataforma de middleware para el desarrollo y la ejecución de aplicaciones par-a-par basadas en el paradigma de agentes las cuales pueden trabajar e interoperar en forma transparente en redes de ambientes cableados e inalámbricas [42, 43].

JADE tiene dos aspectos centrales en su modelo conceptual: (1) la topología de sistema distribuido basado en redes *par-a-par*, y (2) la arquitectura de componentes de software basada en el *paradigma de agentes* [539, 244, 243].

La topología de red define cómo los componentes se interconectan entre sí; mientras que la arquitectura especifica qué debe esperar un componente de otro. Dicha arquitectura está presente en la mayoría de las plataformas de Middleware basado en Agentes.

Si se piensa en interoperatividad de componentes de software a nivel de aplicación es marcadamente necesario tratar con estándares. Una de las principales ventajas de Jade es su compatibilidad con las especificaciones FIPA [170].

3.7.1. El modelo de red "par-a-par"

El modelo cliente-servidor C-S es el modelo más difundido y mejor conocido para aplicaciones distribuidas. Como se vio en Sec. 2.2.2, este modelo se basa en una marcada distinción de roles entre los nodos clientes (aquellos que requieren servicios) y los nodos servidores (aquellos proveedores de recursos). Los servidores en el modelo C-S, son los que proveen servicios o, en forma más general, las capacidades del sistema distribuido. Sin embargo, estos no son capaces de tomar ninguna iniciativa, ya que son completamente reactivos; es decir esperan a que sus servicios sean invocados por un cliente. Por el contrario, son los clientes los que concentran toda la iniciativa del sistema. Los clientes proactivos son los que se comunican con los servidores, usualmente a partir de un requerimiento del usuario, pero no proveen ninguna funcionalidad.

Los clientes pueden aparecer o desaparecer en cualquier momento; generalmente, tienen direcciones dinámicas. Por el contrario, los servidores deben proveer alguna garantía de estabilidad y disponibilidad, y casi siempre tienen una dirección bien conocida y estática. En este contexto, los servidores no pueden contactarse con los clientes, hasta que estos tengan la iniciativa y decidan activar una sesión de comunicación con el servidor.

La web [54] es un claro ejemplo de aplicación basada en el modelo cliente servidor. Los servidores son todos los sitios/portales, los cuales tienen completamente la lógica de aplicación y

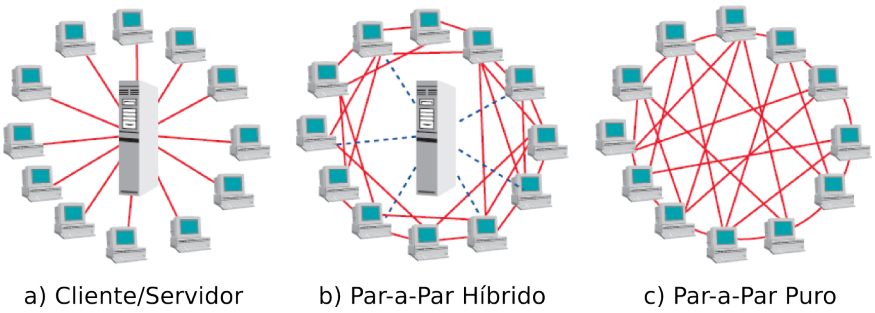

Figura 3.21: Modelo Cliente/Servido y Modelos P2P.

la gestión de recursos. Los clientes son los navegadores de internet cuya única tarea es recuperar, bajo una petición explícita del cliente, información localizada en sitios de Internet y presentarlos.

Sin embargo, existen otras aplicaciones que no adoptan este modelo. Ejemplo de esto son las aplicaciones simples de chat, los sistemas para compartir archivos (como el protocolo eDonkey [156]) o un juego multijugador. Estas aplicaciones requieren que los nodos activos se comuniquen entre sí. Aunque tales sistemas se pueden implementar en una arquitectura cliente/servidor, se perdería las ventajas ofrecidas por otras arquitecturas y prácticas de software; en particular el modelo *par-a-par* resulta ser más adecuado.

En un modelo *par-a-par o P2P (peer-to-peer)* [424, 431, 317], a veces mal denominado punto-a-punto[5], no existen más las distinciones entre roles. De hecho cada ***"par"*** o nodo es capaz de tener iniciativa proactiva de comunicación, cada nodo puede inicializar la comunicación, ser sujeto u objeto de una petición, ser servicio y proveer una funcionalidad. La lógica de aplicaciones está distribuida entre todos los pares en la red; cada nodo es capaz de descubrir a nuevos nodos y puede entrar, agregarse y dejar la red a voluntad. El sistema es completamente distribuido a través de la red.

Una de las grandes diferencias entre los dos modelos es la forma en que los nodos pueden ser descubiertos. En un sistema C-S, los nodos clientes deben conocer los nodos servidores, pero no es necesario que los clientes se descubran entre sí. De la misma forma, escapa al sentido del modelo, que los servidores conozcan a los clientes.

En un sistema P2P, estas relaciones son completamente arbitrarias porque los nodos no tienen roles definidos. El sistema debe proveer servicios que le permita a cada par entrar, unirse y dejar la red en cualquier momento, de la misma forma se debe contar con capacidades para buscar y descubrir nuevos nodos. Estos servicios son las *"paginas blancas"* y las *"páginas amarillas"* del mecanismo que permite publicar y descubrir las características y los servicios ofrecidos por cada par. Hay dos modelos básicos para la implementación de estos mecanismos. El modelo P2P *puramente descentralizado* y el modelo *P2P híbrido*, también llamado *"de índice centralizado"* (véase Fig. 3.21).

En el ***modelo P2P puro*** (Fig. 3.21 c)), la red es completamente descentralizada, los nodos son completamente autónomos. La ausencia de un nodo de referencia hace difícil mantener la coherencia de la red en su conjunto y el descubrimiento de los pares. Los pares deben identificarse uno a uno en forma recíproca a medida que se descubre, lo que genera complejidad y un aumento exponencial en el ancho de banda a medida que aumenta los nodos de la red. La seguridad no es fiable ya que cada nodo es ingresado en la red sin ningún tipo de autenticación.

En el ***modelo híbrido*** (Fig. 3.21 b)), existe un nodo especial encargado de proveer los servicios de búsqueda y descubrimiento de los nodos activos. Mantiene además una lista de las capacidades y servicios provistos por cada par; por tal motivo se lo denomina nodo índice. Este tipo de redes generan menos tráficos y permiten proveer mecanismos de seguridad más fiables

[5]En un modelo punto-a-punto pueden existir distinción de roles. Como por ejemplo la interconexión punto-a-punto entre aplicaciones a través de modelos RPCs (Sec 3.2.2) o MOM (Sec. 3.5.2)

por exigir a cada par autenticarse en el nodo índice. Sin embargo, la principal desventaja del modelo consiste en la necesidad que los nodos índices tengan una alta tasa de disponibilidad, haciéndolos el punto central de falla y ataques.

3.7.2. El paradigma de agentes

El ***paradigma de agentes*** aplica conceptos de Inteligencia Artificial y Teoría del Acto del Habla a la tecnología de objetos distribuidos [245]. El paradigma se basa en la abstracción de agente. Un ***agente inteligente*** es un proceso computacional capaz de realizar tareas en forma autónoma, y se comunica con otros agentes para resolver problemas mediante cooperación, coordinación y negociación. Habitan en un entorno complejo y dinámico con el cual interaccionan en tiempo real para conseguir un conjunto de objetivos [540]. Las propiedades indispensables de un agente son:

- **Autonomía**: es la capacidad de operar sin la intervención directa de los humanos, y de tener algún tipo de control sobre las propias acciones y el estado interno.
- **Sociabilidad/Cooperación**: los agentes han de ser capaces de interactuar con otros agentes a través de algún tipo de lenguaje de comunicación, para contribuir a alcanzar los objetivos integrales del sistema distribuido.
- **Reactividad**: los agentes perciben su entorno y responden en un tiempo razonable a los cambios o eventos detectados.
- **Pro-actividad** o **iniciativa**: deben ser capaces de mostrar que pueden tomar la iniciativa en ciertos momentos.

Otras propiedades destacables serían:

- **Movilidad**: posibilidad de moverse a otros entornos a través de una red electrónica.
- **Continuidad temporal**: los agentes están continuamente ejecutando procesos.
- **Veracidad**: un agente no comunicará información falsa premeditadamente.
- **Racionalidad**: el agente ha de actuar para conseguir su objetivo.
- **Aprendizaje**: mejoran su comportamiento con el tiempo.
- **Inteligencia**: usan técnicas de IA para resolver los problemas y conseguir sus objetivos.

Un ***sistema multi-agente*** *(MAS)* [242] es aquel en el que un conjunto de agentes cooperan, coordinan y se comunican para conseguir un objetivo común. Las plataformas de Middleware basados en Agentes son sistemas multi-agente y como tal *sistemas basado en agentes* son intrínsecamente par-a-par [381, 317]. Cada agente es un par que potencialmente puede iniciar una comunicación con otros agentes; del mismo modo es capaz de proveer capacidades para el resto de los agentes del sistema. El rol de la comunicación es esencial en un sistema basado en agentes, y el modelo tiene las siguientes tres características:

1. *Los agentes son entidades activas, pueden decir "no" y son bajamente acoplados.* Esta característica fundamenta la utilización de comunicación basada en mensajes asincrónicos entre agentes. Esto permite que el receptor de un mensaje pueda seleccionar a cuáles mensajes tratar y a cuáles descartar, y en qué orden. Como se explicó en Sec 3.5.2, el esquema asincrónico deslinda toda posible dependencia temporal entre emisor y receptor: la ejecución del emisor no se bloquea al enviar un mensaje, el receptor puede no estar en línea cuando se recibe un mensaje. En el caso extremo, logrando amplio desacople entre componentes como en los escenarios descritos en la Sec. 3.6, puede no conocerse ni la ubicación ni el nombre

del receptor; basta con definir su intencionalidad o preferencia, por ejemplo el destino de un mensaje puede ser filtrado por un tópico como ser "todos los agentes interesados en fútbol".

2. *Los agentes llevan a cabo acciones, y la comunicación es un tipo de acción.* El hecho de colocar a la comunicación dentro del rango de acciones posibles que puede tomar un gente, implica que pueden incluirse en planificaciones. Por ejemplo, un agente puede diseñar una planificación que tenga acciones físicas (p. ej.: doblar a la izquierda) y acciones comunicativas (p. ej.: pedir habilitar el acceso a un registro). Al ser las comunicaciones planificables implica que las precondiciones y efectos de cada comunicación necesitan estar debidamente especificados.

3. *La comunicación conlleva una semántica.* Un agente objeto de una acción comunicativa (es decir el receptor de un mensaje) debe entender perfectamente el significado de dicha acción y, en particular, los motivos por los cuales la acción ha sido ejecutada (es decir la intención del emisor). Esta propiedad exige la existencia de una semántica universal y la necesidad de un estándar dentro del sistema.

Inspirados por la necesidad de lograr interoperatividad a través de diferentes plataformas y operadores para posibilitar el desarrollo de sistemas multiagentes, en 1996 el *Laboratorio de Telecom Italia TILAB* promueve la creación de la Fundación para Agentes Físicos Inteligentes (FIPA, Fundation of Intelligent Physical Agents) [171] con el propósito de producir estándares para la tecnología de agentes. Basado en el primer grupo de especificaciones liberadas en 1997, a fines del 2002 FIPA consiguió publicar el estándar [170]. El principal objetivo del estándar es la interoperatividad, concentrándose en el comportamiento externo, dejando abiertos los detalles de implementación y estructura interna. El estándar FIPA adopta completamente el paradigma de agentes y, en particular, define un modelo de referencia de una plataforma de agente y un conjunto de servicios que debería proveer. La colección de estos servicios y sus interfaces representan las reglas normativas que permiten la coexistencia de múltiples y la gestión de los mismos (véase Fig 3.22)

Al ser los agentes entidades sociales, es necesario un lenguaje que permita la comunicación. El Lenguaje de Comunicación de Agentes (ACL, Agent Communication Language) [169, 263, 135] es uno de los principales aportes del estándar FIPA.

ACL está basado en la Teoría del Acto del Habla [22] y en el paradigma de agente descrito anteriormente, FIPA estandarizó una librería extendible de 22 actos comunicativos que permiten la representación de diferentes intenciones comunicativas, como ser: proponer, preguntar, informar, consultar, solicitar una propuesta, rechazar, etc. Los estándares FIPA también definen la estructura de un mensaje para representar y transmitir información que sirve para identificar emisor y receptor, el contexto y las propiedades del mensaje (codificación y lenguaje de representación) y, en especial, información adecuada para identificar y seguir los hilos de las conversaciones entre agentes. Uno de los principales aportes de los estándares FIPA es justamente su grado de estándar, definidos y aceptados por toda la comunidad de agentes.

3.7.3. JADE: un ejemplo

JADE [42, 43] es un sistema de middleware desarrollado por TILAB para el desarrollo de aplicaciones distribuidas multiagentes basadas en la arquitectura par-a-par de comunicación. Todas las características del sistema, es decir la iniciativa, la información, los recursos y el control, pueden ser completamente distribuidas tanto en terminales móviles como también en redes fijas. El ambiente puede evolucionar dinámicamente con pares, los que JADE denomina agentes, los cuales aparecen y desaparecen en el sistema de acuerdo a los requerimientos del entorno de la aplicación.

JADE está completamente desarrollado en Java y está basado en los siguientes principios:

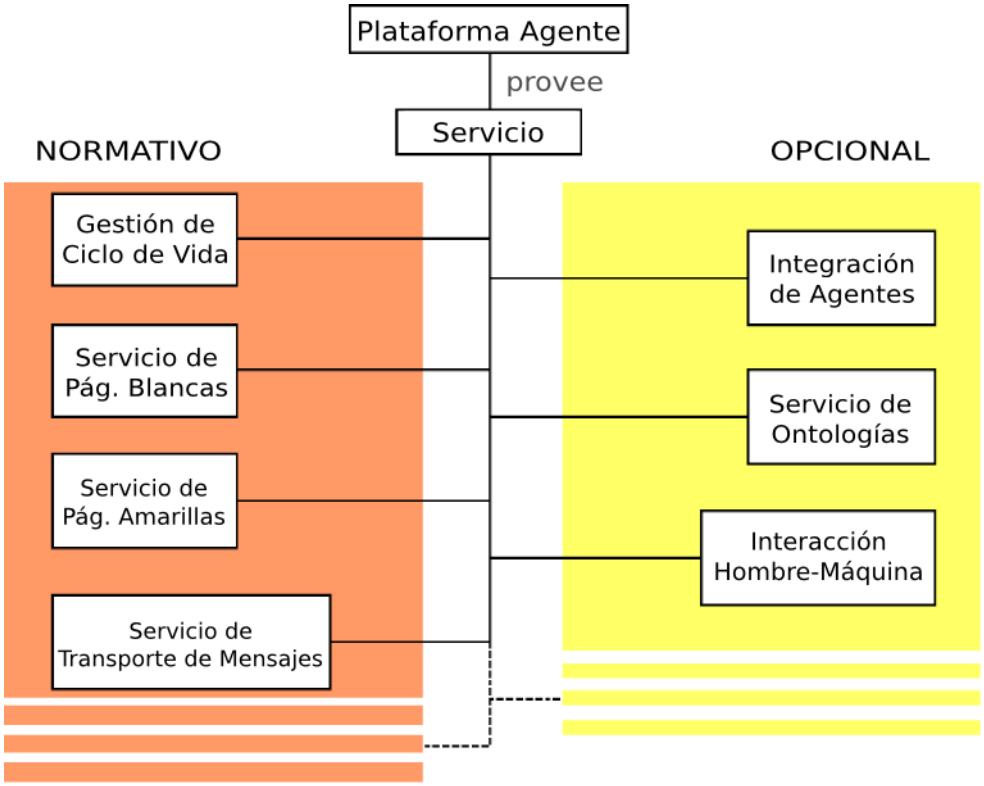

Figura 3.22: Servicios establecidos por el estándar FIPA.

1. **Interoperatividad**: Jade es compatible con las especificaciones FIPA. Como tal, los agentes JADE pueden interactuar con otros agentes que proporcionen compatibilidad con el estándar.

2. **Uniformidad y Portabilidad**: JADE provee un conjunto homogéneo de APIs que son independiente de la red y la versión de Java subyacente.

3. **Facilidad de Uso**: la complejidad de la plataforma de middleware está oculta detrás de un conjunto de APIs simples e intuitivas.

4. **Uso a demanda**: Los desarrolladores no necesitan usar todas las características provistas por la plataforma JADE. No se requiere que se conozcan todas las funcionalidades que no se utilizan, y además, estas funcionalidades no usadas no exigen ninguna sobrecarga computacional.

Modelo Arquitectónico

Es importante destacar que algunas de las características esenciales de JADE están estrechamente ligadas con la tecnología Java. Las tecnologías Java están agrupadas en 4 ediciones de acuerdo al harware en dónde se ejecutará y a las funcionalidades correspondientes. Asi existe J2EE [461] para servidores de aplicaciones y EAI, J2SE [466] para aplicaciones de escritorio, J2ME [462] para aplicaciones en ambientes móviles y Java Card [460] orientados a dispositivos de tarjetas inteligentes y con poca capacidad de memoria y procesamiento (véase Fig. 3.23). JADE está completamente implementado en Java, y puede ser ejecutado en cualquier Máquina Virtual Java, salvo Java Card. En teoría los desarrolladores pueden elegir el ambiente de ejecución de Java en tiempo de instalación del sistema.

JADE incluye las librerías para desarrollar aplicaciones de agentes, es decir las clases Java, y el ambiente de ejecución que provee los servicios básicos que deben estar activos en el dispositivo

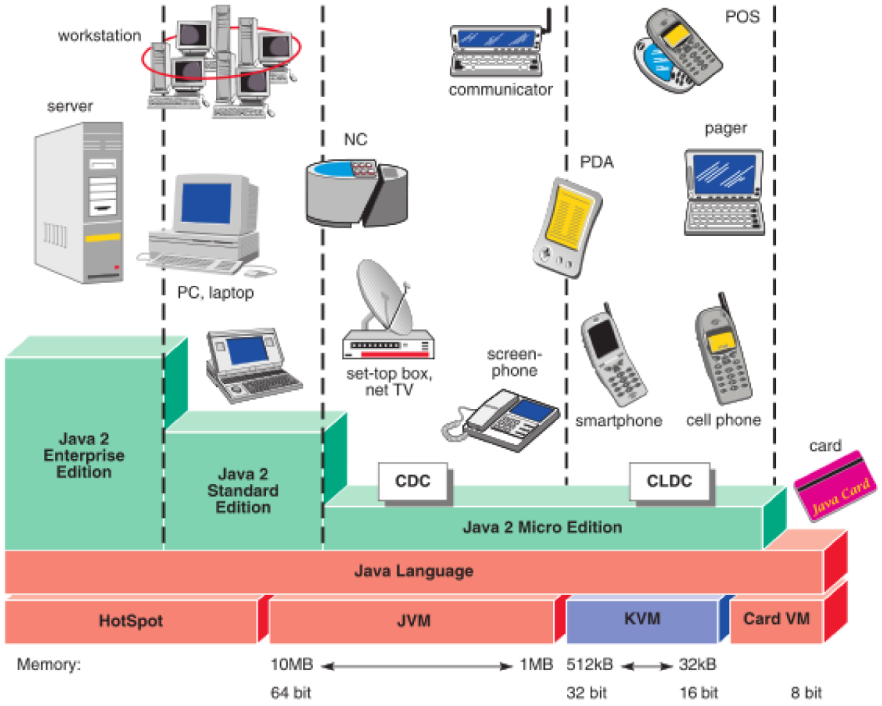

Figura 3.23: Tecnologías Java.

huésped para que el agente pueda ser ejecutado.

Cada instancia del run-time de JADE es denominado ***contenedor***. El conjunto de todos los contenedores es denominado ***plataforma*** y provee una capa homogénea que oculta la complejidad y la heterogeneidad de las capas subyacentes (hardware, sistema operativo, topología de red, JVM, etc). En este aspecto, JADE es compatible con el ambiente J2ME proveyendo APIs CLDC/MIDP1.0, el cual ha tenido éxito dándo soporte a redes GPRS en varios modelos de teléfonos celulares (véase Fig. 3.24).

JADE es extremadamente versátil, y no solamente se adapta a ambientes con limitados recursos como los teléfonos celulares de escasa memoria; sino que también puede ser integrado a arquitecturas complejas como .Net y J2EE en dónde JADE lleva a ser un servicio para ejecutar aplicaciones proactivas interempresariales.

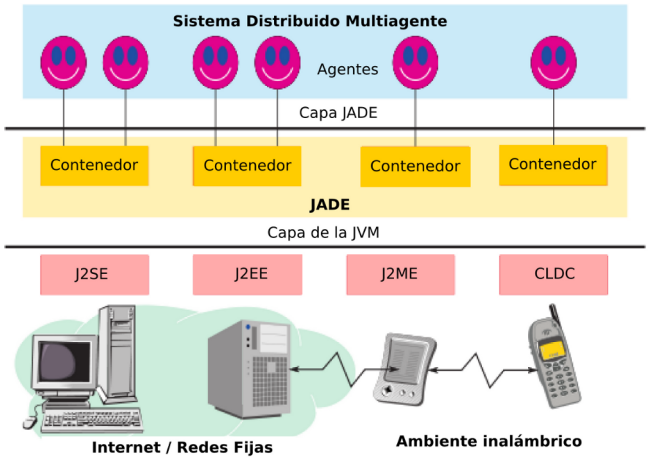

Figura 3.24: Arquitectura de un sistema distribuido basado en JADE.

Modelo Funcional

Desde el punto de vista funcional, JADE provee los servicios esenciales para soportar el paradigma de comunicación par-a-par para aplicaciones distribuidas, tanto en redes fijas como en ambientes móviles. JADE permite a cada agente descubrir dinámicamente a otros agentes y comunicarse con ellos de acuerdo al paradigma par-a-par.

Desde el punto de vista de una aplicación, cada agente es identificado por un único nombre y provee un conjunto de servicios. Es capaz de registrar y modificar sus servicios y buscar agentes que brinden determinadas prestaciones.

Cada agente puede controlar su ciclo de vida y, en particular, comunicarse con otros agentes. De manera similar a las plataformas MOM y a los Brokers de Mensajes (véase Sec 3.2.2 y Sec. 3.5.2 respectivamente), la comunicación entre agentes se realiza por el intercambio de mensajes asincrónicos, siendo éste el modelo universalmente aceptado para comunicaciones distribuidas bajamente acopladas. Este modelo asincrónico libera de dependencia temporal a los agentes intervinientes en una interacción. Debido a que un agente destinatario solamente se identifica por un nombre, cambios en la ubicación y referencia de su proceso en ejecución son transparente a las aplicaciones.

A pesar de esta forma de comunicación, la seguridad es preservada ya que JADE pone a disposición de las aplicaciones apropiados mecanismos de autenticación y verificación, asignando los correctos permisos a cada par, para prevenir acciones no permitidas de los agentes.

Todos los mensajes intercambiados son transportados con una *envoltura*. La envoltura sólo tiene información necesaria para transportación. Esto permite, entre otras cosas, encriptar el contenido del mensaje en forma separada de la envoltura.

La estructura de los mensajes es compatible con el ACL definido por FIPA [169]. Este sistema de mensajes permite soportar conversaciones complejas y múltiples conversaciones paralelas. Cada mensaje cuenta con variables indicadoras del contexto y de tiempos de contestación, lo que posibilita restablecer el hilo de conversación y su correspondiente ambiente. JADE provee también, un conjunto de plantillas de típicos patrones de interacción para lograr determinadas tareas, como ser negociación, delegación, compulsas, etc. Utilizando estas plantillas, los desarrolladores pueden abstraerse de los detalles de sincronización, tiempo de espera, condiciones de error, y todos aquellos aspectos que no están estrictamente relacionados con la lógica de aplicación.

Con el propósito de aumentar la escalabilidad y para satisfacer las restricciones de los ambientes con limitados recursos, JADE permite la ejecución de múltiples tareas dentro de un mismo hilo de ejecución de Java. En los ambientes J2SE y J2ME, JADE permite la mobilidad de código y estado de ejecución. Es decir que un agente puede parar su ejecución dentro de un contenedor, migrar su proceso a otro contenedor remoto, y reestablecer su ejecución en el punto de interrupción. Esta funcionalidad permite distribuir la carga computacional en tiempo de ejecución mediante la mudanza de agentes a una máquina menos cargada sin impacto colateral en el sistema.

La plataforma también incluye el *servicio de nombres* (páginas blancas) y las *páginas amarillas* que puede ser distribuido en múltiples hosts. Estas características de funcionalidad y, en particular, la posibilidad de activación remota, la ejecución en ambientes móviles, el sistema de mensajes relativos al contexto y la generación de nuevos pares en el sistema, hacen de JADE un marco para el desarrollo y la ejecución de aplicaciones distribuidas, interorganizativas, inteligentes y proactivas.

Ventajas y Desventajas

JADE, como toda plataforma de middleware facilita el desarrollo y la ejecución de sistemas distribuidos. En el caso puntual de JADE, y todo producto compatible con las especificaciones FIPA, sirve para producir aplicaciones distribuidas formadas por entidades autónomas que necesitan colaborar y comunicarse entre sí para lograr los objetivos globales del sistema informático.

JADE, como middleware, oculta toda la complejidad de la arquitectura distribuida que hace posible que esas entidades o agentes puedan descubrirse, conectarse e interactuar. De esta forma, el desarrollo se centra en el diseño de una lógica distribuida de aplicación.

El surgimiento de este tipo de aplicaciones distribuidas basadas en agentes, y en particular cuando se aplica a dispositivos móviles, inició una nueva tendencia en la evolución de sistemas distribuidos la cual se la denomina *dispositivos inteligentes* o *interconexión inteligente*; es decir que el software de cada dispositivo está equipado con autonomía, inteligencia y capacidad de colaboración. Con este enfoque, el valor del sistema distribuido está dado por las capacidades de cada dispositivo o nodo, por la interacción entre estos y la posibilidad de cooperación; en contrapunto con los sistemas tradicionales de acceso ubicuo en los cuales el valor está dado por el contenido y la capacidad de acceder a dicho contenido desde cualquier punto.

JADE adopta el estándar FIPA, y como tal soporta el paradigma de agente en sistemas par-a-par. JADE simplifica el desarrollo de aplicaciones que requieren negociación y coordinación entre un conjunto de agentes, en las cuales los recursos y la lógica de aplicación está distribuida en todo el ambiente. A tal efecto JADE provee de librerías de fácil uso para implementar la comunicación par-a-par.

Los agentes son proactivos, y como tal pueden ser desarrollados en JADE para iniciar una ejecución, que puede ser remota o no, sin intervención humana. Esta característica posibilita el desarrollo de sistemas distribuidos con patrones máquina-a-máquinas en sus interacciones.

Al ser las aplicaciones de JADE completamente par-a-par, facilita su adecuación a escenarios intraempresariales. Cada agente puede cumplir los roles de productor y consumidor de información sin la intervención de ningún sistema central que podría generar problemas políticos a la hora de seleccionar una entidad administradora, como sucede en escenarios de EAI. El hecho de que la inteligencia, la información y el control esté descentralizado, permite que la propiedad del sistema está distribuida entre los agentes o pares; y cada agente puede estar autorizado a ejecutar un determinado conjunto de operaciones en la aplicación.

Otro punto que facilita la integración de aplicaciones bajo el esquema distribuido de agentes es la interoperatividad que asegura el estándar aceptado de FIPA. JADE al adoptar este estándar es altamente interoperable. Como punto adicional esta la condición de ser un proyecto de software libre, lo cual garantiza calidad y trascendencia.

Como última característica distintiva de JADE se puede citar la versatilidad. JADE posibilita el desarrollo de aplicaciones que se corren en redes fijas o móviles. Al desarrollarse en Java, posibilita la ejecución en las diferentes plataformas de ejecución que se proveen: J2SE, J2EE y J2ME. Esto favorece el reuso en múltiples dispositivos y permite adaptar el sistemas a diferentes contextos según se lo requiera. JADE permite el desarrollo de aplicaciones distribuidas implementando una arquitectura par-a-par, en las cuales los nodos son agentes cuya necesidad de comunicación y colaboración logran la concreción del las funcionalidades exigidas en el sistema integral.

3.8. Resumen

Hay dos importantísimos aspectos que deben identificarse en los middleware. Por un lado, los middleware proveen abstracciones para programación para el diseño y la construcción de aplicaciones distribuidas. Estas abstracciones pueden encerrar modelos complejos de comunicación, transacciones, colas para interacciones asincrónicas, etc. Por otro lado, los middleware implementan las funcionalidades prometidas por las abstracciones de programación. Es decir, proveen la infraestructura por medio de la disponibilidad de primitivas de comunicación, servicio de directorios y nombre, mecanismos de persistencia y otros. El tipo de middleware, sus capacidades y sus posibilidades de uso depende fuertemente de estos dos aspectos.

Es importante tener una visión evolutiva del desarrollo de las plataformas de middleware. Cada paso en la evolución de middleware fue impulsado por requerimientos concretos generados

por problemáticas especificas en el área de distribución de sistemas.

Como génesis de este proceso están los sistemas basados en RPCs, los cuales ofrecieron una forma transparente de interacción entre componentes remotos, sin tener que tratar con los detalles de bajo nivel del canal de comunicación. Existió, luego de la aceptación de las primeras formas de RPC, un intento de estandarizar su implementación, mediante el DCE.

Los Monitores de Procesamiento de Transacciones tomaron los principios de RPCs para extender los ambientes de distribución a grandes sistemas, proveyendo mecanismos para realizar interacciones complejas transaccionales. Esto originó que se convirtieran en la forma de middleware predominante, jugando un rol determinante hasta la fecha.

Los Brokers de Objetos surgen para compatibilizar el desarrollo de sistemas distribuidos con el nuevo en el paradigma de programación emergente: la orientación a objetos. Es de recalcar que los brokers de objetos, y precisamente su ejemplo canónico CORBA, fue el segundo intento de estandarizar las plataformas de middleware.

Fue difícil concebir una plataforma de middleware que implemente completamente las especificaciones descritas por CORBA. Es por eso que existen pocos Brokers de Objetos implementados desde sus orígenes como tal. Los Monitores de Objetos surgieron para implementar las ideas de los Brokers de Objetos sobre plataformas robustas y aceptadas de Monitores TP.

Con el auge en el uso de clusters de computadores para sistemas distribuidos y los requerimientos de integración de aplicaciones empresariales, se vio adecuado la solución propuesta por los sistemas de colas presentes en los Monitores TP. Estos sistemas de colas llegaron a ser sistemas independientes evolucionando en las plataformas de Middleware Orientadas a Mensajes MOM. Con las plataformas MOM es posible lograr desacoplar a los componetes de un sistema distribuido, dejando la gestión de la interacción a los sistemas de colas.

Como una evolución de las plataformas MOM surgen los Broker de Mensajes, como principal tecnología para la EAI. En este contexto, los Brokers de Mensajes son el componente esencial de cada ambiente de EAI, ya que en estos reside la lógica de aplicación de las aplicaciones integradoras. Se logra más bajo nivel de acoplamiento de componentes por implementar a las aplicaciones de integración como sistemas basados en eventos, los cuales tienen como mecanismo básico de comunicación al esquema publicar/suscribir.

Como último ejemplo de las plataformas de middleware, es necesario citar a las plataformas basadas en Agentes como solución a los planteos propuestos por la computación ubicua. Estas plataformas sustentan la complejidad de interconectar a diferentes dispositivos de cómputo residentes en redes inalámbricos y fijas, brindando la adecuada transparencia de distribución. Por esencia, la computación ubicua exige la cooperación entre dispositivos, promoviendo el uso del paradigma de agente como marco de diseño para las aplicaciones distribuidas.

Capítulo 4

Tecnologías de la Web

Por lo tratado en los capítulos anteriores, se ha podido demostrar que la evolución de las plataformas de middleware fue promovida por la necesidad de integrar sistemas informáticos, con un grado cada vez más alto de heterogeneidad. La necesidad de interoperatividad no se restringe a los sistemas dentro de una misma empresa; sino que también se exige la vinculación de sistemas residentes en dominios empresariales diferentes.

La principal razón para requerir integración interempresarial es lograr mejoras operativas y de eficiencia en los procesos organizativos o flujo de trabajos que traspasan los límites de confianza organizacionales. En ese contexto, existen varios inconvenientes técnicos y no-técnicos a solucionar para lograr el escenario deseado. Desde el punto de vista técnico, el hecho de vincular sistemas informáticos de varias organizaciones en una aplicación interempresarial tienen implicaciones importantes en la elección de políticas de integración, esencialmente en:

- los mecanismos de seguridad y privacidad que cada empresa dará a cada módulo o a cada aplicación que dispone integrar;
- el tipo y protocolo de comunicación que se usarán para comunicar a los componentes o aplicaciones de distintas empresas;
- la semántica de datos y de documentos que se intercambiarán la aplicación interempresarial.

Por otro lado, en cuestiones no-técnica, surge como principales inhibidores de la integración interempresarial las políticas que tienen que ver con el resguardo y discreción de la información que maneja cada empresa:

- falta de confianza que existen entre las empresas respecto a qué debe compartirse y qué no, con sus contrapartes asociadas;
- de existir una plataforma integradora centralizada, resulta dificultoso determinar quién será la organización administradora de tal plataforma;
- desacuerdos entre equipos técnicos respeto a qué tecnología usar para sobrellevar la heterogeneidad de plataformas y aplicaciones.

Este capítulo sirve como nexo entre los primeros, que presentan plataformas de middleware centradas esencialmente en la integración intraempresarial, y los subsiguientes, que presentan a los Servicios Web como principal tecnología de la integración interempresarial. En este capítulo se presenta a la Web es el medio para compartir e intercambiar información en la Internet. Es natural pensar en la Web como un marco propicio para la integración interempresarial, ya que es un vínculo conocido y de relativa confianza para comunicar a clientes y servicios remotos.

En este capítulo se explican las tecnologías que soportan el corazón de la Web; luego se enunciarán aquellas que sirven para crear clientes remotos; y por último se hace una valoración de los servidores de aplicaciones como plataforma de middleware para integrar aplicaciones disponibles a través de la web.

4.1. Intercambio de información a través de Internet

La ***Internet*** es el sistema global de redes de computadoras. Sus orígenes se remontan a 1969, en la Agencia de Proyectos de Investigaciones Avanzadas (ARPA) interconectó al Instituto de Investigaciones de Stanford, a la Universidad de Los Ángeles California (UCLA), a la Universidad de Santa Bárbara California y a la Universidad de Utah en una red llamada ARPANET [3]. Esta red original ARPANET permitió la interconexión de sistema informáticos autónomos; tal proyecto permitió el surgimiento de las primeras organizaciones de estandarización que gobernaron dicha red.

Estos grupos desarrollaron estándares como el Protocolo de Control de Transmisión (TCP, Transmission Control Protocol) el cual maneja la conversión entre mensajes y flujos de paquetes, y el Protocolo de Internet (IP, Internet Protocol) el cual gestiona el direccionamiento de paquetes a través de la red. En su conjunto, TCP/IP es la tecnología definitoria de Internet, ya que permite el envío de paquetes a traves de múltiples redes usando múltiples normas estándares (Ethernet, FDDI, X.25, etc).

4.1.1. Tecnologías previas a la Web

Dos de los estándares más antiguos para el intercambio de información a través de Internet son: el protocolo ***telnet*** [389, 392] y el protocolo para correo electrónico ***SMTP*** *(Simple Mail Transfer Protocol)* [536, 253]. Ambos protocolos especifican diferentes formas para conectar diferentes cuentas en diferentes sistema, sin importar el sistema operativo subyacente y la distintas arquitecturas de computadoras intervinientes.

El protocolo SMTP evolucionó más tarde con el agregado de extensiones de múltiple propósito o extensiones ***MIME*** *(Multi-purpuse Internet Mail Extensions)* [176, 177, 316, 178, 179, 175], las cuales permitieron intercambiar archivos de datos más ricos, como audio, video e imágenes. Luego, telnet y el correo electrónico se fusionaron en el ***Protocolo de Transferencia de Archivos FTP*** *(File Transfer Protocol)* que fue publicado en 1973 [391]. El protocolo FTP permite la transferencia de archivos entre dos sitios de Internet, y le permite a los sistemas publicar sus archivos en un servidor FTP. Una de las innovaciones de FTP, es que, a pesar de que el modo normal del protocolo requiere autenticación, es posible permitir que usuarios anónimos puedan operar con los servidores. Esto significa que los usuarios no necesitan tener una cuenta en cada sistema a los que se conectan.

La existencia de FTP permitió el desarrollo de dos tecnologías que pueden verse como los primitivos sistemas informáticos distribuidos similares a la Web: ***Archie*** [144], de fines de la década del 1980, el cual puede verse como la tecnología original que usa FTP para implementar un sistema de archivos distribuidos; y ***Gopher*** [18] un protocolo de aplicación que establece un sistema cliente/servidor y una interfaz de usuario gráfica para publicar y acceder a publicaciones/archivos de textos a través de Internet [282].

4.1.2. La Web

El corazón de las tecnologías Web que se conocen actualmente, es decir el protocolo *HTTP*, el lenguaje *HTML*, los servidores web y los navegadores web, son una evolución de estas tecnologías tempranas.

En una breve revisión histórica del origen de la Web, es importante destacar la participación de Tim Berners-Lee quien a los principios de los 1980s, trabajando para el Consejo Europeo para la Investigación Nuclear (CERN, Conseil Européen pour la Recherche Nucléaire), propuso un proyecto basado en el concepto de hipertexto, para facilitar la publicación, intercambio y actualización de información entre investigadores. A tal efecto, construyó una aplicación denominada *Enquire*, que la utilizaba como base de datos personal de personas y modelos de software [53].

Para 1989, el CERN poseía el nodo más importante de Internet en Europa, y fue entonces que Tim Berners-Lee vio la posibilidad de integrar la idea de hipertexto y con la de Internet. Motivado por esto, publicó una propuesta [54] en donde presentaba la idea de una gran base de datos de hipertexto con vínculos que permitían la interconección a través de TCP. Esta propuesta fue revisada con la ayuda de Robert Cailliau en 1990 [58, 59]. A partir de allí, Berners-Lee utilizó las ideas subyacentes del proyecto Enquire para crear la World Wide Web, para lo cual construyó las herramientas para la primer Web operativa: el primer navegador web llamado WorldWideWeb desarrollado en NextSTEP [55], el primer servidor web llamado ***httpd*** *(HyperText Tranfer Protocol deamon)* [57] y las primeras páginas [56] que describen el proyecto en sí.

Hablando de las tecnologías desarrolladas para la Web, se puede decir que el ***Protocolo de Transferencia de Hipertexto HTTP*** *(HiperText Tranfer Protocol)* [166, 167], es una protocolo genérico y sin estado que gobierna la transferencia de archivos en una red. El sentido de genérico viene dado por ser un protocolo que puede acceder a otros protocolos como FTP y SMTP. HTTP fue diseñado principalmente para soportar hipertexto; en particular el ***Lenguaje de Marcado para Hipertexto HTML*** *(HiperText Markup Language)* [224], el cual define un conjunto estándar de identificadores o marcas que indican cómo debe verse una página en un navegador Web.

URIs

Cada documento que se intercambia según el protocolo HTTP, es identificado mediante un ***Identificador Uniforme de Recurso URI*** *(Uniform Resource Identifier)*. Los URIs están caracterizados por [49]:

- **Uniformidad**: Permite que diferentes tipos de identificadores de recursos sean usados en el mismo contexto, aún si los mecanismos para acceder a esos recursos difieren. Permite una semánica uniforme para la interpretación de convenciones sintácticas aplicables a los diferentes tipos de recursos. Propicia la extensibilidad, al contar con medios para crear nuevos tipos de identificadores, sin afectar a los existentes. Promueve el reuso de identificadores en diferentes contextos.

- **Recurso**: No existe límite para definir qué es un recurso. En sentido general cada "recurso" es aquello que puede ser identificado por un URI. Los ejemplos más familiares pueden ser un documento electrónico, una imagen, una fuente de información, un servicio o una colección de otros recursos. Un recurso puede no estar disponible vía Internet, por ejemplo una persona, una empresa, un libro en una biblioteca, etc. De la misma forma los conceptos abstractos pueden ser recursos, como ser operaciones y operandos de una expresión matemática, tipos de relaciones (por ejemplo "empleado de", "es un") o valores numéricos.

- **Identificador**: un identificador encierra la información requerida para distinguir un determinado recursos de todos los demás recursos identificados. No se debe asumir que siempre un identificador define o encierra la identidad que lo que se referencia. Tampoco debe asumirse que un sistema que usa URIs accederá a esos recursos; en muchos casos los URIs son usados para denotar recursos, sin ninguna intención de accederlos. De la misma forma, "el" recurso identificado puede no ser singular en su naturaleza, como por ejemplo un conjunto o mapeo que cambia con el tiempo.

Un ***URI*** es un identificador que consiste en una secuencia de caracteres que sigue una sintaxis determinada. Permite la identificación uniforme de los recursos por medio de un conjunto extensible de nombres de esquemas. La forma en que un identificador es especificado, asignado y dispuesto es determinada por la especificación del esquema en particular.

Los URIs tienen un alcance global y son interpretados de manera consistente independientemente del contexto, a pesar que el resultado de la interpretación puede estar en relación con el

contexto del usuario. Por ejemplo `http://localhost/` tiene la misma interpretación para cada usuario, sin embargo la máquina que se corresponde al nombre `localhost` puede ser diferente para cada usuario. Esto muestra que los URIs permiten la independencia entre interpretación y acceso. Las posibles acciones que se pueden efectuar en base a éste URI (`http://localhost/`) se efectuarán en relación al contexto del usuario.

Un URI puede ser clasificado como un localizador, o un nombre, o ambos. El término ***Localizador Uniforme de Recurso URL*** *(Uniform Resource Locator)* [51, 49] se refiere a un conjunto de URIs que, además de identificar un nombre, también provee medios para localizar el recurso describiendo su mecanismo de acceso, por ejemplo su dirección de red.

El término ***Nombre Único de Recurso URN*** *(Uniform Resource Name)* [313] ha sido usado históricamente para referencia a los URIs definidos bajo el esquema "urn" (el cual es usado para mantener unicidad y persistencia aún cuando el recurso deja de existir o no está más disponible), y para cualquier otro URI con propiedades de nombre.

Funcionamiento de HTTP

HTTP es un protocolo de esquema *petición/respuesta*. El mecanismo de este protocolo se basa en el modelo cliente/servidor, típicamente usando TCP-IP sockets. Los sockets son una abstracción que se basa en la generalización del sistema de acceso de archivos UNIX que provee un extremo a una comunicación. Los *sockets TCP-IP* [104] son un identificador de proceso global constituido por una dirección IP y un número de puerto. El puerto por defecto que se utiliza para establecer el socket destinado a HTTP es el 80, sin perjuicio de usar otros.

No existe impedimento para que existan implementaciones de HTTP sobre otros protocolos de Internet (distintos a TCP/IP) u otras redes. HTTP sólo exige transporte fiable, y cualquier protocolo que cumpla con tales garantías es aceptable.

En una interacción HTTP, un *cliente HTTP* (por ej.: un navegador web), que en la especificación de HTTP se denomina ***agente de usuario***, abre una ***conexión*** con un *servidor HTTP* (por ej.: un servidor web) al que se lo denomina ***servidor original***, y envía una mensaje de petición de un recurso o entidad que tiene este servidor.

Dicho mensaje contiene:

- el método de petición
- un URIs del recurso que solicita
- la versión de protocolo HTTP que usa
- y un mensaje basado en MIME, el cual puede contener parámetros de la petición, información del cliente y, posiblemente, contenido relacionado con la conexión.

El servidor contesta con una línea de estado que incluye la versión del protocolo usada y un código de éxito o error, seguido de un mensaje basado en MIME conteniendo la información del servidor, y la ***entidad***[1] (recurso o documento) solicitada.

La mayoría de las comunicaciones HTTP son iniciadas por el *agente del usuario* y consisten en una petición para ser aplicada a algún recurso de un *servidor original*. Un ***servidor original*** es aquel que responderá efectivamente a la petición. Este caso puede ser logrado por una simple conexión desde el agente del usuario (AU) hacia el servidor original (SO) (véase Fig. 4.1).

Una situación más compleja ocurre cuando uno o mas intermediarios están presentes en la cadena de petición o respuesta. Hay tres formas comunes de intermediarios: *proxy*, *gateway* y *túnel* [167]. Un ***proxy*** es un programa para reenvío de mensajes, el cual recibe peticiones de un URL en su forma original, reescribe la petición en parte o completamente y reenvía la petición

[1] *Entidad*: es la información útil o documento propiamente solicitado. Es la carga útil de una respuesta o petición, sin tener en cuenta los datos relacionados con el protocolo o el mensaje. Una *entidad* consiste en *metainformación* (en la forma de campos de cabecera) y el *contenido* (en la forma de cuerpo).

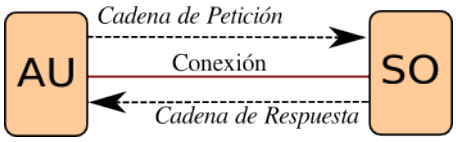

Figura 4.1: Esquema de comunicación básico de HTTP.

Figura 4.2: Interacción con intermediarios en HTTP.

reescrita; se puede decir que un proxy actúa como servidor y cliente con el propósito de hacer peticiones por cuenta y orden de otros clientes. Un ***gateway*** es un programa que recibe mensajes, que actúa como capa por encima de otros servidores y, si es necesario, retransmite la petición en el protocolo que maneja el servidor. Un ***túnel*** *(tunnel)* actúa como punto de retransmisión entre dos conexiones sin cambiar los mensajes.

Se puede decir que un proxy actua en nombre del agente del usuario, que un gateway actúa en el nombre de un servidor de origen y que un túnel simplemente conectan dos redes, haciendo puente sobre los mecanismos de seguridad que pueden aislar a una red. Estos mecanismos de seguridad usualmente se denominan ***cortafuegos*** *(firewalls)* (véase Fig. 4.2).

Los proxies y los gateways procesan las URLs y el contenido de los mensajes HTTP; los túneles no. Los proxies, gateways y túneles son los conceptos esenciales de los que se vale las aplicaciones HTTP para afrontar los retos de integración sobre el ambiente de la Web [10].

La Fig. 4.3 muestra tres intermediarios (A, B y C) entre el agente del usuario y el servidor original. Una petición o una respuesta, que viaje por toda la cadena, pasará por cuatro conexiones separadas e independientes. Este es un concepto importante de aclarar, ya que las opciones de comunicación HTTP se pueden aplicar a las conexiones entre nodos vecinos cercanos, a toda la cadena de nodos (excepto los túneles) o solamente a los nodos extremos de la cadena. A pesar de que el diagrama es lineal, debe entenderse que cada nodo de la cadena, está participando concurrentemente de otras cadenas de comunicación simultáneas.

Cualquier parte de la comunicación (excepto los túneles) puede utilizar *caches internos* para atender a las peticiones. El efecto del cache es que la cadena de petición/respuesta es acortada si uno de los participantes intermedios tiene almacenado en su cache una respuesta que se adecúe a la petición. La Fig. 4.4 ilustra el resultado de la cadena si el nodo intermedio B tiene una copia guardada en cache de una respuesta anterior del servidor SO (vía C) que no fue almacenada por AU ni A.

No todas las respuestas pueden ser guardadas en cache, y algunas respuestas pueden contener modificadores que requieran especial comportamiento del cache. De hecho, hay una amplia variedad de arquitecturas y configuraciones de caches y proxies a través de la Web. Esos esquemas pueden incluir jerarquías nacionales de cache de proxies para disminuir el tráfico transoceánico,

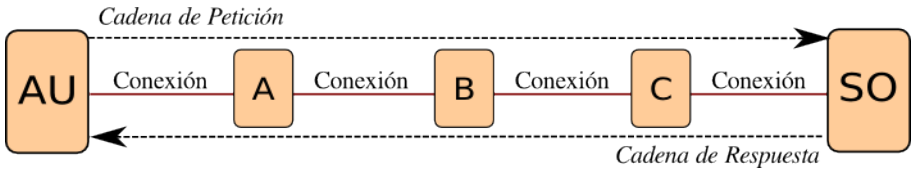

Figura 4.3: Cadena de comunicación de una interacción HTTP.

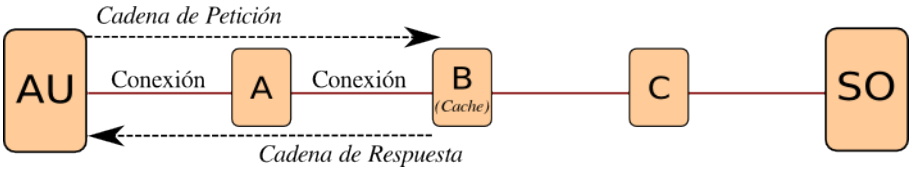

Figura 4.4: Efecto del uso de caches internos en interacciones HTTP.

sistemas que replican caches, organizaciones que distribuyen un subconjunto de datos cacheados vía CD-ROM, etc. Unos de los principales logros de HTTP/1.1 es el soporte que brinda a una amplia variedad de configuraciones, además de proporcionar constructores para adaptarse a las necesidades de quienes construyan aplicaciones web que requieran alta disponibilidad.

En las implementaciones de la versión HTTP/1.0, se usaba una nueva conexión por cada cadena de petición/respuesta que se intercambiaba; es decir que para recuperar una página web se requería una conexión por cada elemento individual de la página (imágenes, sonidos y la propia página). En la versión HTTP/1.1, una conexión puede ser utilizada por uno o más intercambios petición/respuesta ya que se exige que los servidores soporten *conexiones persistentes*, con el propósito de evitar la carga de abrir y cerrar conexiones.

En cada mensaje de HTTP, se debe especificar un *método de petición*. El ***método de petición*** indica la operación a ejecutar del lado del servidor. Algunos métodos típicos son:

- **OPTION**: envía información acerca de las opciones de comunicación soportadas por un servidor particular.
- **GET**: recupera cualquier entidad especificada en la petición, y si la petición especifica un programa, recupera el documento generado por ese programa
- **POST**: adjunta información para incluirla en la petición, destinada al recurso identificado en la petición.
- **PUT**: almacena información incluida en la petición en una ubicación especificada como parte de la petición
- **DELETE**: elimina el recurso indicado en la petición

4.1.3. Limitaciones del HTTP

Si bien HTTP tiene importantes ventajas, también tiene limitaciones. La mayoría de estas limitaciones son sobrellevadas con las correspondientes extensiones del protocolo.

En primer lugar, los datos en el HTTP plano no son encriptados antes de enviarse; esto hace que los mensajes sea vulnerables a ataques de espías de tráfico (*sniffers*) mientras viajan en la red entre el cliente y el servidor. A tal efecto; Netscape desarrollo un protocolo denominado ***SSL*** *(Secure Sockets Layer)* [528, 172, 218], que usa encriptación de claves pública/privada para proteger los datos transferidos sobre TCP/IP. El protocolo HTTP implementado sobre SSL se denomina ***HTTPS*** [380], y les permite a cliente y servidor utilizar SSL para autenticarse respectivamente y establecer una comunicación encriptada entre los mismos. HTTPS se ejecuta por encima de SSL, y requiere que el servidor y el cliente tengan disponible SSL. Una alternativa que surgió posteriormente al SSL es ***TSL*** *(Transport Layer Security)* [133, 134], que provee mecanismos más sofisticados y generales. De igual forma que HTTPS, existe una implementación de HTTP sobre TSL [409].

Otra limitación importante de HTTP es la cualidad de ser un protocolo que no preserva el *estado de la conexión*; por ende, no guarda información de las posibles transacciones que se establecen entre cliente y servidor. Una *transacción* o *sesión*, en el contexto de la Web, es una secuencia de varias peticiones/respuestas. Sin embargo en HTTP básico, cada una de las

peticiones es independiente y no preserva relación con otras. Los desarrolladores deben entonces, tomar la responsabilidad de mantener y gestionar la información de una sesión y su estado. Una forma por la cual un servidor HTTP puede almacenar información en un cliente es a través de la utilización *cookies* de HTTP. Una ***cookie*** [260, 325, 214] es una pequeña estructura de datos que el cliente mantiene a petición del servidor. Las cookies son utilizadas para guardar información del estado de una sesión por medio de la creación de una traza, en la máquina del cliente, de cualquier información necesaria que se pretende sea propagada en varias invocaciones de una sesión. En cada invocación, la información contenida en la cookie se envía al servidor, de forma tal que éste pueda recuperar la información relacionada con el "diálogo" de la sesión establecida con el cliente.

4.2. Tecnologías de la Web para soportar clientes remotos

Los objetivos originales de la Web y de las tecnologías explicadas previamente se focalizaban en vincular y compartir documentos a través de Internet. Sin embargo, rápidamente se vió el potencial que tenía la posibilidad de enmascarar los sistemas informáticos propios y exponer su Nivel de Presentación en la forma de documentos HTML para lograr tener clientes distribuidos por toda Internet, usando conexiones basadas en HTTP y navegadores Web.

4.2.1. La necesidad de soportar clientes remotos

Como se detalló en el Capítulo 3, las plataformas de middleware convencionales están diseñadas para operar dentro de una única organización. Esto significa, por ejemplo, que las aplicaciones clientes de un sistema 3-capas serán operadas por empleados de la empresa, y el intercambio de datos se hará dentro de los límites seguros de la organización. Sin embargo, no existe un impedimento para que un sistema sea abierto a otros usuarios, como por ejemplo proveedores y clientes comerciales.

El mejor ejemplo de esta situación es el uso de *cajeros automáticos* o *ATMs (automatic teller machine)* que posee una entidad bancaria [145]. Básicamente un cajero automático es una computadora conectada en red con los sistemas de información del banco [60]. Los cajeros automáticos son sistemas cliente/servidor y aportan varias ventajas significativas. Primero, un cajero automático brinda a los usuarios un mejor acceso a sus cuentas bancarias; esto sirve para que los bancos ahorren costos de operación, mantenimiento, atención en ventanillas y la instalación de nuevas sucursales. Segundo, una parte significativa del trabajo manual lo hace el usuario, generando él mismo la transacción, sin necesidad de un empleado. Estas características resultan en significativos ahorros en la práctica para el banco, y en una mejor y más eficiente interacción con sus clientes.

Si bien los cajeros automáticos proveen ventajas importantes, existen limitaciones relacionadas con la potencial cantidad de cajeros automáticos que se pueden tener en manera eficiente. Por más que los bancos se esfuercen, siempre los usuarios deberán trasladarse al cajero automático más cercano para hacer sus transacciones. Esto no sería necesario si existiera un cajero automático "personal" en cada uno de los hogares de los usuarios-clientes del banco. En este escenario un usuario no tiene necesidad de compartir su cajero automático, y de esa forma determinar los momentos y tiempos de realización de las operaciones financieras, ya que él es dueño de su propio cajero.

La arquitectura resultante está descripta en la Fig. 4.5. Tales interacciones se denominan *Business-to-Consumer (B2C)* [376, 97, 205], indicando que las empresas permiten a sus usuarios-clientes acceder a los servicios informáticos directamente.

Una de las más grandes contribuciones de la Web ha sido, precisamente, proveer el marco de desarrollo para estos escenarios. En vez de desarrollar un software-cliente para cada usuario que se instalaría en su computadora y personalizándolo en cada situación, la web provee un cliente

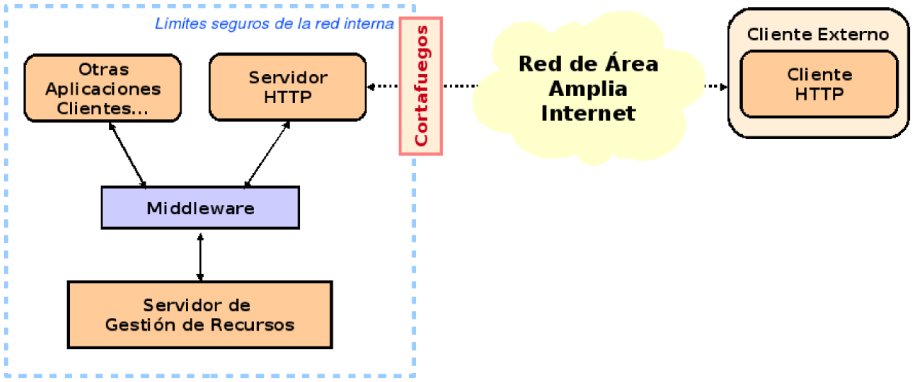

Figura 4.5: Efecto del uso de caches internos en interacciones HTTP.

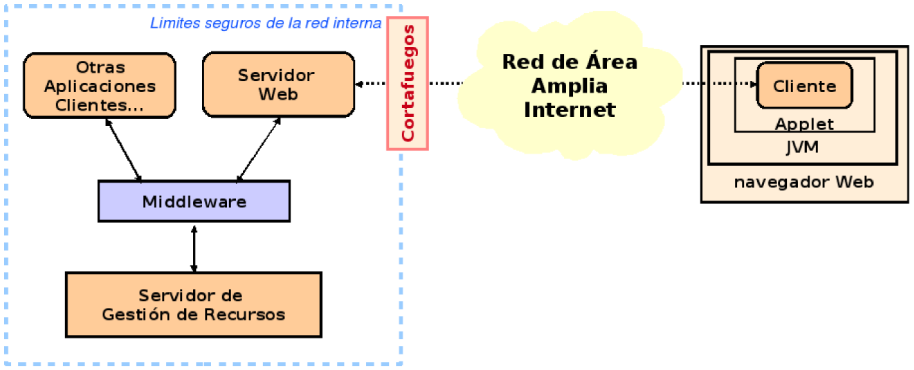

Figura 4.6: Applet como forma de implemetar clientes remotos.

universal para tales situaciones: el navegador web.

En la actualidad, las arquitecturas descriptas en la Fig. 4.5 son implementadas, no moviendo el cliente a la máquina del usuario, sino permitiendo que la máquina del usuario utilice un navegador web como cliente. Debido a que los navegadores web son herramientas estándares, no se requiere el desarrollo de clientes específicos. Las empresas pueden tomar ventaja de esta aplicación universal y desarrollar canales de acceso a sus aplicaciones, basados en HTTP. Esto se logra en la práctica, con la utilización de herramientas Web que exponen los sistemas informáticos internos de la empresa para soportar este nuevo canal de acceso.

4.2.2. Applets

Uno de los primeros problemas que surgen al usar las tecnologías Web para implementar clientes remotos es la limitada funcionalidad que aportan los navegadores web, los cuales fueron diseñado originariamente para mostrar documentos estáticos HTML que retornaba como respuesta a una petición HTTP. Esto genera dificultad a la hora de diseñar clientes sofisticados para los navegadores.

Una de las soluciones a esta necesidad son los ***Applets*** [475, 522], que son aplicaciones escritas en Java que pueden ser incrustados en un documento HTML. Cuando el documento es recibido en el navegador web cliente, el programa se ejecuta en la Máquina Virtual Java (JVM) presente en el navegador. Así, una forma de convertir el navegador en una aplicación cliente es mandar el código de la aplicación como un applet. Si bien, el código debe bajarse cada vez que el cliente es usado, es una solución común para clientes livianos. De esta forma, el escenario planteado en la Fig. 4.5 podría mostrar la implementación del cliente a través de applets ejecutándose en el navegador web, como lo muestra la Fig. 4.6.

Los applets proponen importantes ventajas al convertir un navegador web en una aplica-

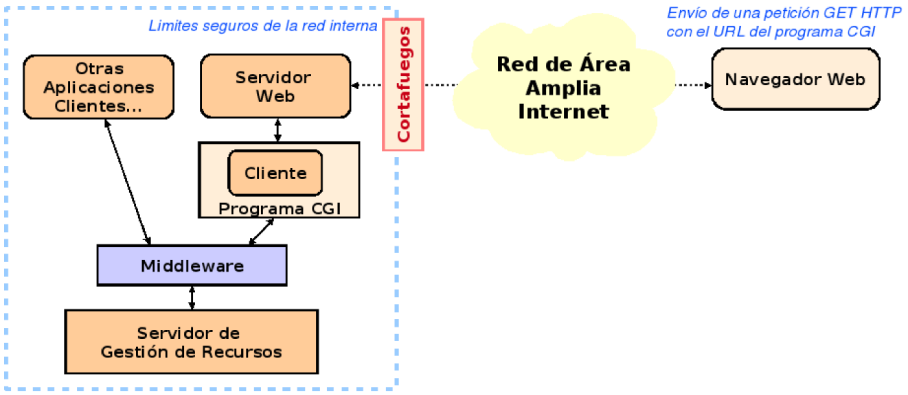

Figura 4.7: Los programas CGI son una forma de acceder a una aplicación del lado del servidor, por medio de un URL.

ción cliente específica sin mayores complicaciones de configuración e instalación. Como principal desventaja, se puede decir que los applets tienen el tiempo de vida de una instancia particular del browser; por consiguiente no son adecuados para soportar clientes complejos o con interacciones frecuentes. Una alternativa sería desarrollar un cliente, no basado en el navegador web, con el código necesario para interactuar con el servidor web a través de HTTP. Si bien esto soluciona el problema de trascendencia de los applets, requiere un cliente especializado. En este sentido, las ventajas y desventajas son idénticas a las expuestas en el Capítulo 2 sobre sistemas clientes-servidor (véase Sec. 2.2.2).

4.2.3. CGIs

En los casos planteados hasta aquí, se supuso que el servidor web retorna solamente documentos estáticos, posiblemente conteniendo un applet el cual se ejecuta en el navegador web cliente. Otro enfoque sería considerar al servidor web como interfaz a los sistemas informáticos locales. En este caso, los servidores deben ser capaces de retornar contenido dinámico; por ejemplo, información recuperada de una base de datos de acuerdo a una consulta parametrizada. Para lograr esto, el servidor web debe tener o estar vinculado a un mecanismo que permita invocar a una determinada aplicación la cual automáticamente generará el documento a ser retornado.

Uno de las primeras tecnologías que surgió para abordar esta necesidad fue ***CGI*** *(Common Gateway Interface)* [410, 284], que es un mecanismo estándar que permite que los servidores HTTP sean interfaces de aplicaciones externas a estos. La traducción literal del nombre es *Interfaz de Entrada Común*, lo cual tiene sentido porque los servidores funcionan como entrada o *puerta* (*gateway*) hacia los sistemas locales. CGI asigna URLs a los programas, de forma tal que cuando un URL es invocado desde el navegador web cliente, un programa es ejecutado en la máquina del servidor.

Los programas CGI están escritos en una variedad de lenguajes de programación, y deben ubicarse en un determinado directorio en la máquina dónde reside el servidor web. De esta forma el servidor web identifica si debe contestar con un documento estático o ejecutar un programa. Cuando un URL relacionado con un programa CGI es recibido, el servidor web extrae la información que se requiere pasar a la aplicación como parámetros y ejecuta dicha aplicación en un proceso separado. Estos programas, los invocados por CGI, son los que en definitiva interactúan con el middleware subyacente (véase Fig. 4.7).

4.2.4. Servlets

Desde el punto de vista de la performance, los programas CGI insumen cierta cantidad de carga de procesamiento. En primer lugar, un proceso separado es creado por cada invocación

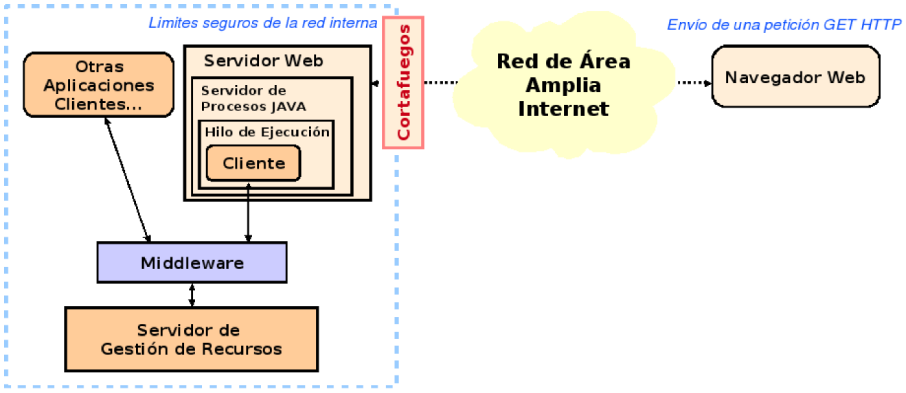

Figura 4.8: Los Servlets son una forma de acceder a una aplicación del lado del servidor, por medio de un URL.

al CGI[2]. La creación de los procesos insume tiempo y recursos computacionales; además, se requiere la intervención del sistema operativo que debe pasar el control al programa CGI. Esto genera una disminución en la performance general del sistema. Más aún, si por cada petición se crea un proceso, los procesos creados por varias peticiones similares competirán por los mismos recursos, es decir, por las mismas conexiones a bases de datos, por memoria, etc. Por esta causa, se considera que la tecnología de CGI tiene limitación en escalabilidad de los sistemas.

Esta limitación de los CGIs, propició la creación de los ***Servlets*** de Java [457, 227]. La idea de los servlets es exactamente la misma que los CGI, pero la implementación es diferente (véase Fig. 4.8). La ejecución de los servlets se disparan a través de URLs al igual que los programas CGIs; y el resultado es el mismo, se retorna un documento.

Sin embargo existen diferencias entre los servlets y CGIs. Estas diferencias afrontan el problema de performance y escalabilidad presentes en los últimos. Las principales diferencias son [547]:

- Un servlet no se ejecuta en un proceso separado. Esto remueve la sobrecarga de la creación de un nuevo proceso por cada petición.
- Un servlet permanece en memoria entre peticiones. Un programa CGI, y posiblemente su runtime o intérprete, necesitan ser cargados e inicializados por cada petición.
- Existe una única instancia del servlet la cual atiende a todas las respuestas concurrentes. Esto ahorra memoria y le permite a los servlets gestionar la persistencia de datos.
- Un servlet es ejecutado en un *motor* o ***contenedor de servlets*** en una zona segura restringida (*sandbox*), que previene al servlet de ejecutar acciones no deseables al resto del sistema.

La gestión de sesiones, el compartir conexiones a bases de datos y otros casos típicos de optimización, son aplicables a los servlets, lo que favorece la escalabilidad en contraposición a la tecnología CGI.

4.3. Servidores de Aplicación

Al plantearse la necesidad de usar a la Web como canal de acceso a los sistemas de información, las plataformas de middleware fueron forzadas a proveer tal soporte. Este soporte es generalmente provisto en la forma de *Servidores de Aplicación*.

[2]Una alternativa al CGI estándar, es ***FastCGI*** [213], el cual tiene ventajas sobre la performance ya que reduce la carga asociada a la creación de procesos. FastCGI mantiene en ejecución a un único proceso que realiza las invocaciones, en vez de generar un nuevo proceso por cada invocación.

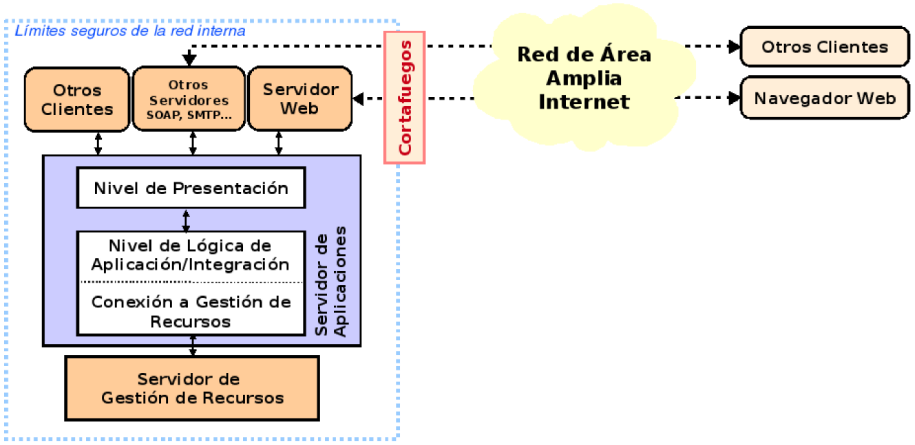

Figura 4.9: Los Servidores de Aplicación abarcan funcionalidades del Nivel de Presentación y de Lógica de Aplicación/Integración.

Un ***Servidor de Aplicación*** *(Application Servers)* es equivalente a las plataformas de middleware presentadas en el Capítulo 3. La diferencia fundamental es la incorporación de la Web como principal canal de acceso a los servicios implementados a través del middleware.

Esta adopción de la Web como canal de acceso tiene significativas implicaciones. Por ejemplo, el Nivel de Presentación adquiere mayor relevancia que en los middlewares tradicionales; como consecuencia directa de la forma de trabajo del HTTP y de la Web, todos los intercambios de información se realizan a través de documentos. Una de las mayores partes de infraestructura que contiene un Servidor de Aplicación está relacionada a las funcionalidades de preparación, generación dinámica y gestión de estos documentos.

En un Servidor de Aplicaciones, estas funcionalidades relacionadas con el Nivel de Presentación en la Web típicamente son fusionadas con el Nivel de Aplicación residente en la plataforma de middleware. La razón fundamental para integrar el Nivel de Presentación Web con la Lógica de Aplicación es para lograr eficiencia en la entrega de contenidos a través del canal de la Web y, también simplificar la gestión de las aplicaciones Web. Por su parte, la Gestión de Recurso está lograda gracias al uso de arquitecturas estándares de conexión y APIs de conectividad a bases de datos, de igual forma que los middleware convencionales (véase Fig. 4.9).

4.3.1. Funcionalidades principales de un Servidor de Aplicación

Existen dos tecnologías reinantes en el mercado que compiten para proveer un marco de trabajo para middlewares basados en Web: ***Java EE*** de Sun Microsystems [461] y ***.Net*** de Microsoft [305, 319]. Estos dos productos, en términos de funcionalidad, son similares. Con el objeto de mostrar las principales características y funcionalidades de un Servidor de Aplicación, se mostrará solamente uno de ellos, en este caso Java EE (Java Enterprise Edition).

En la Fig. 4.10 se muestran los principales componentes de la especificación Java EE, en concordancia con las funcionalidades descritas en la Fig. 4.9. Además, Java EE incluye otras APIs cuya implementación provee la funcionalidad común requerida para los proyectos de integración, tales como broker de objetos y manejo de transacciones.

Como se puede observar, la complejidad de la plataforma Java EE pone de manifiesto que los Servidores de Aplicaciones tienden a agregar más y más funcionalidades dentro de la plataforma de middleware. Esto es consistente con la tendencia de lograr un producto integrado que soporte diferentes abstracciones de middleware, como se explicó al tratar la convergencia de las plataformas middleware (véase Sec. 3.1.4).

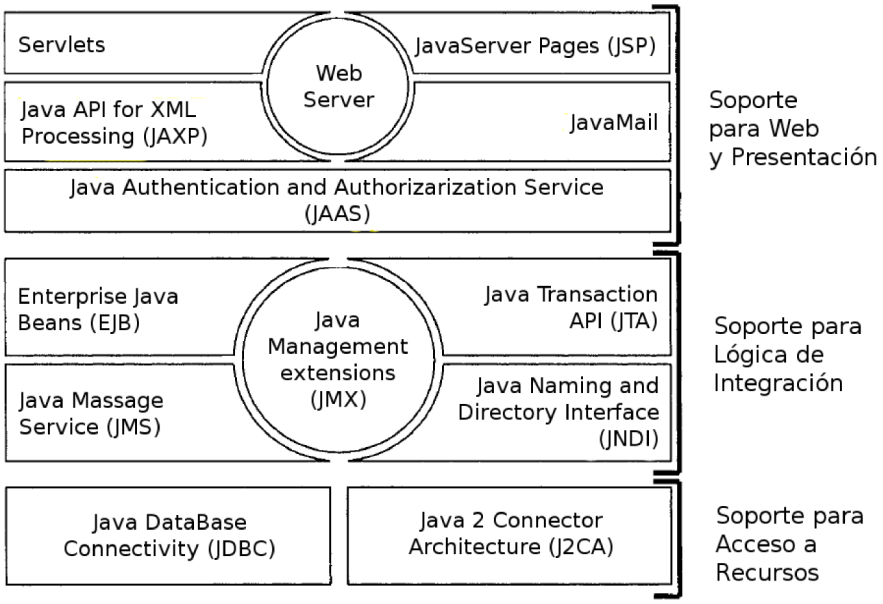

Figura 4.10: Las principales APIs de la especificación Java EE.

4.3.2. Servidores de Aplicación como suporte al Nivel de Lógica de Aplicación

Tomando en cuenta el Nivel de Lógica de Aplicación, los Servidores de Aplicación ocupan el lugar de los middlewares tradicionales, con funcionalidades muy similares a CORBA, Monitores TP y Brokers de Mensajes. Esto es porque los Servidores de Aplicación no se limitan a la integración basada en Web, sino que también pueden ser usados para integración de aplicaciones empresariales. El principal objetivo de los fabricantes de Servidores de Aplicación es ofrecer un único ambiente que albergue todo tipo de Lógica de Aplicación, sea basada en la Web o no. Como parte de este esfuerzo, los Servidores de Aplicación ponen a disposición funcionalidades típicas de middleware (por ej.: transacciones, seguridad, persistencia) para las aplicaciones que se desarrollen en su ambiente.

En Java EE, el soporte al Nivel de Lógica de Aplicación se concentra en tres principales especificaciones: EJB [458], JNDI [464] y JMS [463] (véase Fig. 4.11).

La especificación ***EJB*** *(Enterprise Java Beans)* [458] es aquella en cuya implementación se concentra la columna vertebral de la Lógica de Aplicación. Un ***Enterprise Java Bean*** *(EJB)* es un componente, residente del lado del servidor, que encapsula la lógica de negocios de alguna aplicación. La especificación EJB define tres tipos de beans, clasificados según la forma en que ellos interactúan con otros componentes y en la forma en que estos manejan su estado y persistencia.

- **Beans de Sesión**: manejan sesiones con los clientes. Estos pueden contener estado o no. Ejemplos de beans que mantienen el estado de la conversación con el cliente son los carritos de compras de Internet. Los beans que no mantienen estados, no guardan ninguna conversación con el cliente y, por ende, su procesos pueden ser reutilizados por varios clientes.

- **Beans de Entidades**: son trascendentes a los límites de una sesión con un cliente. Estos tienen estado y son almacenados en una base de datos o otro almacenamiento persistente. La persistencia puede ser manejada por el bean mismo (es decir, se pueden programar las sentencias SQL para salvar el estado del bean en una base de datos), o puede ser manejada por la plataforma EJB.

- **Bean Orientados a Mensajes**: son aquellos que atienden a las interacciones asincrónicas

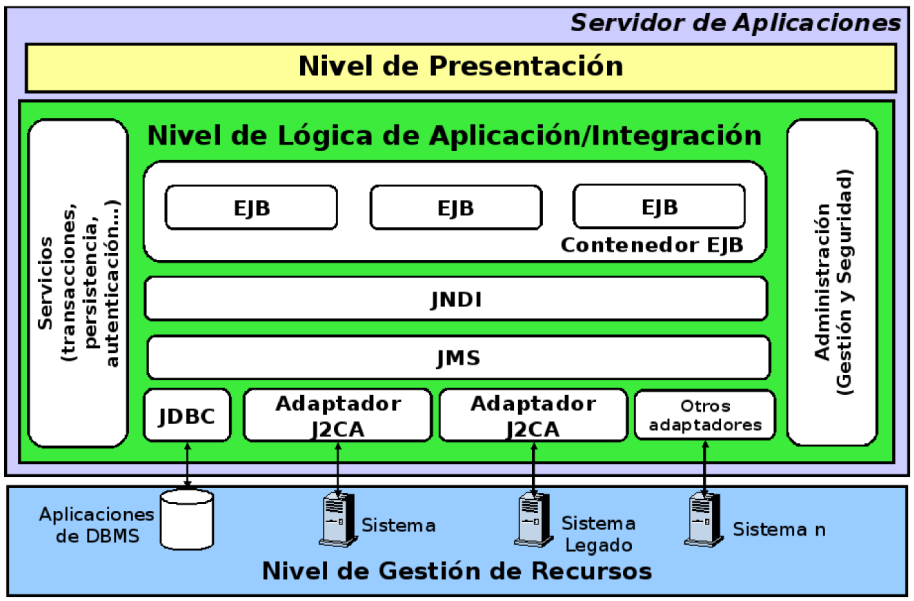

Figura 4.11: Soporte del Nivel de Lógica de Aplicación de los servidores de aplicación basados en Java EE.

con los clientes. Normalmente actúan como escuchas de mensajes de JMS. Los mensajes pueden ser remitidos por cualquier otro componente Java EE o por cualquier aplicación o sistema JMS que no use Java EE. Sin embargo, un bean orientado a mensajes puede procesar otros mesajes que no sean de la especificación JMS.

El ***contenedor EJB*** provee el ambiente en el cual los beans se ejecutan. Todas las interacciones entre los EJBs y otros objetos se producen a través del contendor. El contenedor cumple el rol de mediador o broker (véase Sec. 3.6.2) y como tal, provee una series de servicios entre los cuales sobresalen en soporte a las transacciones y a la persistencia.

Las ligaduras entre EJBs se realizan a través de JNDI *(Java Naming and Directory Interface)* [464]. JNDI define una interfaz para interactuar un directorio de servicios. Con JNDI los clientes pueden ligarse con los servidores basándose en los nombres de los objetos; en el caso de EJB, la ligadura con un servidor involucra ligarse a un objetos que implemente una interfaz de un determinado servicio.

Además de proveer mecanismos y herramientas para desarrollar y soportar el Nivel de Lógica de Aplicación, Java EE también asiste a la conectividad con el Nivel de Gestión de Recursos. Para esta función, Java provee dos APIs y arquitecturas:

- ***JDBC*** *(Java Database Conectivity)* [469]: que permite el acceso a cualquier a casi cualquier fuente de datos tabular, a través de la ejecución de comandos SQL.

- ***J2CA*** *(J2EE Connector Architecture)* [459]: que es la generalización del enfoque de JDBC, ya que permite definir cómo desarrollar un adaptado de cualquier tipo de recurso. Cada *adaptador de recurso* es caracterizado por un contrato entre la aplicación (gestor de recurso) y la plataforma Java EE. El *contrato de aplicación* esencialmente define el API de Java, que las aplicaciones pueden usar para acceder al recurso particular de la Gestión de Recursos. J2CA brinda las posibilidades para interactuar con aplicaciones legadas (véase Sec. 2.1.3) en el entorno Java EE.

Además de implementar Java EE y otras especificaciones no-Java, los Servidores de Aplicación también ofrecen servicios que simplifican la administración de la aplicación y logran performance y alta disponibilidad. También se provee administración y seguridad de objetos, definiendo qué usuarios o aplicaciones pueden acceder a los componentes y forzar restricciones de acceso.

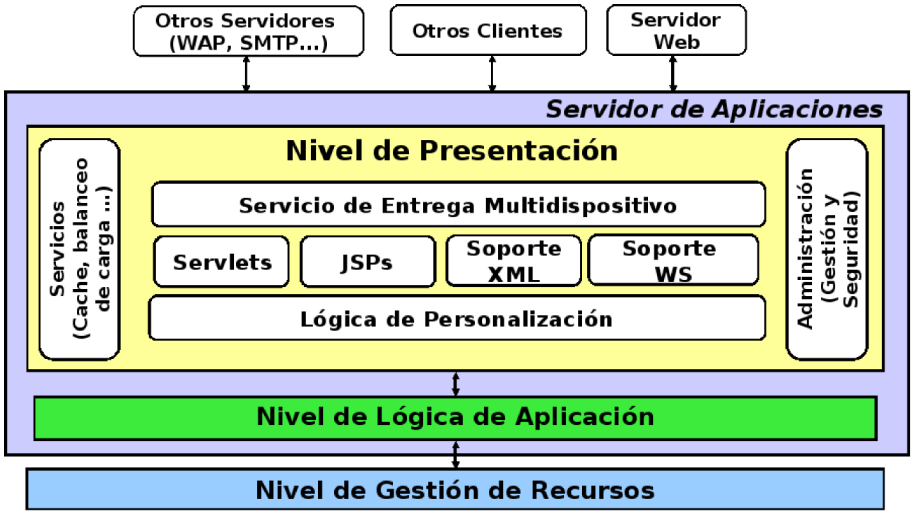

Figura 4.12: Soporte del Nivel de Presentación de los servidores de aplicación basados en Java EE.

La plataforma EJB y la mayoría de los servidores de aplicación están mejorando su manejo automático de persistencia, transacciones y otras funcionalidades [257]. Aún hoy existen balances entre facilidad de desarrollo y funcionalidad. Los Servidores de Aplicación todavía no pueden superar la performance de los Monitores TP, por ejemplo. Los Monitores de Transacciones son una excelente plataforma para soportar aplicaciones de alta carga de procesamiento cuyas características son más bien estáticas, las cuales exigen que las fases de configuración y puesta a punto deban ser delicadas y consumir bastante tiempo. Por otro lado, los Servidores de Aplicación tratan de ser versátiles a la hora de desarrollar y buscan facilitar la evolución de aplicaciones.

4.3.3. Servidores de Aplicación como suporte al Nivel de Presentación

El soporte del Nivel de Presentación y para la gestión de los documentos como la unidad básica de transferencia, es la esencial diferencia que presentan los Servidores de Aplicación con respecto a los middlewares convencionales.

Los Servidores de Aplicación tienen mecanismos similares a los CGI, que han evolucionado lentamente hacia implementaciones más sofisticadas, que hacen que la transición entre los documentos y los argumentos de funciones o procedimientos sea más eficiente, flexible y manejable. Se ofrecen una variedad de posibilidades de presentación para lograr la entrega dinámica de documentos.

Un Servidor de Aplicación moderno provee soporte para interactuar con diferentes tipos de clientes; como por ejemplo navegadores web, aplicaciones locales vinculadas a middlewares convencionales, dispositivos móviles y clientes de Servicios Web. (véase Fig. 4.12)

Los navegadores web interactúan con los servidores de aplicación a través de los servidores web, mediante el protocolo HTTP o HTTPS. La comunicación se realiza a base de intercambio de páginas HTML la cuales pueden ser estáticas o generadas automáticamente a demanda. Para la generación de páginas dinámicas, la plataforma Java EE cuenta con las tecnologías de servlets y JSP (Java Active Pages) [468]. Otras soluciones para generación dinámicas de páginas HTML son PHP [277], ASP [303] y los controles ActiveX de Microsoft [297]. En los principales casos, Java EE y .Net, los componentes dinámicos están vinculados a la Lógica de Aplicación. Por ejemplo, algunas herramientas de desarrollo simplifican la vinculación de los fragmentos JSP con los datos generados por los EJBs a través del contenedor EJB. De esta forma, la funcionalidad JSP adapta la presentación de la información, generada por un EJBs, en un formato adecuado para los navegadores Web.

Es posible que la comunicación se produzca sobre otros protocolos distintos a HTTP; por

ejemplos los applets pueden interactuar con el servidor de aplicación a través de RMI o CORBA/IIOP.

Algunos otros dispositivos, como los teléfonos móviles o PDAs utilizan un protocolo llamado WAP (Wireless Application Protocol) [526, 367], en vez de HTTP; el cual coexiste con el lenguaje de presentación WML (Wireless Markup Language) [525], cumpliendo el rol del HTML.

Como se puede ver, los servidores de aplicaciones proveen la transparencia para interactuar con diferentes protocolos y lenguajes de presentación, ya que transforman el documento de la forma que lo necesita el cliente. Esto hace que los desarrolladores no tengan que escribir diferente código para diferentes clientes.

Además de soportar diferentes clientes, los Servidores de Aplicación pueden soportar diferentes usuarios. Una de las prestaciones que ofrecen es la *personalización*, es decir la habilidad de proveer diferentes contenidos y diferentes estéticas basándose en las cualidades del usuario destinatario de los documentos. Esto es logrado a través de la definición de un conjunto de reglas condición/acción. La condición sirve para identificar si la petición cumple con un determinado perfil. La acción define el formato y contenido que debe ser remitido en respuesta a la petición que satisface la condición. Por ejemplo, puede personalizarse el contenido mostrado en un portal web eductivo, dependiendo si el usuario es un profesor o un alumno. Como se puede apreciar en la Fig. 4.12, los Servidores de Aplicaciones proveen un grupo de servicios que permiten manejar la performance, la disponibilidad y la seguridad de los servlets, JSPs y otros componentes que permiten exponer el Nivel de Presentación.

4.4. Tecnologías Web para la Integración de Aplicaciones

HTTP puede ser utilizado para permitir que dos sistemas informáticos diferentes puedan interactuar a través de Internet. Si bien no es una solución perfecta y presenta considerables limitaciones, las tecnologías desarrolladas para que HTTP cumpla este propósito fueron las bases las tecnologías de los Servicos Web.

4.4.1. Arquitecturas para la Integración de Áreas Amplias

En la integración de sistemas informáticos a través de redes de área amplia, hay un número de estrategias que, en principio, pueden tomarse. Si como ejemplo, se considera que se deben integrar dos sistemas 3-capas, las posibles estrategias para la integración, pueden aplicarse en cualquiera de los tres niveles o inclusive a nivel de los clientes (véase Fig. 4.13); es decir, integrando los módulos clientes o los Niveles de Presentación, integrando los sistemas de middleware, o bien integrando las capas de Gestión de Recursos.

Como integración a Nivel de Gestión de Recursos se pueden citar los sistemas *federativos de bases de datos (federated database system)*. Un sistema federativo de bases de datos es un sistema de meta-gestión de base de datos, el cual integra en forma transparente múltiples sistemas autónomos de bases de datos (generalmente, ubicados en puntos geográficos distantes) en un único sistema federativo [212].

También es posible crear esquemas de integración que no se den en niveles equivalentes. Por ejemplo, se puede integrar funcionalidades ofrecidas por las dos plataformas distintas de middleware en un mismo cliente; o bien, un mismo middleware, puede tener acceso a los dos sistemas de gestión de recursos.

Antes de que emerja la Web, existían dos opciones realistas para integración. Esta limitación esencialmente se debía a la falta de estandarización. La primera opción viable era utilizar clientes especializados para integrar sistemas al nivel de clientes; siendo esta estratégia la principal desventaja de las arquitecturas Cliente/Servidor (Veáse Sec. 2.2.2). La segunda opción era integrar directamente las capas de middleware. Esto se lograba mediante el reenvío y el redestino

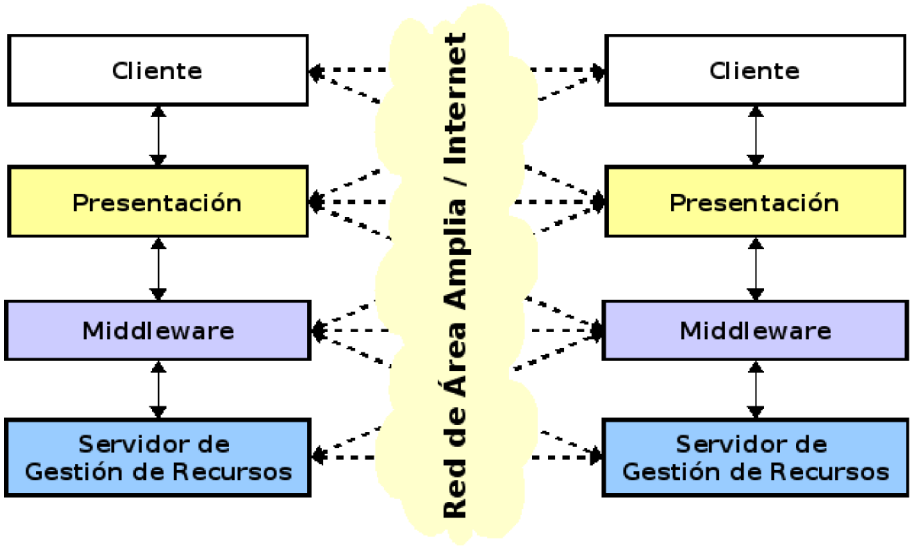

Figura 4.13: Posibles formas de integración de una aplicación 3-capas a través de Internet.

de mensajes desde una plataforma a otra. Este modelo fue el que optó CORBA a través de la utilización del protocolo GIOP, para interoperatividad entre sus ORBs (Veáse Sec. 3.4.4).

Con la aparición de la Web, el escenario de soluciones aplicables se amplió más. Entre varias ventajas, se puede decir que en la Web hay un cliente universal: el navegador Web.

4.4.2. Extensiones de Middleware

Desde el punto de vista de la integración de aplicaciones, la Internet requiere capas adicionales de middleware entre clientes y servidores. Las empresas vendedoras de plataformas de middlewares desarrollaron entonces extensiones en sus mecanismos para soportar a Internet como un canal más de acceso. Algunos de estas extensiones forman el núcleo de los Servicios Web.

Inicialmente, las plataformas de middlewares fueron diseñadas para trabajar dentro de una red local (LAN) de computadoras de un mismo dominio confiable organizacional, dentros de los límites seguros de la red interna. A medida que tales redes se hicieron más grandes, (no solamente en número de máquinas, sino también en diversidad de plataformas y dispositivos que interconectaban) se requirió un esfuerzo de evolución de los middleware que pudieran lidiar con la heterogeneidad de tales universos.

Además de ésto, surge la problemática de la proliferación de LANs, dentro de una misma organización, a medida que informatizan más dependencias o se crean nuevas sucursales. Cada una de estas sucursales puede contener aplicaciones distribuidas que exigen tener su propia plataforma de middleware, las cuales deberán interactuar si se avanza en un proceso de integración de aplicaciones empresariales.

Una situación similar es el escenario dispuesto entre dos compañía con fuertes vínculos comerciales, en donde los insumos de una son provistos por la otra. En ambas situaciones, es necesario integrar las plataformas de middleware con el propósito de automatizar completamente las transacciones entre las diferentes organizaciones o empresas. A este tipo de interacción, cuando excede los límites de la empresa, se la denomina *B2B (Business-to-Business)* [292, 190, 314].

El intercambio electrónico, logrado en tales esfuerzos de integración, es mucho más rápido y más eficiente en costos respecto al intercambio "basado en papel". Es menos propenso a errores y se puede llevar registros del intercambio de datos para monitoreo y análisis. Las compañías pueden reaccionar más rápido a las demandas, tener una noción más actualizada de las actividades comerciales y reducir significativamente la sobrecarga organizativa destinadas a los procesos de intercambio comercial.

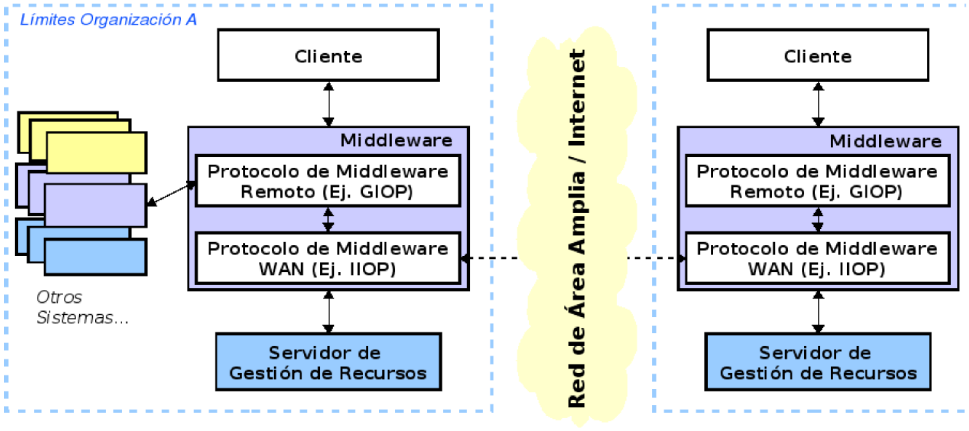

Figura 4.14: Integración directamente a través de las plataformas de middleware.

Técnicamente, implementar este tipo de interacciones requiere la habilidad de invocar servicios residentes en diferentes organizaciones. No hay impedimentos conceptuales para hacer esto, y por consiguiente surge el fuerte interés de realizar integraciones a través de Internet usando extensiones de los protocolos de middleware originales. Un ejemplo de esto es CORBA. Como se explico en la Sec. 3.4.4, CORBA soporta el acceso a objetos residentes en el mismo dominio; es decir, vinculados a un ORB determinado. La integración a escala de Internet es lograda gracias a la integración de varios ORBs, esto es posible en el caso de CORBA por el Protocolo General Inter-ORB o GIOP (General, Inter-ORB Protocol) que especifica cómo reenvíar llamadas de un ORB a otro y obtener la respuesta. Existe una versión especializada del protocolo GIOP, el protocolo IIOP (Internet Inter-ORB Protocol) que traduce las llamadas GIOP en llamadas TCP/IP las cuales pueden trasportarse por Internet. Este es un enfoque simple y eficiente, resultando en una arquitectura similar a la mostrada en la Fig. 4.14.

Sin embargo, esta buena idea tiene la desventaja que rara vez se sostienen las suposiciones subyacentes a este diseño. Por empezar, los ORBs generalmente son incapaces de intercomunicare debido a que están ocultos detrás de cortafuegos, con restricciones significativas en la comunicación. Otro punto no resuelto en la interconexión de dos ORBs, es el acuerdo que debe existir en las definiciones de interfaces y en la representación de datos a ser utilizada por ambos extremos de la interacción. Por último, un servidor de directorio es requerido para el servicios de descubrimiento y ligadura de objetos; y, especialmente en la configuración de inter-organizacional, no queda debidamente claro en dónde este servicio debe residir y cuál empresa debe administrarlo. Como se verá en el capítulo siguiente, este problema es uno de los desafíos más grandes que las tecnologías de Servicios Web quieren solucionar.

4.4.3. Cortafuegos y tuneleo a través de HTTP

En si, los cortafuegos son un impedimento importante para la integración de sistemas interempresariales. Los cortafuegos [83, 5] se instauran en una estructura informática empresarial con el propósito de proteger las redes internas, los datos y los recursos informáticos. Los cortafuegos actúan como una barrera en contra del tráfico no deseado de red. Para lograr esto, se bloquean determinados canales de comunicación. Un cortafuegos bien configurado, inhabilita aquellos canales utilizados por los productos más comunes de EAI. Es así, que los desarrolladores no pueden utilizar RPCs, RMI o GIOP/IIOP para enlazar diferentes componentes de diferentes organizaciones, en una aplicación distribuida.

En las plataformas tradicionales este problema no existe, porque todos los componentes operan dentro de la misma red, o a través de redes que tienen tráfico transparentes entre ellas. De esta forma, es fácil ligar servicios, intercambiar datos, y controlar todas las interacciones

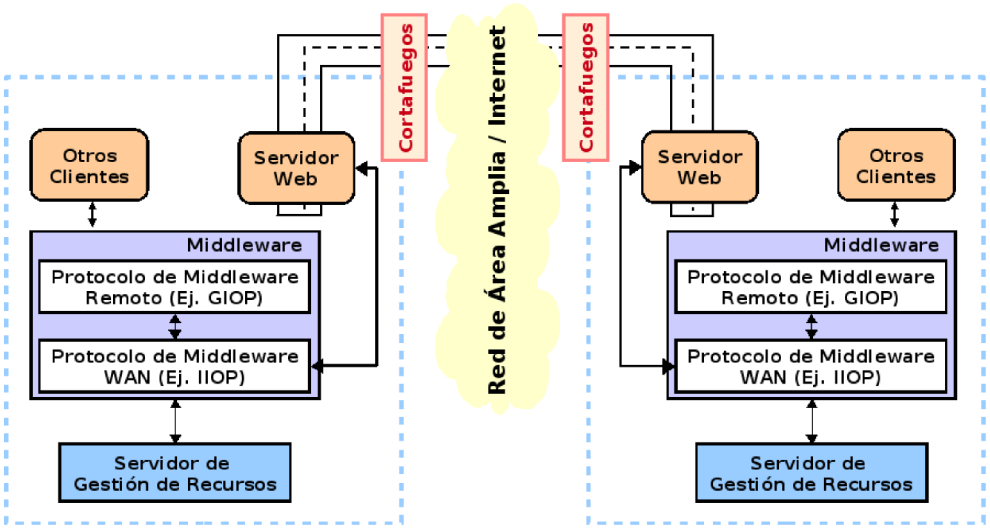

Figura 4.15: Integración B2B a nivel de middleware usando tuneleo.

entre clientes y servicios, entre servicios y la gestión de recursos, y entre dos o más servicios. Por ende, no es complicado reencaminar peticiones, administrar balanceo de carga del sistema, replicar servicios, etc. El principio básico que sustenta tal manejo, es que cada componente del sistema tiene confianza en los otros, y cualquier componente puede conocer a sus pares.

Para el escenario de clientes remotos descrito en la Fig. 4.5 y el de integración de a través de middleware de la Fig. 4.14, los cortafuegos obligan a cambiar el diseño de dos modos bien diferenciados. En primer lugar, no existe comunicación directa entre los sistemas a integrar, por lo menos no con los protocolos utilizados en las plataformas de middleware convencionales. En segundo término, los cortafuegos están presentes por considerar que las partes externas a la organización no son confiables. El problema de no tener comunicación directa se explica por la falta de confianza en el tráfico generado por fuera de los límites del cortafuego. Por esta razón, el grado de interacción no es el mismo al mostrado por sistemas que están dentro de la misma intranet, o red corporativa, o red interna.

La solución más aceptada a este escenario es ocultar los protocolos que se bloquean por el cortafuegos dentro de protocolos con tráfico permitido. A este estrategia se la llama ***tuneleo*** *(tunneling)* [383, 207]. En términos más simples, una llamada en un protocolo que es rechazado por el cortafuego, se *encapsula* dentro de la llamada de otro protocolo, el cual es admitido. Los mejores ejemplos de esta estratégica es realizar "túneles" a través de HTTP o SSH (Secure Shell). En el caso de HTTP, el tuneleo consiste en usar un software intermediario que convierta el mensaje o petición original en un documento HTML o XML, enviar el documento usando HTTP, y extraer en el destino el mensaje del documento.

Una vez que se traspasan los cortafuegos por tueneleo, los mensajes son tratados de la manera y con las herramientas de middleware convencionales. La arquitectura resultante de este esquema de trabajo es bastante compleja ya que requiere de capas adicionales que agregan un nivel más de indirección. (véase Fig. 4.15)

A pesar de su recientes comienzos, las técnicas de tuneleo se han convertido en la solución de facto elegida para pasar los cortafuegos. El tuneleo se aplica a sistemas de middleware convencionales (como por ejemplo GIOP/IIOP sobre HTTP) y sistemas de Servicios Web (SOAP sobre HTTP), como se verá más adelante. En las mayoría de los casos, el tueneleo se hace en base a HTTP ya que es uno de los pocos protocolos universalmente aceptado por los cortafuegos. Es de destacar, que los cortafuegos han reforzado el establecimiento de la Web como vínculo entre sistemas remotos, ya que las técnicas de tuneleo asumen que existen un servidor Web en ambos extremos de la comunicación.

4.4.4. Representación común de datos

La posibilidad de utilizar los protocolos fundacionales de Internet, FTP, SMTP y HTTP, para automatizar el intercambio de mensajes entre empresas, ha planteado otro importante desafío: tratar de identificar y definir una sintaxis y semántica común para la información intercambiada entre las aplicaciones.

Lo fundamental en una interacción automatizada entre dos partes es contar con la posibilidad de interpretar correctamente la información que está siendo intercambiada. En las plataformas de middleware convencionales (véase Capítulo 3) el problema de la representación e interpretación de datos estaba oculto detrás de los lenguajes de definición de interfaces IDLs. En tales sistemas los IDLs cumplen con dos roles: (1) el lenguaje es utilizado para definir interfaces y (2) la implementación del IDL usa una representación intermedia que define cómo cada tipo de dato utilizado en IDL, es representado en una manera independiente de plataforma. Esta representación intermedia ayuda a superar diferencias entre sistemas operativos y arquitecturas de hardware.

En los sistemas distribuidos interempresariales la complejidad es mayor, ya que no sólo existe heterogeneidad de plataformas interconectadas, sino que los detalles de tales plataformas pueden estar ocultos detrás de cortafuegos.

En el contexto de la integración de aplicaciones empresariales de dominios diferentes, el intercambio de datos debe lidiar con conjuntos de tecnologías, definiciones de mensajes y protocolos diferentes. Aquí surge la necesidad de tratar con definiciones de datos de más alto nivel, denominados ***metadatos*** [402]. Los metadatos son datos acerca de datos y usualmente describen una plantilla o un esquema para los mensajes y para los contenidos de los mensajes.

Intercambio Electrónico de Datos

La importancia de estandarizar la transferencia digital de datos entre empresas surgió hace tiempo ya, principalmente motivado por la posibilidad de realizar transacciones comerciales entre ellas. El ***Intercambio Electrónico de Datos EDI*** *(Electrónic Data Interchange)* [219] es el movimiento de datos electrónicos dentro y entre empresas en una forma estructurada y con datos procesables por computadora. EDI es un esfuerzo mundial destinado a proveer estándares de patrones de metadatos orientados a documentos o transacciones electrónicas para intercambiar información comercial. El escenario del mercado electrónico global fue promovido y desarrollado gracias a que la comunicación de sistemas interempresariales confió en estos estándares. De este modo, la tendencia a utilizar documentos electrónicos surgió y se desarrolló, originando una era digital [377, 448, 219].

Varios estándares fueron formulado y usados para EDI, los dos más importantes son: *ANSI ASC X-12* [20] usado ampliamente en Estados Unidos y su contraparte europea *EDIFACT* [505].

Para explicar algunos conceptos se utilizará el modelo de ***EDIFACT*** *(Electronic Data Interchange for Administration, Commence and Transport)*. EDIFACT provee *plantillas* para los mensajes y los contenidos de los mensajes del un intercambio comercial. EDIFACT define las diferentes partes de un mensaje y cómo organizar sus contenidos, para que sean adecuadamente analizados por el receptor. Un *mensaje* EDIFACT en propiamente un documento, en vez de un mensaje en el sentido de middleware convencional.

Por su parte, un mensaje EDIFACT es una fiel réplica de un documento tradicional en papel, diseñado para su intercambio electrónico. El mensaje EDIFACT contiene toda la información esencial del documento original en forma de estructura de datos jerárquica. Como en el documento papel, existen datos imprescindibles y otros datos complementarios. El mensaje EDIFACT es un formato idóneo para compactar la información del documento original y transmitirla por medio de telecomunicación [345].

Un mensaje EDIFACT es una secuencia de ***segmentos de información*** (segmentos de servicio y de datos, como se explicará). Un *segmento* está integrado por elementos de datos

(simples y/o compuestos, como se verá). Un *elemento de datos compuesto* está formado a su vez por elementos de datos simples. Un *elemento de datos simple* puede ser codificado o no codificado.

Un mensaje se prepara añadiendo todas las informaciones necesarias en sus lugares correspondientes, dentro de la estructura definida, como el documento papel. Para ello el mensaje contiene *componentes* con diferentes *propiedades*, obligatorios u opcionales y únicos o repetibles. Los ***componentes*** afectados por dichas *propiedades* son segmentos y elementos de datos, compuestos o simples. Por último, los elementos de datos deberán contener los datos precisos y los datos codificados incluir los códigos adecuados.

La estructura genérica de un mensaje o paquete EDIFACT contiene típicamente los siguientes campos:

- Un **encabezado de intercambio**: indicando la versión de EDIFACT que se utiliza, los ID del remitente y destinatario, fecha y hora y otros datos de control.
- Un **encabezado de mensaje**: indicando el tipo de mensaje, por ejemplo una factura.
- Uno o varios **segmentos de datos** del usuario: los datos relacionados con la operación comercial o empresarial.
- Una **finalización de mensaje**: utilizado para indicar el fin del mensajes y como chequeo de integridad.
- Una **finalización de intercambio**: utilizado para indicar el fin del intercambio y como chequeo de integridad.

Cada uno de los segmentos está identificado por un código de tres letras. Los mensajes son codificados en texto plano, en ASCII; y manejados a ambos lados de la conexión por *convertidores*, los cuales traducen el mensajes a formatos específicos de aplicación y viceversa. Generalmente estos convertidores son analizadores sintácticos (parsers) automáticos.

Además de definir la estructura de un mensaje, EDIFACT también define y estandariza una colección extensa de tipos de mensajes o de tipos de documentos, entre los cuales se cuenta: factura, petición de orden de compra, respuesta a orden de compra, consulta de estado de orden de compra, informe de estado de orden de compra, etc.

Mediante la provisión de representaciones universales estándares de diferentes tipos de intercambios comerciales, EDIFACT facilita realizar tales intercambios electrónicamente. La desventaja de EDIFACT, es que trata de proveer representaciones universales para cada uno de los posibles intercambios comerciales que pueden producirse. El resultante es un estándar extremadamente complejo. Tal complejidad en algunos casos no es necesaria porque pueden existir aplicaciones que sólo usen una fracción de la información de un mensajes; es decir, que algunos segmentos del mensajes son inútiles. En tales casos, para preservar el estándar, los convertidores debe construirse para la completa generalidad del mensaje, de todas formas. Además, EDIFACT puede ser utilizado solamente para intercambios estandarizados. Razón por la cual, cualquier otra información que se quiera intercambiar que escape a cualquier patrón de un documento estándar, debe ser tratadas con mecanismos especializados que deben integrarse a los convertidores de EDIFACT.

En la perspectiva de la representación común de datos para posibilitar la integración de aplicaciones diferentes a través de la Web, lo propuesto por el Intercambio Electrónico de Datos y sus estándares como EDIFACT es fundacional, por proveer un marco inicial para la estandarización de las transacciones comerciales, las cuales son cubiertas casi en su totalidad. Sin embargo, por ser una tecnología estándar especializada en los patrones comerciales, deja afuera gran cantidad de intercambios de información relacionadas con otros campos. Esto tiene su mayor manifestación con la Web, la cual transporta datos en formatos tan variados y ricos que desafían cualquier proceso de estandarización.

XML

Con el amplio uso de Internet y su aplicación para la integración de aplicaciones, se hizo sumamente difícil estandarizar todas sus formas de interacción. El éxito de EDI llega a ser casi imposible de extender a determinadas áreas, simplemente porque el cúmulo de variedad de información a intercambiar es enorme. Es imposible formalizar todos los patrones probables de mensajes que pueden intercambiarse. Para este problema, la respuesta fue el lenguaje ***XML*** *(Extensible Markup Language)* [276]. XML afronta el problema de representación de datos enfocándose en la sintaxis, antes que en la semántica del documento a intercambiar. A diferencia de sus predecesores como EDIFACT, el cual provee plantillas estándares para el contenido de los mensajes, XML provee una forma universal para definir estructuras de documentos adecuadas para el procesamiento automático. Además, el hecho de contar con una manera estándar de codificar la estructura posibilita el desarrollo de herramientas automatizadas para analizar los documentos y extraer la estructura y contenido.

Otra forma de automatización soportado por los documentos XML es la validación. Por un lado la especificación del estándar XML y el mecanismo que la implementa definen si un documento esta bien formado, en concordancia con un conjuntos de reglas de sintaxis. Un paso siguiente es lograr definir tipos de documentos para poder establecer si un documento conforma o no cierto tipo. Por ejemplo, puede definirse el tipo de documento relacionado con los pedidos de presupuestos. En tal definición de tipo, se podrá exigir que los mensajes tengan como *campos* o *segmentos*: datos del cliente (el que pide el presupuesto), un id de producto y una fecha de solicitada de entrega. Estas y otras restricciones sobre la estructura del documento pueden ser definidas por las ***especificaciones DTD*** *(Document Type Definitions)* [520, 271] o por los llamados ***esquemas de XML*** *(XML Schemas)* [524].

Cuando una especificación de tipo es hecha en DTD o XMLSchema, pueden declararse que un documento en particular es de ese tipo determinado. Herramientas automáticas pueden validar si la estructura y los contenidos de un documento se corresponden con el tipo o esquema de su declaración. Sin embargo, determinar a través de una especificación DTD o un esquema XML cómo debe constituirse la estructura y cuáles son los campos exigidos en un mensaje, no es suficiente para clarificar la semántica del mismo y establecer la forma en que débe tratarlo el receptor.

Para comparar la diferencia en enfoques, consideraremos una factura electrónica representada por EDIFACT y XML. EDIFACT provee un formato de factura universal que contiene toda la información posible que puede ser asociada con una factura. Este formato o formulario electrónico es el mismo para todas las aplicaciones y el significado de cada campo o segmento está estipulado por el estándar. Por otro lado, XML no define un formato universal para facturas. Una aplicación, además del documento que representa la factura electrónica, necesita su definición DTD o el esquema XML de la misma. Puede darse, que diferentes sistemas utilicen diferentes tipos de facturas, cuyos tipos son especificados por diferentes DTDs. Además, el significado de cada campo presente en la factura no está definido por XML; queda bajo la responsabilidad de la aplicación que consume el documento, darle significado a cada elemento una vez extraído.

Sin embargo, gracias a que se provee una sintaxis que hace posible desarrollar herramienta de análisis y validación automáticas, XML establece las bases en las cuales la semántica puede ser definida. Algunos consorcios de estandarización están en la tarea de especificar tipos de documentos a través de DTDs o esquemas XML y también definir su semántica. Un ejemplo de esto es consorcio RosettaNet [416], el cual ha establecido una serie de especificaciones estándares para tipos documentos y también sus semántica, relacionados con transacciones B2B en el dominio de las tecnologías de información.

4.5. Resumen

Lograr sistemas de arquitecturas multicapas ha tomado décadas. Las tecnologías que lo hacen posible están en su mayoría maduras, pero algunas siguen evolucionando. En este contexto, se puede decir que se comprende y se sabe cómo desarrollar sistemas n-capas en forma robusta, eficiente y extensible. Debido a que este logro tecnológico ha exigido mucho esfuerzo, es natural ver que los vendedores y productores de software, y los diseñadores y desarrolladores de sistemas traten de utilizar esta misma tecnología para integrar operaciones a través de Internet.

En si, el principal problema no es la interconectividad de sistemas a través de Internet. En la mayoría de los casos, no es mayor problema tener un cliente que acceda por Internet y tratarlo de la misma forma que a uno que exista en una LAN. El problema es como tratar con las implicaciones arquitecturales y las restricciones de diseño que se generan por usar a la Web cómo canal de comunicación entre dos sistemas a integrar.

En este capítulo se estudiaron estos problemas y se exploraron las soluciones a estas problemáticas. En primer término se introdujeron las tecnologías fundamentales de la Web; en especial el protocolos HTTP que sirve para el transporte de información, posibilitando la interacción entre clientes y servidores remotos. En sí, el protocolo HTTP es liviano y eficiente, y cuenta con la aceptación de la mayoría de los cortafuegos. Además, convive con HTML como medio de descripción de los documentos de Web. En conjunto HTTP y HTML se forjaron como estándares para el transporte y la presentación de recursos a través de la web; sin embargo poseen carencias estructurales que los hacen limitados para la integración de aplicaciones.

Para solucionar las limitaciones de HTTP y HTML, se presentan a varias tecnologías que extienden sus posibilidades. En tal sentido, se desarrollan arquitecturas de clientes remotos (Applets), y también extensiones a los servidores web para que puedan acoplarse a aplicaciones empresariales (CGI y Servlets).

Sin embargo, estas extensiones no fueron suficientes para los requerimientos cada vez más crecientes de integración. Como alternativa, las plataformas de Servidores de Aplicación surgen para proveer todas las características y funcionalidades de las plataformas de middleware tradicionales, con la diferencia fundamental de utilizar a la Web como principal canal de acceso a los servicios implementados. Es por eso que existen Servidores de Aplicaciones que surgen de la evolución de plataformas de middleware tradicionales, y otros productos que fueron desarrollados especialmente para la Web.

El capítulo finaliza planteando los problemas concretos de llevar una infraestructura de middleware a usar a la Web como principal medio de acceso a sus componentes y aplicaciones. En este sentido se explica las extensiones de los middleware para soportar interacciones B2B a cualquier nivel de integración, el tuneleo de HTTP para lograr aceptación de trasporte en los cortafuegos y la representación común de datos a intercambiar.

Estas soluciones son construida a partir de extensiones de tecnologías y productos utilizados en los sistemas de middleware tradicionales. Sin embargo; tiene un punto de limitación al querer buscar estandarización, escalabilidad y adaptabilidad de los sistemas distribuidos. En el siguiente paso de la evolución de las plataformas de middleware, surgen como paradigma los Servicios Web. La principal directriz de este paradigma es lograr el nivel de estandarización en los intercambios de datos y en el formato de los mismos; brindar un marco común de descripción de servicios para eliminar las barreras de especificaciones heterogéneas y establecer un mecanismo común de interacción logrando la interoperatividad, costosa hasta aquí.

Capítulo 5

Servicios Web y sus Tecnologías Fundamentales

En los capítulos anteriores se exploró la evolución de las tecnologías que hacen posible la construcción y ejecución de sistemas distribuidos. Cada una de esas tecnologías fueron exitosas en menor o mayor medida; todas trataban de solucionar grandes problemáticas: la integración e interoperatividad de aplicaciones y la reutilización de componentes.

Sin embargo el éxito de tales tecnologías ha sido restringido a determinados contextos y dominios tecnológicos, por ejemplo: sistemas basados en LANs, plataformas de middleware homogéneos, etc. Para lograr derribar las barreras que tales tecnologías imponen a problemas de integración reales y complejos, se requiere ir un paso más adelante de lo que ofrecen las tradicionales plataformas de middleware y de EAI. Los Servicios Web y las tecnologías asociadas han tomado la responsabilidad de dar dicho paso [10].

Este capítulo presenta los fundamentos de los Servicios Web. Se muestra a la tecnología que brinda posibilidades para exponer la funcionalidad de una aplicación y hacerla disponible para su reutilización a través de la Web. El principal logro de los Servicios Web como paradigma tecnológico es el hecho que sus principios básicos de funcionamiento son implementados a través de estándares. El uso de tecnologías estándares reduce la heterogeneidad, y por consiguiente facilita la integración e interoperatividad de aplicaciones, aumentando las posibilidades de reutilización de cada componente expuesto como servicio web.

Las tecnologías de Servicios Web promueve la creación de nuevos paradigmas de desarrollo y arquitecturas de sistemas. En particular, son el engranaje principal para la computación orientada a servicios, un paradigma instaurado en la teoría hace tiempo, pero nunca llevado a la práctica.

Ciertos autores [327] consideran a los Servicios Web como un importantísimo avance en la evolución de los sistemas distribuidos. En forma optimista, consideran que cuando todos los programas y sistemas de software sean finalmente expuestos como servicios web, el mundo de la computación distribuida será muy diferente al que es hoy.

5.1. Introducción y Motivación

Los intentos de integración de sistemas, en base a las primeras tecnologías de middleware explicadas en los capítulos anteriores (RMI, Monitores TP, CORBA), han generados sistemas distribuidos cuyos componentes están altamente acoplados para ser efectivos en escenarios de EAI o integraciones B2B sobre Web [483]. Si bien, las plataformas MOM y los Brokers de Mensajes disminuyen el grado de acoplamiento, exigen lograr un contexto común entre las organizaciones intervinientes para que se establezcan los sistemas B2B. Obtener este contexto y escenario de trabajo común requiere que se realicen demasiados acuerdos empresariales relacionados con establecer confianza en los accesos externos a los sistemas de cada empresa, a la información

que se comparte y a la administración compartida o delegada de la plataforma que soporte la integración (véase Sec. 4.1).

Algunos otros problemas planteados desde los enfoques de middleware anteriores son los siguientes:

- La mayoría de los proyectos de integración basados en estas tecnologías producen sistemas altamente complejos y requieren una gran infraestructura de inversión [98].
- Las plataformas de middleware tradicionales dependen fuertemente de un ambiente centralizado de administración y gestión. En el caso de integración de varias aplicaciones empresariales, la administración de la plataforma de integración genera problemas en la definición de roles entre los equipos técnicos de cada empresa. Se requiere un alto grado de acuerdo para definir el contexto compartido entre sistemas de diferentes organizaciones, para así permitir una comunicación B2B abierta y confiable [484].
- La infraestructura de software provista por un determinado vendedor de software es, en la mayoría de los casos, limitado a una plataforma específica. Por ejemplo, en una aplicación distribuida basada en COM+/DCOM, los componentes deben estar residentes en plataformas de Microsoft [484].
- Los middleware estudiados, utilizan protocolos de comunicación complejos, y en algunos casos propietarios; cada uno con un formato de mensajes y una representación de datos propia. En el caso de integrar componentes que sean administrados por dos plataformas diferentes, los desarrolladores deben lidiar con esta heterogeneidad. Una integración inter-plataforma aplicación-a-aplicación A2A de varias aplicaciones es onerosa en tiempo y dinero, y en muchos casos inviable [98].
- Generalmente, la posibilidad de invocación de métodos o programas remotos esta restringida a un determinado ambiente con límites bien definidos, y en la mayoría de los casos dentro de dominios de confianza. Las plataformas tradicionales, no están diseñadas para soportar interacciones con partes fuera de ese ambiente.
- Las plataformas de middleware convencionales (especialmente RPCs, CORBA, DCOM/-COM+) no están diseñadas para traspasar cortafuegos (véase Sec. 4.4), para hacer posible el acceso a sus componentes a través de la Web.

En la búsqueda de alternativas para hacer frente a estos problemas, el concepto de la Web ha sido tenido muy en cuenta. La Web está basada en estándares abiertos y simples. Como se explicó en el Capítulo 4, el intercambio de mensajes se realiza a través de HTTP, siendo este un protocolo no propietario de fácil uso el cual está disponible para todas las plataformas [98]. Además, la mayoría de los cortafuegos en la Web permiten el tráfico HTTP [430]. Como medio de presentación, se utiliza HTML, el cual es ampliamente aceptado en la Web [394]. Todas estas características permiten establecer a la Web como un sistema abierto y viable para ser utilizado como canal de comunicación entre empresas y plataformas.

Sin embargo, en la configuración actual de la Web sus interfaces están orientadas a la comprensión humana; es decir, la interpretación semántica de su contenido es dejada a las personas [394]. La Web está adecuada para el acceso de usuarios humanos, no está orientada a integraciones A2A aplicación-a-aplicación [281].

El concepto *Servicios Web* encierra el propósito de adoptar los principios de la Web, como sistema abierto y estándar de alta interoperatividad y acceso, para aplicarlos a la integración A2A. De esta forma poder lograr el mismo nivel de apertura entre sistemas con bajo grado de acoplamiento, a pesar de las diferencias en lenguajes de programación y múltiples plataformas involucradas.

El aspecto fundamental para la amplia aceptación de la Web, como canal de interoperatividad, es su cualidad de estar basada en estándares. Es lógico entender que los Servicios Web estén

basados en estándares también. En sí, las tecnologías de los Servicios Web utilizan estándares ya conocidos para la Web, por ejemplo HTTP como protocolo de transporte, aunque no se limita a este único. Por otro lado, nuevos estándares fueron definidos bajo la supervisión de autoridades y consorcios independientes conformados para tal fin.

Como ejemplo principal de este nuevo grupo de estándares se puede citar a XML (véase Sec. 4.4.4). El hecho de usar XML como tecnología base, hace posible que los Servicios Web sean un concepto robusto e independiente de plataformas, vendedores, aplicaciones y lenguajes de programación [98, 430].

5.2. Definición de Servicio Web

Existen varias formas de definir qué es un servicios web. Las definiciones van desde las más genéricas y flexibles hasta las más restringidas y estrictas [10, 115].

Por ejemplo, un **servicio web** puede definirse como *"cualquier aplicación accesible por otra aplicación a través de la Web"* [477, 441, 74]. Esta es una definición bastante abierta, y considera que es un servicio web cualquier cosa software que accesible a través de un URL. De esta forma, un servicio web puede ser un programa CGI.

Una definición de igual sentido, pero haciendo incapié en las cualidades intrínsecas de los servicios web, dice que *"son aplicaciones modulares autocontenidas y autodescriptivas que pueden publicarse, localizarse e invocarse a través de la Web"* [231]. Esta definición, un poco más específica que la anterior, hace referencia a la posibilidad de acceder a un programa a través de la Web el cual tiene una interfaz estable publicada en algún servicio de directorio y que puede recuperarse para hacer posible la invocación de dicho programa.

Una descripción más precisa, es la que ofrece el Consorcio UDDI, que caracteriza a los servicios web como *"aplicaciones empresariales modulares y autocontenidas que tienen interfaces abiertas, basadas en estándares y orientadas a Internet"* [334]. Esta definición es más específica, enfatizando la cualidad de ser compatible con los estándares Web y la adopción de interfaces abiertas y públicas. Sin embargo, la definición no es puntual a la hora de especificar qué se entiende por "aplicaciones modulares y autocontenidas".

Una definición más refinada es la propuesta por el World Wide Web Consortium (W3C), y específicamente en el grupo involucrado en la Actividad de Servicios Web. Define a un servicio web como *"una aplicación de software identificada por un URI, cuyas interfaces y ligaduras son capaces de ser definidas, descritas y descubiertas por artefactos XML. Un servicio web soporta interacciones directas con otros agentes de software usando mensajes basados en XML intercambiados por medios de los protocolos de Internet"* [21]. Esta es una definición más adecuada y precisa, la cual también indica cómo deben trabajar los servicios web. La definición puntualiza que un servicio web debe ser capaz de ser definido, descrito y descubierto, dejando claro el sentido del concepto de *"accesible"* y dando una acabada noción de "interfaces orientadas a Internet basadas en estándares". Por otro lado, asevera que los servicios web son componentes que pueden ser integrados en aplicaciones distribuidas más complejas; y por consiguiente son bienes reutilizables en varias aplicaciones y contextos. Esta es la definición que secunda el presente trabajo, y es la principal razón de explicar previamente los conceptos de middleware como primer paso para entender la tecnologías de los Servicios Web.

5.3. Servicios Web como plataforma de Middleware

Servicios Web, como paradigma tecnológico, han sido considerado como una revolución en la forma de ver a los sistemas distribuidos [327] y también como una evolución lógica producto de la maduración de los principios y tecnologías de las plataformas de middleware precedentes [10].

En este proceso de revolución/evolución, Servicios Web han mejorado el estado de arte en varios sentidos. A tal efecto, se explicarán los principales efectos sobre los conceptos de middleware:

los Servicios Web como implementación de SOA, la evolución de protocolos de comunicación y la estandarización.

5.3.1. Servicios Web como implementación de la Arquitectura Orientada a Servicios.

Las ideas promovidas por la ***Arquitectura Orientadas a Servicios SOA*** *(Service Oriented Architecture)*, deseadas durante tantos años, se convierten en realidad por encontrar en Servicios Web su medio tecnológico de realización; de la misma forma que las RPCs lo son para las arquitecturas cliente/servidor.

Para entender el concepto de servicio, como construcción de software, es necesario ver su evolución a partir de los principios de orientación a objetos y de diseño basado en componentes [146]. Al tratarse de *objetos*, se describe la esencia del análisis y diseño orientado a objetos como el considerar "al dominio de un problema y su solución lógica desde la perspectiva de objetos (cosas, conceptos, o entidades)" [269]. Y estos objetos deben ser "caracterizados por un número de operaciones y un estado que recuerde los efectos de estas operaciones" [237].

En el análisis orientado a objetos, tales objetos son identificados y descriptos en el dominio del problema; mientras que en el diseño orientado a objetos, estos objetos son traducidos en objetos lógicos de software que pueden ser implementados en un lenguaje de programación orientado a objetos. Con el análisis y diseño orientado a objetos, ciertos aspectos de los objetos (o grupos de objetos) pueden ser encapsulados para simplificar el análisis de escenarios empresariales complejos. Ciertas características de los objetos también pueden ser abstraídas de manera tal que sólo los aspectos más importantes o esenciales sean capturados, con el propósito de reducir la complejidad.

Una evolución a partir de los objetos es la concepción de *componente*. El diseño basado en componentes no es una nueva tecnología. Es una evolución del paradigma de objetos y surge como una necesidad de constituir entidades de software que sean de mayor granularidad que los objetos, ya que la fina granularidad de los objetos hace inviable la práctica del reuso. Los componentes de alta granularidad comenzaron a ser el principal objetivo de las prácticas de reuso en el desarrollo de aplicaciones y la integración de aplicaciones. Estos componentes de alta granularidad encierran una determinada funcionalidad bien definida por medio de un conjunto cohesivo de objetos de baja granularidad.

Una vez que una organización alcanza un alto nivel de maduración arquitectónico en sus estructuras informáticas, basada en una clara distinción de funcionalidades separadas de componentes, las aplicaciones empresariales pueden ser particionadas en un conjunto de componentes cada vez más grande en granularidad. Los componentes pueden ser vistos como el mecanismo para empaquetar, gestionar y exponer servicios.

Algunos autores [9] introducen la noción de "servicio" estrechamente vinculada con la noción de "componente". Un *componente* es una unidad de código ejecutable que provee una encapsulación física similar a una caja-negra de los servicios que implementa. Este servicio solamente puede ser accedido a través de una interfaz pública/publicada que incluye un estándar para la interacción. Un componente debe ser capaz de ser conectado a otros componentes (a través de sus interfaces).

Un *servicio* es generalmente implementado como una entidad de software, de granularidad moderada/gruesa, capaz de ser descubierto y que existe como una única instancia e interactúa con otros servicios por medio de modelo de comunicación simple, desacoplado, orientado a mensajes.

Una Arquitectura Basada en Servicios trabaja bajo la suposición que las funcionalidades disponibles de una compañía serán expuestas como servicios. En términos de middleware, un servicio es un componente de software con una interfaz publicada y estable que puede ser invocada por una aplicación cliente. La invocación es hecha por un programa; es decir, que la petición y la ejecución de un servicio web involucra una llamada de un programa a otro programa.

En las Arquitecturas Basadas en Servicios (y en desarrollo basado en componentes), el diseño de las interfaces es hecho de forma tal que la entidad de software implemente y exponga parte de su definición. Por consiguiente, la noción y el concepto de *"interfaz"* es clave en el éxito del diseño orientado a componentes y orientados a servicios. Algunas definiciones relacionadas con interfaces son:

- **Interfaz**: define un conjunto de signaturas de métodos, lógicamente agrupadas, sin proveer realización de tales métodos. Una interfaz define un contrato, o manifiesto, entre el consumidor y el productor de servicio. Cualquier implementación de una interfaz debe proveer la realización de todos los métodos.

- **Interfaz Publicada**: es una interfaz unívocamente identificable y hecha disponible por medio de un registro para el descubrimiento dinámico de los clientes.

- **Interfaz Pública**: una interfaz que está disponible para que los clientes la usen, pero no está publicada; de esta forma, se requiere conocimiento estático de la interfaz de parte del cliente.

En términos de uso, los servicios web no son diferentes a cualquier servicio expuesto a través de un middleware, con la excepción que es posible invocarlos a través de la Web, cruzando eventualmente los límites entre varias compañías. Como consecuencia de esto, los servicios web son de bajo acomplamiento, debido a que son definidos, desarrollados y gestionados por diferentes empresas. Los Servicios Web son la tecnología más ampliamente adoptada y popular por ser exitosa al implementarse en escenarios adecuados para arquitecturas orientadas a servicios.

De hecho, con Servicios Web, los diseñadores y desarrolladores son alentados a pensar que *"todo es un servicio"*, y que los servicios son autónomos e independientes. Esta forma de construcción de software tiene importantes implicaciones ya que conduce a aplicaciones desacopladas y más modulares. Por consiguiente, los componentes individuales pueden ser reutilizados y ensamblados más fácilmente y en diferentes formas.

Hay que tener en cuenta que no todo lo que está disponible a través de la Web es un servicio web. Este es un error común que trae consigo bastante confusión cuando se define las tecnologías de Servicios Web. Hay diferencia entre los servicios en el sentido de software y los servicios en el sentido general. Los servicios -en el sentido general- son actividades llevadas a cabo por una persona o empresa en el nombre de otra persona o empresa. Como por ejemplo, las agencias de viajes, los restaurantes, prestan servicios a personas. En algunos casos esas personas clientes pueden obtener tales servicios a través del servidor web de la empresa; esto no es un servicio web. Un servicio web es una aplicación de software con una interfaz de programación pública y estable, no es un sitio web.

5.3.2. Servicios Web y la evolución de los protocolos.

Un punto importante en la historia de los sistemas distribuidos es la evolución de los protocolos de comunicación. El enfoque de Servicios Web requiere del rediseño de los ***protocolos de middleware*** para que trabajen en forma par-a-par y traspasando los límites "informáticos" de las compañías. Los protocolos tradicionales de middleware, como el 2PC (Pág. 49), no fueron diseñados para soportar interacciones interorganizacionales. Puntualmente el protocolo 2PC supone la existencia de un coordinador central con facultades para bloquear recursos (cohortes) indefinidamente. Como se explicó en Capítulo 4 (Pág. 83) las cuestiones relacionadas con la falta de confianza y la confidencialidad entre empresas partes, van en desmedro de coordinadores centrales. De cierta forma, los esquemas de trabajo concebidos por las plataformas centralizadas tradicionales deben ser rediseñados para implementar protocolos de comunicación que puedan trabajar de forma descentralizada y a través de varios dominios.

5.3.3. Servicios Web y la estandarización.

El último aporte esencial de los Servicios Web en la evolución de las plataformas middleware es la ***estandarización***. En varios sistemas previos de middleware se vio cómo beneficiaba la adopción de estándares en lo relacionado a interoperatividad, adopción de abstracciones y modelos comunes, reducciones de líneas de aprendizaje, etc. Para la tecnología de Servicios Web, donde las interacciones ocurren entre empresas y a escala global, la estandarización no es sólo ventajosa, sino esencial. Esta situación se ve materializada en la concreción de diversos consorcios de empresas de software para lograr estándares. Ejemplo de estos consorcios son el *World Wide Web Consortium (W3C)* [494] orientado a desarrollar tecnologías interoperables para llevar a la Web a su completo potencial, la *Web Services Interoperability Organization (WS-I)* [529] enfocada a establecer Buenas Prácticas para la interoperatividad en servicios web y la *Organization for the Advancement of Structured Information Standards (OASIS)* [331] destinada a promover el desarrollo, la convergencia y adopción de estándares abiertos para la sociedad global de la información. Estas entidades intentan estandarizar todos los aspectos de la interacción; desde la definición de lenguajes para interfaces hasta los formatos de mensajes.

La necesidad de estandarización es también la razón por la cual se plantea a los Servicios Web como plataforma con promesas reales para las necesidades de integración, interoperatividad y reusabilidad. La Web tiene altos niveles de estandarización de por sí, lo cual le permite operar y prosperar sin coordinación centralizada y ha permitido su expansión a niveles insospechados. Las tecnologías de la Web son aceptadas ampliamente y exitosas en la interacciones entre humanos y aplicaciones (a través de navegadores web y servidores web). Es natural entonces que los Servicios Web tengan a la Web como plataforma fundacional y trate de avanzar en el mismo sentido que ésta en términos de estandarización.

5.4. Arquitectura de Servicios Web

La tecnología de Servicios Web es relativamente nueva y ha recibido amplia aceptación como una importante implementación de *Arquitecturas Orientadas a Servicios*. Los principios de Servicios Web promueven un nuevo enfoque a los sistemas distribuidos para integrar aplicaciones extremadamente heterogéneas usando a la Web como canal de acceso. Las especificaciones o descripciones de los servicios web son completamente independiente del lenguaje de programación, del sistema operativo y del hardware subyacente a la implementación, con el propósito de promover el bajo acoplamiento entre el consumidor y el proveedor del servicio [146]. Los Servicios Web permite la construcción de aplicaciones a partir de componentes y el reuso de aplicaciones web tradicionales [74].

Una plataforma de Servicios Web, en concordancia con lo recomendado por las Arquitecturas Orientadas a Servicios, debe tener soporte para las siguientes funcionalidades:

- **Desarrollo** de servicios web de acuerdo a sus funcionalidades previstas
- **Descripción** de servicios a través de un formalismo. Esta descripción debe contener todos los detalles necesarios para interactuar con el servicio.
- **Publicación** de la interfaz del servicio web a través de un servicio de directorio o *registro* de servicios, brindando la descripción del servicio web a aplicaciones clientes.
- **Localización** de servicios web mediante consultas al registro, el cual contesta a las aplicaciones interesadas (clientes) los detalles del servicio que se corresponden con la consulta.
- **Ligadura** entre la aplicación cliente y los servicios web en base a la información contenida en el Registro de las descripciones de los servicios.

- **Invocación** de servicios web a través de la red, usando los datos de ligadura y descripción. La invocación de servicios web se realiza sobre protocolos estándares de transporte, como ser HTTP.

- **Composición** de servicios web, para permitir la construcción de nuevos servicios.

El marco de trabajo de Servicios Web estan basado en estándares tecnológicos. Los principales son [418, 146, 231, 112, 327]:

- El *lenguaje* ***XML*** *(eXtensible Markup Language)* [276] que es la tecnología fundacional para Servicios Web y es la base para las otras tecnologías estándares que utiliza. Provee un lenguaje para definir datos, estructuras y formas de procesamiento. XML es simple de usar, ofrece un formato flexible y extensible para representar datos y está soportado en la mayoría de las plataformas [484].

- El *Lenguaje para descripción de servicios web* ***WSDL*** *(Web Services Description Language)* [71] el cual es una especificación abierta basada en XML que describe las interfaces y las instancias de los servicios. Es ampliable, de modo que se pueden describir los puntos finales *(endpoints)* de una interacción, independientemente de los formatos de mensaje y de los protocolos de red que se utilicen para comunicarse.

- El *Protocolo* ***SOAP*** *(Simple Object Access Protocol)* [264] que provee los medios y modos de comunicación entre los servicios web y las aplicaciones cliente. SOAP es un estándar basado en XML para la transmisión de mensajes en HTTP y otros protocolos de Internet, el cual puede emplearse para interacciones tipo RPC o basadas en mensajes. Es un protocolo ligero para el intercambio de información en un entorno descentralizado y distribuido. SOAP permite el enlace y la utilización de servicios web encontrados definiendo una ruta de mensaje para el direccionamiento de mensajes.

- El *Protocolo* ***UDDI*** *(Universal Description, Discovery and Integration)* [335] utilizado para registrar y publicar servicios web y sus características; de forma tal que puedan ser descubiertos por potenciales aplicaciones clientes. La especificación UDDI define estándares abiertos, independientes de la plataforma, que permiten a las empresas compartir información en un registro público global o privado, encontrar servicios en el registro y definir cómo interactuar con esos servicios conjuntamente en la Web.

Existen otras especificaciones y tecnologías derivadas. Hay que tener en cuenta que el marco de trabajo de Servicios Web es modular; es decir, puede optarse por utilizar sólo algunos estándares. Por consiguiente, los desarrolladores pueden seleccionar las especificaciones disponibles e incorporarlas a medida que maduran las tecnologías, sin perjuicio a los desarrollos previos [112].

El uso de estándares promueve un contexto de interoperatividad amplio entre vendedores de plataformas de middleware y soluciones relacionadas basadas en la tecnología de Servicios Web. Gracias a estos principios, las empresas pueden implementar sus propios servicios web sin requerir ningún conocimiento de los posibles sistemas consumidores de esos servicios y viceversa. Esto facilita la integración de aplicaciones empresariales sin grande limitaciones, promoviendo la integración a demanda o "en tiempo justo" *(just-in-time integration)*; que les permite a las organizaciones establecer aplicaciones B2B en forma dinámica y fácil, integrándose con servicios web de nuevos socios [146].

5.4.1. Roles e interacciones de servicio

Un componente en una arquitectura de servicios Web puede desempeñar uno o varios roles fundamentales: *proveedor de servicios*, *intermediario de servicio* y *cliente de servicio*.

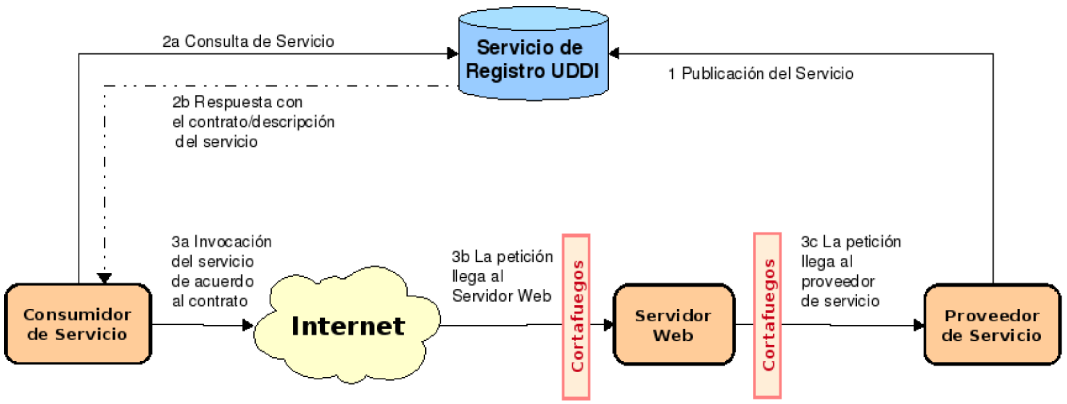

Figura 5.1: Colaboración basada en Servicios Web.

- ***Cliente de servicio*** o ***Consumidor de servicio*** es una aplicación, un módulo u otro servicio que requiere un servicio. Inicializa la búsqueda en un servicio de directorio y nombres, o registro de servicios web, como por ejemplo el Registro UDDI, para descubrir un servicio descrito en WSDL, enlazarse a su proveedor e invocar sus funciones. El enlace implica establecer todos los prerrequisitos de entorno necesarios para realizar satisfactoriamente los servicios; en otras palabras, el cliente debe respetar el contrato o manifiesto que se establece en la interfaz del servicio. Como ejemplos de prerrequisitos de entorno cabe citar la seguridad, la supervisión de transacciones y la disponibilidad de HTTP. El consumidor del servicio ejecuta el servicio de acuerdo a su interfaz de contrato.

- ***Proveedor de Servicios***: es una entidad accesible a través de un URL que acepta y ejecuta peticiones de los consumidores de servicios. Crean y despliegan sus servicios web y pueden publicar la disponibilidad de los servicios descritos mediante interfaces de contrato WSDL en un registro de servicios (como un registro UDDI de empresas). Desde la perspectiva de la empresa, un proveedor es el dueño del servicio; desde una perspectiva arquitectural, es la plataforma que provee acceso al servicio web.

- ***Intermediario de Servicio*** o ***Registro de servicios***: registra y categoriza servicios publicados y proporciona servicios de búsqueda y descubrimiento. Puede contener un repositorio de las interfaces de los servicios disponibles (o bien un vínculo a éstas) y posibilita la localización a través del filtrado de dichas interfaces, de interés para los consumidores. Por ejemplo, UDDI es muy adecuado para actuar como intermediario para Servicios Web descritos por WSDL.

Las relaciones de colaboración entre estos roles se describen en la Fig. 5.1. En la figura se detalla el proceso de publicación -1-. En este sentido, para que un servicio sea accesible, su descripción debe ser publicada de manera tal que pueda ser descubierto e invocado por un consumidor de servicios. También están representadas las tareas de búsqueda y localización -2-. Un consumidor localiza un servicio a través de consultas al Registro de servicios según un criterio de búsqueda y filtrado. Y por último se realiza la ligadura e invocación -3-. Después de recibir la descripción de servicio, el consumidor procede a invocar el servicio de acuerdo a la información del Registro de descripción de servicio. La ligadura puede realizarse en dos fases diferentes del ciclo de vida del consumidor de servicios: al momento del diseño (ligadura estática); o en tiempo de ejecución (ligadura dinámica).

5.4.2. Características de los servicios web

Algunas características esenciales de los servicio web [231], como bienes de software, son:

- Los servicios web son *autocontenidos*. Desarrollar un cliente de un servicio web sólo requiere un lenguaje de programación que soporte XML e interacción a través de HTTP. Del lado del servidor, se necesitan un servidor HTTP y un motor de servlets o servidor de aplicaciones que soporte la interacción con SOAP. El cliente y el servidor pueden implementarse en entornos distintos. En determinados entornos de desarrollo, es posible habilitar como servicio web a cualquier aplicación existente sin escribir una sola línea de código.

- Los servicios web son *autodescriptivos*. El cliente y el servidor sólo deben reconocer el formato y el contenido de los mensajes de petición y respuesta. La definición del formato de mensaje viaja con el mensaje; no se necesitan repositorios de metadatos externos ni herramientas de generación de código.

- Los servicios web son *modulares*. Los servicios web simples pueden componerse para formar servicios web más complejos utilizando técnicas de flujo de trabajo o invocaciones transaccionales.

- Los servicios web son *independientes de la plataforma*. Los servicios Web se basan en un conjunto conciso de estándares abiertos, basados en XML y diseñados para promocionar la interoperatividad entre un servicio web y las aplicaciones clientes en una gran variedad de plataformas informáticas y lenguajes de programación.

Los servicios web pueden ser cualquier cosa documentable, por ejemplo artículos de crítica teatral, partes meteorológicos, comprobaciones de crédito, cotizaciones en bolsa, guías de viaje o procesos de reserva de líneas aéreas. Cada uno de estos servicios de negocio autocontenidos es una aplicación que puede integrarse fácilmente con otros servicios, de la misma compañía o de otra, para crear un proceso de negocio completo. Esta interoperatividad permite a las empresas publicar, descubrir y enlazar dinámicamente una gran variedad de servicios Web por Internet.

5.4.3. Categorías de servicios web

Los servicios web pueden agruparse en tres categorías [146]:

- **Información de empresa**. Una empresa comparte información con consumidores u otras empresas. En este caso, la empresa utiliza servicios web para expandir su ámbito. Ejemplos de este tipo de servicios, son los partes de informaciones financieras, los canales de noticias, partes meteorológicos o cotizaciones en bolsa.

- **Integración empresarial**. Una empresa proporciona servicios transaccionales "de pago" a sus clientes. En este caso, la empresa se vuelve parte de una red global de proveedores de valor añadido que puede utilizarse para comerciar. Como ejemplos de servicios web de integración empresarial pueden incluirse las casas virtuales de pujas y subastas, los sistemas de reserva y la comprobación de créditos.

- **Externalización de procesos de negocio**. Una empresa se diferencia a sí misma de sus competidoras a través de la creación de una cadena de valor global. En este caso, la empresa utiliza servicios Web para integrar dinámicamente sus procesos. Un ejemplo de servicios web de externalización de proceso de negocio lo constituye la asociación entre diversas compañías para combinar la producción, el ensamblaje, la distribución al por mayor y las ventas al detalle de un producto determinado.

Para una mejor comprensión de las tecnologías intervinientes en los Servicios Web, a continuación se hará una breve descripción de cada una.

5.5. Descripción de la Información: XML

Se puede pensar en las tecnologías básicas de los Servicios Web tienen su fundamento en la necesidad de invocar servicios a través de la Web. Para lograr este propósito, en primer lugar se debe contar con una sintaxis común para todas los datos a intercambiar. En el caso de Servicios Web, la sintaxis común es provista por XML. Así, todos los estándares son basados en XML, con estructuras de datos y formatos descritos como documentos XML [10].

El lenguaje ***XML*** *(Extensible Markup Language)* es un estándar aprobado y promovido por el W3C [276]. Define una sintaxis genérica usada para marcar o delimitar datos dentro de un documento, con el uso de etiquetas básicas. Provee un formato estándar para los documentos electrónicos. Este modo de formato es lo suficientemente flexible para adaptarse a diferentes dominios, como intercambio electrónico de datos, genealogía, bases de datos multimediales, serialización de objetos y servicios web [209].

El lenguaje XML es la base fundamental en la cual las tecnologías de Servicios Web se sustentan. XML provee medios para la descripción y formateo para los datos intercambiados a través de los servicios web en forma de documentos. La sintaxis del XML usada en las tecnologías de los Servicios Web especifica cómo son representados los datos, define cómo y con qué calidades de servicios es transmitido un datos, y detalla cómo deben ser publicados y descubiertos los servicios web [327, 78].

Se pueden desarrollar programas específicos que interactúan con documentos XML. La mayoría de los lenguajes de programación proveen librerías para leer y escribir XML. Existen, además una serie de herramientas que pueden abrir cualquier documento XML; o bien, son adecuadas para a un determinado formato XML específico de un dominio. Pero en cualquier caso, la misma sintaxis subyacente es utilizada, aún si los documentos XML tienen un formato amigable para un conjunto de aplicaciones genérica (como pueden ser los documentos XHTML para los navegadores web) o si tiene un formato específico como puede ser un mensaje XML consumido por un cliente de servicios web.

XML is un *metalenguaje* de demarcado para documentos de textos. Los datos son incluidos en los documentos XML como cadenas de caracteres y son rodeadas con marcas o etiquetas de texto que describen los datos. Esta unidad básica de un documento XML es llamada ***elemento***.

La idea de utilizar lenguajes de marcado para representar formatos y datos, ya tiene varios años. En particular, un lenguaje típico de demarcado es el HTML, que sirve para dar formato a las páginas web. En una manera simplificada de verlo, el demarcado de XML es similar al demarcado de HTML, ya que también contiene *elementos*, *atributos* y *valores*; pero existen diferencias cruciales. Lo más importante a destacar al respecto, es la característica de metalenguaje que tiene XML. Esto significa que no tiene un número fijo de etiqueta y elementos como lo posee HTML. Por el contrario, XML le permite a los desarrolladores definir elementos de acuerdo a las necesidades propias de cada dominio. Por ejemplo, un dominio relacionado con la Química, puede contener elementos que describan moléculas, átomos, reacciones, etc; el dominio relacionado con el negocio de propiedades puede contener elementos asociados a departamentos, casas, alquileres, etc. XML es fundamentalmente extensible, lo que significa que puede adaptarse a diferentes necesidades y dominios de software. Un dominio de software es un lenguaje de programación, un sistema de middleware, o un gestor de bases de datos.

Los programas y herramientas que interactúan con XML pueden reconocer, mapear, y transformar datos de tipos de XML en datos de tipos específicos del software. El transformar datos representados en XML en los tipos de datos propios de una aplicación, o de una representación específica de un dominio de software, es un aspecto esencial de XML. Las implementaciones de servicios web decodifican estos archivos XML para interactuar con las aplicaciones y los dominios de software subyacentes al servicio.

A pesar que XML es bastante flexible en la definición de elementos, es estricto en otros aspectos. La especificación XML provee una *gramática* que define la *sintaxis* exacta que el demarcado debe seguir: cómo los elementos son delimitados por etiquetas, cómo debe componerse

una etiqueta, cuáles nombres son aceptables para los elementos, dónde deben emplazarse los atributos, etc. Esta gramática es lo suficientemente estricta como para desarrollar reconocedores sintácticos de los documentos XML. Un documento que satisface la gramática se dice que está ***bien-formado***. Documentos que no están bien-formados no son aceptados por los analizadores sintácticos, de la misma forma que un compilador rechaza un código fuente con errores.

Por razonas de interoperatividad en los contextos de integración de aplicaciones empresariales (EAI), se establecen convenciones entre organizaciones para el uso de determinadas etiquetas y formatos de documentos XML. Este conjunto de etiquetas y su correspondiente significado, establecido por convención, se denomina *aplicación XML*. Una ***aplicación XML*** no es un sistema informático; en si, es una adaptación de XML a un dominio específico.

El demarcado en un documento XML describe su estructura; permite describir cuáles elementos están asociados con otros. En un documento XML bien-formado, el demarcado también describe la semántica del documento. Por ejemplo, el demarcado puede indicar si un elemento es una fecha, una persona o un código de barras. En una aplicación XML bien diseñada, el demarcado no especificará cómo deben presentarse o mostrarse los datos. Ya que en su esencia, XML es un lenguaje para establecer estructura y semántica de datos; no es un lenguaje para presentación.

El demarcado permitido en una aplicación particular de XML puede ser documentado en un ***esquema*** *(schema)*. Instancias particulares de documentos pueden ser contrastado con el esquema. Si un documento se corresponde con su esquema respectivo, se dice que es ***válido***. La validez del documento depende del esquema. No todos los documentos requieren ser válidos; en algunos casos, basta con exigir que esté bien-formado.

Hay muchos lenguajes diferentes para expresar esquemas de XML, con diferentes niveles de expresividad. Uno de estos, y el único definido en la misma especificación XML 1.0 [276], es el lenguaje para definición de tipos ***DTD*** *(document type definition)* [520]. Con este lenguaje se especifican definiciones de tipos de documentos, como su nombre lo indica. Un documento DTD detalla todas las marcas o etiquetas legales y especifica como deben incluirse y usarse en un documento XML. Hay que entender que las especificaciones DTDs son opcionales en XML.

Otro lenguaje, que logra mayor expresividad que los DTDs, es el lenguaje ***XML-Schema*** *(W3C XML Schema Language)* [524] estandarizado por el W3C. XML-Schema ha ganando gran popularidad y esta sustituyendo a los DTDs, en especial en el escenario de los Servicios Web. Otras opciones de lenguajes de esquemas son RELAX NG [102, 511] y Schematron [421].

Todos los lenguajes de esquemas son puramente declarativos; y son limitados a la hora de agregar controles de validación que requieran algún procesamiento. Por ejemplo, establecer una regla de validación que exija que un valor de elemento esté presente en una determinada base de datos, no se puede hacer en un esquema.

XML ofrece la tentadora posibilidad de lograr formatos inter-plataformas. Un documento XML puede ser escrito bajo un software que se ejecuta en una plataforma, e interpretado por otro software de la misma plataforma o de otra completamente distinta.

XML propone un formato muy simple, claro y bien documentado. Los documentos XML son texto y pueden ser leídos por cualquier herramienta que pueda leer un archivo de texto. No sólo los datos son texto sino que también el demarcado; es cual se presenta con el uso de etiquetas. Estas etiquetas son, entendibles por personas; eliminado los detalles oscuros de cualquier representación interna o codificación intermedia. Todos los detalles importantes acerca de la estructura del documento están explícitos; y no es necesario realizar ingeniería inversa para descubrir el formato de un documento.

Existen todavía algunas prácticas de vendedores de software que tienden a brindar formatos de datos indocumentados, propietarios y cerrados a los usuarios. En la actualidad, es mejor contar con esquema de formatos claramente documentados, bien entendidos, de fácil análisis sintáctico, basados en texto; con las mismas cualidades que provee XML. El lenguaje XML permite a los documentos y datos moverse de un sistema a otro, con buenas expectativas de

```xml
<?xml version="1.0"?>
<producto>
  <fabricante>Verbatim</fabricante>
  <nombre>DataLife MF 2HD</nombre>
  <cantidad>13</cantidad>
  <tamaño>3.5''</tamaño>
  <color>black</color>
  <descripcion>floppy disks</descripcion>
  <enStock />
</producto>
```

Código 5.1: Ejemplo básico de Documento XML

que el sistema receptor pueda interpretar y darle sentido al documento. Además, el poder que brinda la validación del documento con un esquema, provee controles en todos los puntos que tratan con el documento. Así como Java promete código portable, XML entrega datos portables. En muchos escenarios, puede considerarse a XML como el lenguaje más portable y flexible de formato de documentos desde el ASCII [209].

5.5.1. Fundamentos Básicos

La unidad de lógica y de transferencia en XML es el ***documento*** (véase Cód. 5.1). Este documento de ejemplo refleja un registro de un sistema de inventario. Demarca los datos con etiquetas y atributos, los cuales describen el color, el tamaño, número de código de barra, el fabricante, el nombre y demás datos relacionados con un producto.

Los documentos XML son construidos con texto demarcado con etiquetas que tratan de identificar el contenido entre ellas. El documento es texto y puede ser almacenado en un archivo de texto. La aplicación cliente que trata de entender los contenidos del documento XML usa un *analizador sintáctico* para leer el documento. El ***analizador sintáctico XML*** *(XML parser)* es el responsable de dividir el documento en elementos individuales, atributos y otras piezas. Se encarga de pasar al sistema cliente cada uno de los elementos del documento XML que reconoce a medida que hace el análisis, en el formato establecido por la represetación interna del cliente. Si el documento viola las restricciones que exige la calidad de bien-formado, se reporta un error, se detiene el análisis y se rechaza el documento.

Además del análisis y control de buena formación, un documento XML puede ser validado en base a un esquema. Por ejemplo se puede validar un documento contrastándolo con una especificación XML-Schema, a través de ***analizador validador*** *(validation parser)*. Un documento XML puede contener un URL indicando el lugar en dónde se encuentra su esquema. El analizador validador determinará si un documento cumple con las restricciones impuestas por su esquema. Si se produce una violación a tales restricciones se genera un error de validación; el cual no es fatal cómo el caso de los errores de buena formación. La aplicación cliente puede decidir si procesa o no el documento si no cumple con la validación. Hay que aclarar que la especificación XML determina que la cualidad de buena formación es obligatoria, en tanto que la validación es opcional.

Un documento XML es un conjunto de elementos; los cuales pueden contener atributos con valores asociados. Formalmente, un ***elemento XML*** es un contenedor consistente en: una *etiqueta de inicio*, *contenido* o *"carga útil"* (que puede ser texto, subelementos o elementos hijos, o ambos) y una *etiqueta de fin*.

Las etiquetas son las marcas que empiezan con "<" y acaban con ">"; y encierran un nombre que es sensible a mayúsculas. Este nombre se usa sólo cuando se trata de la etiqueta de inicio y precedido de "/" cuando se trata de una etiqueta de fin.

Existe una sintaxis especial para los *elementos vacíos*; es decir, aquellos elementos que no tienen contenido. Tales elementos pueden representarse con una única etiqueta vacía, que cumple

```
1  <?xml version="1.0"?>
2  <producto codbarra="2394287410">
3    <fabricante>Verbatim</fabricante>
4    <nombre>DataLife MF 2HD</nombre>
5    <cantidad>
6      <numero>13</numero>
7      <envase>Caja 10 u.</envase>
8    </cantidad>
9    <tamaño>
10     <magnitud>3.5</magitud>
11     <medida>pulgadas</medida>
12   </tamaño>
13   <color>black</color>
14   <descripcion>floppy disks</descripcion>
15   <enStock />
16 </producto>
```

Código 5.2: Ejemplo más complejo de Documento XML

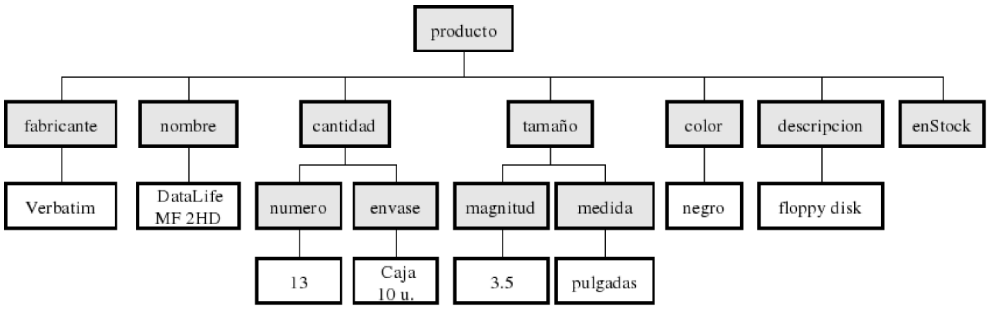

Figura 5.2: Diagrama de árbol XML.

el rol de las etiquetas de inicio y fin. Al tratarse de una etiqueta vacía, el nombre va sucedido con un `espacio` y una barra ("/"). En el ejemplo (Cód 5.1, línea 9) el elemento vacío es `<enStock />`.

En el Cód. 5.2 se observa un ejemplo más complejo de documento XML. En este caso hay varios elementos cuyo contenido son otros elementos: por ejemplo el elemento `tamaño` está formado por dos subelementos `magnitud` y `medida`. De esta forma, se dice que el elemento `tamaño` tiene dos elementos hijos *(child elements)* o subelementos. Si se establece las relaciones jerárquicas entre elementos, se podrá definir un *árbol* a partir del documento XML (véase Fig. 5.2). En este árbol XML existe un ***elemento raíz*** *(root element)*; que en el ejemplo es `producto`. Asi también podemos definir ***elementos hermanos*** *(siblings elements)*; como es el caso de `nombre`, `cantidad`, `tamaño`, `color`, `descripcion` y `enStock`. Un documento XML bien formado tiene un único elemento raíz.

Los elementos XML pueden tener *atributos*. Un ***atributo*** es un par nombre-valor que se ubica en la etiqueta de inicio de un elemento. Los nombres de atributos se separan de los valores asociados con un signo igual. Los valores van encerrados entre comillas dobles. Para el ejemplo, en el Cód. 5.2, el elemento `producto` tiene un atributo `codbarra="2394287410"`.

Surge la pregunta de cuándo utilizar elementos hijos o atributos para contener información de un elemento. Esta es una pregunta sin respuesta categórica. Por un lado, algunos consideran que los atributos están destinados a metadatos acerca de los elementos; en tanto que los elementos son la información en sí. Por ejemplo un atributo de un elemento que represente una fecha, sería el formato en el que se expresa la misma, o el carácter utilizado como delimitador; por ejemplo: `<fechaCarga formato="d/m/Y">23/12/2002</fechaCarga>`. En otros casos, no está clara la distinción entre qué es metadato y qué es dato.

No existe restricciones en cuántos atributos puede llevar un elemento. Pero si, están limitados

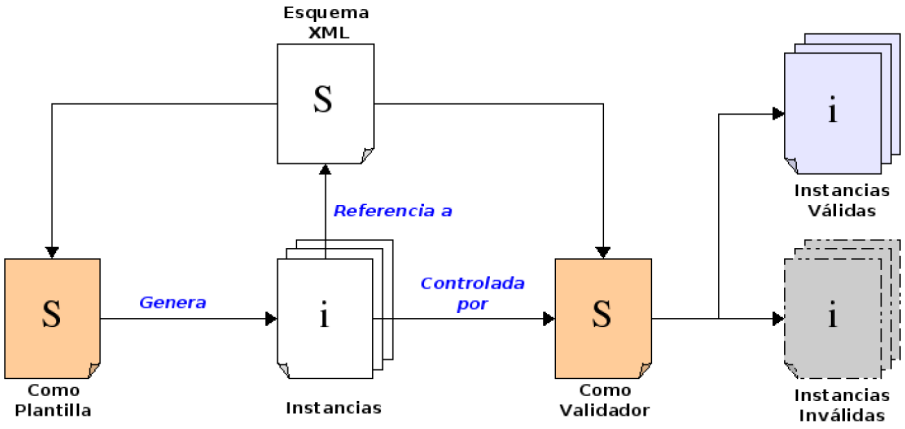

Figura 5.3: Esquemas e Instancias.

a ser solamente una cadena de caracteres, no pueden ser estructuras. Para los casos en que sea necesario categorizar a un elemento con más detalle, puede ser necesario utilizar subelementos para tal fin.

Los nombres de elementos y atributos de XML son sensibles a mayúsculas y minúsculas y pueden contener cualquier carácter alfabético, letras latinas y con signos diacríticos como ö, ç, ñ, etc. Se acepta sólo tres signos de puntuación: el carácter subrayado "_" , el guión "-" y el punto ".". Los nombres de XML no pueden tener otros signos de puntuación, ni tampoco espacios de ningún tipo. Los nombres no pueden comenzar con la cadena de caracteres "XML", o cualquiera de combinación de mayúsculas y minúsculas, ya que el W3C reserva tales nombres para especificaciones relacionadas con el lenguaje XML. Los nombres XML sólo pueden comenzar con letras y el carácter subrayado; no pueden comenzar con dígitos, guión o punto. No existen límites en la longitud de un nombre.

5.5.2. XML Schema

XML Schema es un lenguaje de definición que permite obligar que un determinado documento XML esté conforme a un vocabulario y una estructura jerárquica específicos [115]. XML Schema es un estándar recomendado por el W3C [524].

En un lenguaje de definición de esquemas, se especifica cómo será una aplicación de XML. En una especificación de esquema XML se define los tipos de los elementos y de los atributos, y la composición de ambos en tipos compuestos. Como se muestra en la Fig. 5.3, existen dos tipos de documentos: un documento que representa el esquema (o documento de definición); y los múltiples documentos instancias que están conforme con el esquema. Una buena analogía sería considerar a los documentos XML Schema como *plantillas* de un tipo de documentos XML, y cada *instancia* es una materialización de esa plantilla. En la figura también se pueden observar los roles que juega un esquema:

- Como plantilla para un generador de instancias de un tipo de documento XML
- Como instrumento de validación de concordancia de un documento con el esquema

En ambos casos, el documento esquema y el documento instancia usa sintaxis XML. Esta cualidad es un factor motivador para reemplazar a los documentos DTDs, los cuales tenían sintaxis propia.

Se puede pensar en los esquemas XML como en los esquemas de bases de datos, en los que se define el tipo de las columnas y el tipo de tablas. De la misma forma, cada registro de tabla es una instancia del esquema. Cada instancia, al tratarse de documentos XML debe declarar a cuál esquema adhiere; o mejor dicho, de cuál esquema es instancia.

```
<xsd:element name="producto">
  <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="descripcion" type="xsd:string"
           minOccurs="0" maxOccurs = "1" />
       <xsd:element name="categoria" type="xsd:string"
           minOccurs = "1" maxOccurs = "unbounded" />
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID" />
     <xsd:attribute name="titulo" type="xsd:string" />
     <xsd:attribute name="precio" type="xsd:decimal" />
  </xsd:complexType>
</xsd:element>
```

Código 5.3: Definiciones de esquemas de elementos complejos.

Los esquemas XML permiten la validación de instancia para asegurarse la correctitud de los valores y la estructura de un documento. La correctitud de los valores es controlada en base al tipo de dato definido para el elemento. La estructura del documento es validada con la legalidad de los nombres de elementos y atributos de acuerdo a los especificado en el esquema, al correcto número de subelementos, a la existencia de los atributos requeridos, etc. Todos los documentos XML deberían validarse antes de ser transferido a otros sistemas u organizaciones.

Un documento XML Schema usa la sintaxis de XML para describir un conjunto de declaraciones de tipos simples y compuestos. Un nombre de un tipo es una plantilla nombrada que puede contener uno o más valores. Los valores simples contiene un sólo valor. Un tipo complejo esta compuesto por múltiples tipos simples. De esta forma, un tipo tiene dos características: un nombre y un conjunto de valores legales.

Un tipo simple es una declaración de elemento que incluye su nombre y sus limitaciones de valor. Por ejemplo, un elemento llamado `autor`, cuyo valor pueden ser cualquier cadena de caracteres se especificaría en un esquema de esta forma:

```
<xsd:element name="autor" type="xsd:string" />
```

Esta declaración validaría correctamente a un documento instancia que presente un elemento similar a:

```
<autor>Antoine Saint Exupéry</autor>
```

Un tipo complejo es un elemento que contiene a otros elementos o que tienen atributos adjuntos. Por ejemplo, en el código siguiente se define un elemento llamado `libro` con dos atributos denominados `titulo` y `paginas`:

```
<xsd:element name="libro">
    <xsd:complexType>
      <xsd:attribute name="titulo" type="xsd:string" />
      <xsd:attribute name="paginas" type = "xsd:int" />
    </xsd:complexType>
</xsd:element>
```

Una instancia de esta definición de elementos podría ser:

```
<libro titulo = "El Principito" paginas="253" />
```

Como ejemplo de la definición de un elemento con atributos y elementos hijos, se muestra el Cód. 5.3 definiendo el elemento `producto`. Este elemento tiene tres atributos: `id`, `título` y `precio`. También tiene un elemento hijo llamado `categoria` que es exigido y repetible; y un elemento hijo llamado `descripción` que es opcional. Instancia de este esquema de elemento sería la presentada en el Cód. 5.4.

```
<producto id="P01" titulo="El Principito" precio="9.99">
    <descripcion>
    En apariencia un libro infantil, en él se tratan temas
    tan profundos como el sentido de la vida, la amistad y el amor.
    </descripcion>
    <categoria>infantil</categoria>
    <categoria>ficción</categoria>
</product>
```

Código 5.4: Instancias de definiciones de esquemas de elementos complejos.

5.5.3. Espacios de Nombre

Los ***espacios de nombres*** *(namespaces)* son un mecanismo simple para crear nombres únicos para los elementos y atributos para los documentos hechos XML. Es importante el uso de espacios de nombre por dos razones fundamentales: sirven para no generar conflictos, o *colisiones*, en el caso de que existan idénticos nombres para diferentes aplicaciones de XML; y, gracias a ésto, permite que dos o más aplicaciones XML se usen conjuntamente. En este caso, los espacios de nombres hace que no se generen ambigüedades al referenciar un elemento o atributo [115].

Los espacios de nombres no son compatibles con las definiciones DTD, por esta razón su adopción fue lenta. Sin embargo, los lenguajes de esquemas actuales, como XML Schema, tiene completo soporte de espacios de nombre.

Para su adopción, los espacios de nombres requieren que cada nombre XML consista de dos partes: un ***prefijo*** (el espacio de nombres en sí) y una ***parte local*** (en nombre propiamente dicho). Un ejemplo de un nombre completamente calificado sería:

```
<xsd:element>
```

La parte local es el identificador para el metadato, para el caso es `element`. Y el prefijo `xsd` es la abreviación del espacio de nombres que se utilizó en la declaración de espacio de nombres del documento. El espacio de nombres es realmente un URI (véase Sec 4.1.2). La declaración del esquema en cuestión sería:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

El ejemplo declara un espacio de nombres para todos los elementos de XML Schema a ser utilizados en el documento esquema XML (véase códigos Sec. 5.5.2). Se define el prefijo `xsd` para representar el espacio de nombres `http://www.w3.org/2001/XMLSchema`. Es importante aclarar que el prefijo no es el espacio de nombres, es sólo una abreviatura que puede cambiar de documento en documento.

Hay dos formas de aplicar un espacio de nombres a un documento: adjuntar el prefijo asociado al espacio de nombres a cada uno de los elementos; o declarar un espacio de nombres que tomará el documento por defecto. Un espacio de nombres a considerar por defecto se declara omitiendo el prefijo, como lo muestra en el siguiente Cód.:

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>La página del Principito</title></head>
   <body>Cuando yo tenía seis años vi en un libro sobre la selva virgen...</body>
</html>
```

5.5.4. XML y los Servicios Web

XML es generalmente representado por un conjunto de especificaciones publicadas en el W3C (World Wide Web Consortium). Muchas aplicaciones incorporan una o más de estas especificaciones. Dos aspectos de XML son de primordial importancia para los Servicios Web: el formato

de representación de datos (o las instancias de datos), y la posibilidad de asociar esquemas para definir la semántica de los datos.

XML es aplicado en una amplia variedad de escenarios. Su principal uso en el contexto de los Servicios Web es el formateo, la serialización (o linealización) y la transformación de datos [327]. Los servicios web se comunican mediante el intercambio de instancias formateadas (validadas) de documentos XML que transportan datos. Los servicios web son descriptos usando esquemas XML que definen los tipos de datos, estructuras y semánticas.

La principal ventaja que ofrece XML a los Servicios Web es la independencia de datos; lo que logra que los tipos y estructuras de datos no estén atados a la implementación subyacente del proveedor de servicio. Para ser posible tal independencia, se debe contar con herramientas que permitan la transformación de datos entre XML y los formatos nativos de la plataforma.

5.6. Acceso a los Servicios Web: SOAP

Como siguiente paso para lograr conectar a dos servicios a través de la Web, debe existir un mecanismo que permita a los sitios remotos interactuar entre sí. La especificación de este mecanismo debe describir tres aspectos: un formato común para los mensajes a intercambiar, una convención para soportar formas de interacción (basada en mensajes o RPCs), y un conjunto de convenciones para mapear mensajes a protocolos subyacentes de transporte o aplicación.

El mecanismo de interacción debe mantener a las aplicaciones distribuidas con bajo grado de acoplamiento. Por esta razón, el mecanismo depende del mensaje como unidad básica de comunicación, en lugar de las llamadas a procedimientos. Sin embargo, es posible que los mecanismos de interacción de los Servicios Web soporten (o simulen soportar) también las interacciones en la forma de RPCs.

En Servicios Web las interacciones se basan en el protocolo ***SOAP*** *(Simple Object Access Protocol)*. Usando SOAP, los servicios web pueden intercambiar mensajes por medio de convenciones estandarizadas para convertir una invocación de servicio en un mensaje XML, para intercambiar el mensaje entre aplicaciones, y para convertir el mensajes de XML en una invocación real al servicio [10].

El principal uso de SOAP es la comunicación Aplicación-a-Aplicación A2A (Application-to-Application). Es usado especialmente en la Integración de Aplicaciones Empresariales y en los escenarios B2B (Business-to-Business), los cuales son dos caras de la misma moneda y están fuertemente enfocados a la integración de aplicaciones e intercambio de información.

SOAP, para que sea verdaderamente efectivo en proyectos de B2B y EAI, debe ser independiente de la plataforma; flexible y basado en estándares. A diferencia de las tecnologías anteriores de B2B y de EAI, como CORBA y EDI, SOAP cumple plenamente con tales requerimientos; cuenta con una aceptación y amplio uso, y ha sido patrocinado por los principales vendedores de software empresarial y las organizaciones de estándares más destacadas (W3C, WS-I, OASIS, etc.) [315].

En el Capítulo 4 se discutió las dificultades de integrar aplicaciones a través de la Web: la existencia de cortafuegos, la falta de protocolos estandarizados, la necesidad de interacciones de bajo acoplamiento, etc. El protocolo SOAP es la respuesta que la tecnología de los Servicios Web proponen a estas problemáticas. SOAP es el protocolo subyacente a todas las interacciones en el ámbito de los Servicios Web; y como tal, define cómo organizar la información usando XML en una manera estructurada y basada en tipos, de tal forma que se pueda establecer el intercambio entre dos aplicaciones pares.

En particular, SOAP especifica:

- Un formato de mensaje y modo de comunicación *en-un-sentido*, describiendo cómo la información debe ser empaquetada dentro de un documento XML. En este sentido, para definir los tipos de documentos válidos, SOAP utiliza XML Schema.

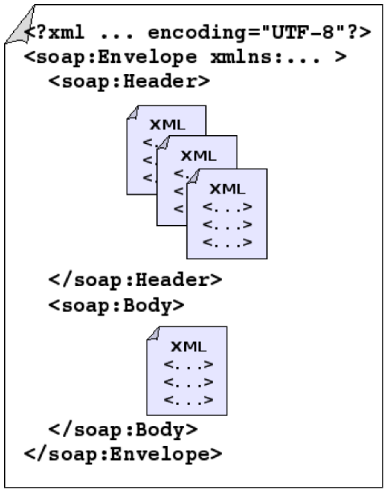

Figura 5.4: Estructura de un documento SOAP.

- Un conjunto de convenciones para usar SOAP para implementar el patrón de interacciones RPC, definiendo cómo los clientes pueden invocar una llamada a procedimiento remoto por medio del envío de mensajes SOAP y cómo los servicios deben replicar con otro mensaje SOAP a la aplicación cliente.

- Un conjunto de reglas que debe seguir cualquier entidad que procese un mensaje SOAP. En estas reglas se define los elementos que debe leer y entender las entidades; y también, las acciones a seguir en el caso de que no se entienda el contenido del mensaje.

- Una descripción de *ligaduras SOAP*. Una **ligadura SOAP (SOAP binding)** [264] especifica cómo un mensaje SOAP puede ser intercambiado usando protocolos subyacentes de transporte de red u otros protocolos de aplicación. Al respecto, la especificación SOAP detalla la ligadura con HTTP y brinda ejemplos de ligaduras con protocolos de correo electrónico, especialmente SMTP.

SOAP es fundamentalmente un protocolo sin estado y de mensajes de intercambio *en-un-sentido (one-way messages)*. También ignora la semántica de los mensajes que se intercambian a través de éste. La verdadera interacción entre dos partes debe ser incrustada dentro de un documento SOAP, y cualquier patrón de interacción complejo (por ejemplo: petición/respuesta, broadcasting, etc.) debe ser implementado por los sistemas subyacentes. Esto significa que SOAP ha sido creado para soportar aplicaciones de bajo acoplamiento que interactúan entre si mediante el intercambio asincrónico de mensajes en una dirección. La especificación SOAP trata este tipo de intercambio; describe cómo deben escribirse los documentos que llevan los mensajes y como debe organizarse el dialogo en las comunicaciones más complejas como RPCs.

5.6.1. Estructura y Contenido de un mensaje SOAP

Como se sabe, un mensaje SOAP es una clase de *documento XML*. SOAP tiene su propio esquema XML, espacios de nombre y reglas de procesamiento [315].

Un mensaje SOAP es análogo a un *sobre de carta* o *envoltura (envelope)* usada en el correo postal. De la misma forma que un sobre contiene una carta en papel, un mensaje SOAP contiene datos en XML. Por ejemplo, un mensaje SOAP que encierra una orden de compra (*purchaseOrder*) se muestra en el Cód. 5.5. En el ejemplo, se ve que los espacios de nombres de XML son usados para mantener a los elementos específicos de SOAP (en negritas) separados de los propios de la orden de compra. En el ejemplo, se mantienen los nombres originales basados en inglés.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <soap:Envelope
3   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
4    <soap:Body>
5      <po:purchaseOrder orderDate="2003-09-22"
6       xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO">
7        <po:accountName>Amazon.com</po:accountName>
8        <po:accountNumber>923</po:accountNumber>
9        <po:address>
10         <po:name>AMAZON.COM</po:name>
11         <po:street>1850 Mercer Drive</po:street>
12         <po:city>Lexington</po:city>
13         <po:state>KY</po:state>
14         <po:zip>40511</po:zip>
15       </po:address>
16       <po:book>
17         <po:title>J2EE Web Services</po:title>
18         <po:quantity>300</po:quantity>
19         <po:wholesale-price>24.99</po:wholesale-price>
20       </po:book>
21       <po:book>
22         <po:title>XML Bible</po:title>
23         <po:quantity>120</po:quantity>
24         <po:wholesale-price>34.50</po:wholesale-price>
25       </po:book>
26     </po:purchaseOrder>
27   </soap:Body>
28 </soap:Envelope>
```

Código 5.5: Ejemplo de un mensaje SOAP.

El Cód. 5.5 es un ejemplo de un mensaje SOAP que contiene un elemento arbitrario XML, el elemento `purchaseOrder` (Orden de Compra). En este caso, el mensaje SOAP es en-un-sentido y será enviado desde el emisor al receptor último sin esperar respuesta alguna.

Los clientes de la editorial `Monson-Haefel Books` usarán este mensaje SOAP para solicitar la compra de libros, indicando los datos del cliente (elementos `po:accountName`, `po:accountNumber`), la dirección de envío (elemento `po:address`) y los datos de los libros a comprar (dos elementos `po:book`).

Un mensaje SOAP puede tener una declaración XML, la cual establece la versión de XML que es usada y el formato de codificación, como se muestra en la línea 1 del Cod. 5.5. La declaración XML no es obligatoria: en caso de no existir se toman como valores por defecto XML 1.0 y UTF-8 respectivamente.

Como se explicó en la sección anterior (Sec. 5.5.1), cada documento XML tiene un elemento raíz, y en los documentos SOAP el elemento raíz es **`Envelope`**. Este elemento puede contener un elemento de encabezamiento de mensaje denominado **`Header`**, que es opcional, y obligatoriamente debe contener un cuerpo de mensaje representado por el elememento **`Body`**. Si se usa el elemento de encabezado, éste debe ser un elemento hijo del elemento raíz `Envelope`, y debe preceder al elemento `Body`. El elemento `Body` contiene los datos a intercambiar entre las aplicaciones interactuantes, específicos de aplicación. En otras palabras, el cuerpo del mensaje SOAP contiene la *carga útil del mensaje*, espresada en XML.

Un mensaje SOAP adhiere a (o es instancia de) el ***esquema SOAP XML***, el cual requiere que los elementos y atributos sean completamente calificados: es decir, lleven prefijo o espacios de nombres por defecto.

SOAP es extremadamente flexible, debido a que no limita el tipo de datos XML que pueden ser incrustados en el cuerpo de un mensaje. Los datos del cuerpo del mensaje SOAP pueden ser de diferente índole, como ser los relacionados a una orden de compra, o a aquellos que sirven

para mapear los argumentos de una llamada a procedimiento.

El elemento de encabezado `Header` contiene información acerca del mensaje, en la forma de uno o más elementos XML distintos; cada uno describe algunos aspectos o calidad de servicio asociado con el mensaje. Cada elemento hijo del elementos de encabezado se lo denomina **bloque de encabezado (header block)**.

El elemento de encabezado puede contener bloques que describen credenciales de seguridad, IDs de transacción, instrucciones de enrutado de mensaje, información de depuración o cualquier otra información acerca de los mensajes que se considere importante para procesar los datos contenidos en el cuerpo. Por ejemplo, se puede asociar cada mensaje a un identificador único y a una estampilla de tiempo, de forma tal que sirva para autenticación y sincronización de mensajes (véase Cód. 5.6, líneas 8 y 9). De la misma forma, un bloque de encabezado puede tener cualquier tamaño y ser tan complejo como se quiera. Por ejemplo en el Cód. 5.6 se especifica la firma digital en el encabezado (Líneas 10 a 31).

Los espacios de nombres en los mensajes SOAP juegan un rol importante. Para evitar la colisión de nombres, cada uno de los elementos del mensaje pertenece a un único espacio de nombres y tienen un único prefijo. El uso de los espacio de nombres XML hacen de SOAP un protocolo flexible y extensible. Como se explico en la Sec. 5.5.3 un espacio de nombres califica completamente a un elemento o atributo.

En el ejemplo del Cód. 5.6 se presentan cinco espacios de nombres, asociados con los elementos propios de la envoltura SOAP (línea 3), de la orden de compra (línea 33 y 34), de la identificación de mensaje (línea 6) y de la firma digital (líneas 4 y 5).

El primer atributo (Línea 3) declarado en el elemento `Envelope`, define el espacio de nombres para los elementos comunes de un mensaje SOAP: `Envelope`, `Header`, y `Body`. Se utiliza el prefijo `SOAP` para referencial al espacio de nombres `"http://schemas.xmlsoap.org/soap/envelope/"`.

El segundo, tercero y cuarto espacio de nombres (Líneas 4 a 6) declarados en el elemento `Envelope` están asociados a los bloques de encabezado. En este caso en particular, se puede observar el uso de una variedad de espacios de nombre estándares que cubren las necesidades relacionadas con seguridad, transacciones, y otras cualidades de servicios. Tales estándares de espacio de nombres fueron desarrollados por organizaciones de estandarización como ser W3C, OASIS, y IETF; o por los principales vendedores de software de distribución como Microsoft, BEA, e IBM.

El último espacio de nombres fue declarado dentro del cuerpo del mensaje (Líneas 33 y 34), en el elemento `po:purchaseOrder`. Este espacio de nombres pertenece a la editorial `Monson-Haefel Books` que, cómo organización, define el nombre calificado de cada uno de los elementos del formulario de su propia orden de compra.

Hay que tener en cuenta que el poder real de los espacio de nombres de XML va más allá que la tarea de eliminar ambigüedades y evitar colisiones de nombres. Los espacios de nombres sirven para lograr un adecuado manejo de versiones y procesamiento del mensaje. El hecho de usar nombres calificados completos para el mensaje SOAP y los datos específicos de la aplicación, le indica al receptor con cuáles esquemas se debe validar el contenido y cómo debe procesar el mensaje. Cada espacio de nombres puede estar asociado a un módulo de software que trate los correspondientes datos. De esta forma, los espacios de nombres habilita al receptor del mensaje a tener distintas versiones de módulos que traten distintas versiones de espacios de nombres. Este hecho es esencialmente útil en el contexto de Servicios Web, ya que los consumidores y productores pueden preservar compatibilidad con todas las versiones diferentes de un mismo tipo de mensajes SOAP.

5.6.2. Ruta de Mensajes SOAP

La especificación SOAP define reglas para el procesamiento de los bloques de encabezados a lo largo de la *ruta del mensaje* SOAP. Una ***ruta de mensaje SOAP*** es la secuencia de nodos que recorre un mensaje SOAP desde la aplicación remitente hasta la aplicación destinataria. Las

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
 xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <mi:time-stamp>2005-10-30 T 10:45 UTC</mi:time-stamp>
    <sec:Signature>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm=
           "http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
          <ds:SignatureMethod Algorithm=
           "http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
          <ds:Reference URI="#Body">
            <ds:Transforms>
              <ds:Transform Algorithm=
               "http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm=
             "http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>u29dj93nnfksu937w93u8sjd9=
            </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>CFFOMFCtVLrklR...</ds:SignatureValue>
      </ds:Signature>
    </sec:Signature>
  </soap:Header>
  <soap:Body sec:id="Body">
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.Monson-Haefel.com/jwsbook/PO">
      <po:accountName>Amazon.com</po:accountName>
      <po:accountNumber>923</po:accountNumber>
      <po:address>
        <po:name>AMAZON.COM</po:name>
        <po:street>1850 Mercer Drive</po:street>
        <po:city>Lexington</po:city>
        <po:state>KY</po:state>
        <po:zip>40511</po:zip>
      </po:address>
      <po:book>
        <po:title>J2EE Web Services</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
      </po:book>
  </soap:Body>
</soap:Envelope>
```

Código 5.6: Estructura básica de un mensaje SOAP.

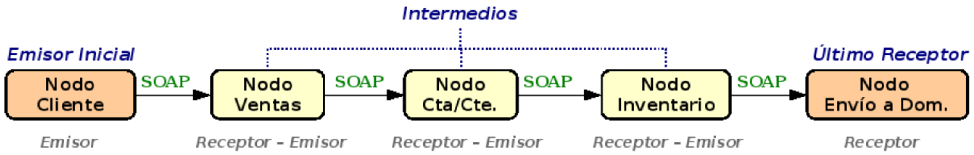

Figura 5.5: Ruta de mensaje SOAP relacionado con una orden de compra.

reglas SOAP especifican cómo deben ser tratados los bloques de encabezados por determinados nodos en la ruta del mensaje; y también, define qué se deben hacer con el encabezado una vez que se procesa en cada nodo.

Un mensaje SOAP viaja a través de una ruta de mensaje desde el remitente o ***emisor inicial*** hasta el destinatario o ***último receptor***. Desde el punto de vista del protocolo, el emisor inicial es la aplicación que genera el mensaje SOAP, en tanto que el receptor último es la última aplicación de la ruta que recibirá en mensaje. En el contexto de Servicios Web, el emisor inicial es la aplicación cliente consumidora de servicio; y el receptor último es la aplicación proveedora de servicios.

Conforme un mensaje SOAP viaja a través de su ruta, su encabezamiento puede ser tratado por cualquier número de nodos *intermediarios SOAP*. Un ***intermediario SOAP*** cumple con los dos roles de receptor y emisor. Cada intermediario puede recibir un mensaje SOAP, procesar uno o más bloques de encabezado y reenviar el mensaje a otra aplicación SOAP (véase Fig. 5.5).

Para ilustrar la forma en que los nodos de una ruta procesan los bloques de encabezado, se ejemplifica en base al ejemplo tratado en la sección anterior relacionado con la orden de compra de libros. En este caso, el mensaje SOAP de orden de compra pasa diversos nodos intermedios antes de alcanzar el receptor último. La Fig. 5.5 muestra que la ruta de un mensaje SOAP de una orden de compra recorre los nodos de que contienen los sistemas de ventas (sales), cta/cte. (AR), inventario (inventory) y envío a domicilio (shipping).

Los nodos intermedios de una ruta de mensaje SOAP no deben modificar los contenidos del cuerpo del mensajes relacionados con la aplicación específica. En el ejemplo, los datos relacionados con la orden de compra serían de aplicación específica.

Para ilustrar la forma en que los nodos de una ruta procesan los bloques de encabezado, se ejemplifica un mensaje con dos bloques de encabezado: `message-id` y `processed-by`. El bloque `processed-by` mantiene un registro de los nodos SOAP que procesaron el mensaje desde el emisor inicial hasta el receptor último. Al igual que `message-id`, `processed-by` contienen información útil para la autenticación y depuración.

En el ejemplo del Cód. 5.7, cada intermediario tiene la obligación de agregar un elemento `node` al bloque de encabezado `processed-by`. En el elemento agregado `node`, se registra la URI del intermediario y el momento en que se proceso el mensaje en dicho intermediario.

Cuando se procesa un bloque de mensaje, cada nodo puede leer, modificar o eliminar un bloque de encabezado antes de reenviar el mensaje al siguiente nodo receptor. Sin embargo, en ocasiones en necesario indicar cuál bloque debe ser tratado por un determinado nodo en particular. En este sentido, las aplicaciones SOAP utilizan el atributo **`role`** para identificar los nodos que deberían procesar bloques específicos.

Como ejemplo, se puede pensar en un rol de `auditor (logger)`. En la situación hipotética, los nodos que cumplen con este rol mantiene una bitácora que registra la información de los mensajes que pasan por estos. Los nodos que implementan este rol tendrán un módulo de software que lleve el registro en medio de almacenamiento. Varios nodos pueden identificarse con este rol; de la misma forma que varios roles pueden ser cumplidos por un mismo nodo. En tales casos, el nodo contendrá varios módulos asociados a cada uno de los roles que cumple. Por ejemplo, el rol de auditor puede ser cumplido por el nodo de Ventas y el de Cta.Cte.; y a su vez, el nodo Venta se puede contener módulos para atender a roles relacionados con validación de usuario, seguridad, transacciones, etc.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <proc:processed-by>
      <node>
        <time-in-millis>1013694680000</time-in-millis>
        <identity>http://www.customer.com</identity>
      </node>
      <node>
        <time-in-millis>1013694680010</time-in-millis>
        <identity>http://www.Monson-Haefel.com/sales</identity>
      </node>
      <node>
        <time-in-millis>1013694680020</time-in-millis>
        <identity>http://www.Monson-Haefel.com/AR</identity>
      </node>
      <node>
        <time-in-millis>1013694680030</time-in-millis>
        <identity>http://www.Monson-Haefel.com/inventory</identity>
      </node>
      <node>
        <time-in-millis>1013694680040</time-in-millis>
        <identity>http://www.Monson-Haefel.com/shipping</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
     <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Código 5.7: El bloque de encabezado processed-by.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.Monson-Haefel.com/jwsbook/message-id"
 xmlns:proc="http://www.Monson-Haefel.com/jwsbook/processed-by">
  <soap:Header>
    <mi:message-id soap:role="http://www.Monson-Haefel.com/logger" >
      11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <proc:processed-by>
      <node>
        <time-in-millis>1013694680000</time-in-millis>
        <identity>http://www.customer.com</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
     <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

Código 5.8: El atributo `role`.

El atributo **`role`** usa un URI para identificar el rol que un nodo debe cumplir para procesar un bloque de encabezado. El atributo rol es usado en combinación con los espacios de nombres XML para determinar cuáles módulos procesarán un bloque de encabezado particular. Conceptualmente, un nodo al recibir un mensaje, examinará cada bloque de encabezado y determinará si se corresponde al rol definido en el atributo `role`. Si es así, elegirá el módulo de software correcto basándose en el espacio de nombres del bloque.

Por ejemplo, el atributo `role` identifica el rol de auditor *(logger)* con la URL `http://www.Monson-Haefel.com/logger`, y en el Cód 5.8 se muestra que dicho rol está asociado al bloque de encabezado `message-id`. Sólo los nodos, en la ruta de mensaje, que se identifican con ese rol procesarán el bloque de encabezado `message-id`, en tanto que los otros lo ignorarán.

Además de los roles propios que se pueden encontrar en la aplicación distribuida de la editorial `Monson-Haefel Books`, como el rol de auditor; SOAP identifica tres roles estándares: `none`, `next` y `ultimate receiver`.

Si un bloque es asignado con el rol **`none`**, significa que dicho bloque no debe ser procesado por ningún nodo en la ruta del mensaje. Este rol es usado comúnmente para indicar que información de nodo puede ser leída y sirve para el procesamiento de otros nodos.

El rol **`next`** indica que el siguiente nodo de la ruta del mensaje debe procesar el bloque de encabezado en el cual se encuentra el atributo role con la URI `http://schemas.xmlsoap.org/soap/actor/next` como valor. Virtualmente, todos los nodos de la ruta del mensaje cumplen con este rol.

Al rol **`ultimate receiver`**, lo cumple solamente el nodo último receptor de la ruta. Este rol no tiene un URI explícito, por considerar que con la ausencia del atributo `role` en cualquier bloque de encabezado, el bloque está orientado último receptor.

El uso de estos roles estándares tiene interesantes implicancias en el tratamiento de los mensajes a lo largo de la ruta, especialmente `next`. En los casos de los roles específicos de la aplicación distribuida (como *logger*), se supone que existe en la ruta un nodo que cumpla con ese rol y, como tal, tenga un módulo que procese el bloque de encabezado con ese atrubuto. Sin embargo, al declarar por ejemplo el rol `next` para que se procese un bloque de encabezado `proc:processed-by`, implica que cada uno de los nodos de la ruta deban procesar el bloque, utilizando algun módulo de software para tal caso. Puede darse que algunos de estos nodos no tengan los módulos adecuados definidos para tratar a estos bloques; en este caso, el nodo puede ignorar el bloque y

```
1 package com.jwsbook.soap;
2 import java.rmi.RemoteException;
3
4 public interface BookQuote extends java.rmi.Remote {
5    // Get the wholesale price of a book
6    public float getBookPrice(String ISBN)
7      throws RemoteException, InvalidISBNException;
8 }
```

Código 5.9: Interfaz remota del servicios `BookQuote`.

reenviar el mensaje al siguiente nodo de la cadena. Para forzar el tratamiento de un bloque de encabezado en un nodo por medio de un módulo correspondiente, se puede agregar el atributo **`mustUnderstand`** en dicho bloque de encabezado. El atributo **`mustUnderstand`** con valor `1`, exige al nodo que cumpla con el rol al procesar de alguna forma el bloque, si no se cuenta con el modulo adecuado se generará un error.

5.6.3. Modos de Comunicación SOAP

Un modo de mensaje es definido por su *estilo de intercambio de mensajes* (RPC o orientado a mensajes) y por su *estilo de codificación*. De esta forma, SOAP soporta cuatro modos de mensajes: *RPC/Literal*, *Documento/Literal*, *RPC/Codificado* y *Documento/Codificado*.

Documento/Literal

En el modo ***Documento/Literal*** de mensaje, el cuerpo del mensaje SOAP contiene un *fragmento de documento XML*; un elemento XML bien formado que contiene una cantidad arbitraria de información específica de la aplicación y que se corresponde con un esquema XML y un espacio de nombres distinto al utilizado para el mensaje SOAP.

Por ejemplo, en el Cód 5.6 los elementos XML incrustados en mensaje SOAP que describen la orden de compra (líneas 33 a 48), son considerados un fragmento de documento XML.

RPC/Literal

En el modo ***RPC/Literal***, se estructuran los mensajes SOAP para modelar llamadas a procedimientos o invocaciones a métodos remotos y su correspondientes retornos de resultados. En el modelo RPC/Literal, el contenido del elemento `Body` se formatea como una estructura. Un *mensaje de petición* de RPC contiene el nombre del método y los parámetros de entrada para la llamada. El *mensaje de respuesta* de RPC contiene el nombre del método, el resultado y cualquier otro parámetro de salida o falla.

Es común, que el modo RPC/Literal sea usado para exponer a componentes de software tradicionales como servicios web. Se podría considerar componente "tradicional" a un servlet, a un bean de sesión sin estado, a objetos RMI, CORBA o DCOM, etc. Estos componentes no están construidos específicamente para intercambiar datos XML; en general, todos tienen métodos con parámetros formales y valores de retorno.

Como ejemplo, la editorial `Monson-Haefel Books` tiene un servicio web desarrollado en JAX-RPC denominado `BookQuote` (Cotización de Libro) que, como es obvio, la oficina de ventas utiliza en forma recurrente. La interfaz remota de tal servicio puede ser como lo descrito en Cód 5.9.

El método `getBookPrice()` declara un único parámetro en la forma de ISBN. El ISBN es una cadena de caracteres asignada unívocamente a cada libro que se vende. Cuando una aplicación cliente, o consumidor de servicio, invoca este método con un ISBN adecuado, el servicio web contestará con el precio de venta del libro con el ISBN correspondiente.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
  <soap:Body>
    <mh:getBookPrice>
       <isbn>0321146182</isbn>
    </mh:getBookPrice>
  </soap:Body>
</soap:Envelope>
```

Código 5.10: Un mensaje de petición en el modo RPC/Literal

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote" >
  <soap:Body>
    <mh:getBookPriceResponse>
       <result>24.99</result>
    </mh:getBookPriceResponse>
  </soap:Body>
</soap:Envelope>
```

Código 5.11: Un mensaje de respuesta en el modo RPC/Literal

Esta aplicación JAX-RPC proveedora de servicio, puede utilizar el modelo RPC/Literal para los mensajes. El servicio web usará dos mensajes, uno de petición y otro de respuesta. El mensaje de petición es generado y enviado por el emisor inicial al servicio web (último receptor); el mensaje contiene el nombre del método (`getBookPrice`) y el valor del parámetro `ISBN` (véase Cód. 5.10). El mensaje de respuesta es enviado al emisor inicial y contiene el resultado de la operación, el precio del libro, como un número real de punto flotante (véase Cód. 5.11).

A diferencia del modo Documento/Literal, el cual no realiza ninguna suposición acerca del tipo y estructura de elementos contenidos en el cuerpo del mensaje, el modelo RPC/Literal transporta un conjunto básico de argumentos. Como se vio en el capítulo 3, en el estilo RPC es un idioma común para las tecnologías de middleware distribuido, como por ejemplo CORBA, DCOM, EJB, DCE y otras. Como es de suponer, SOAP define un formato estándar XML para que se soporte el modelo de llamadas a procedimiento remoto través de los mensajes SOAP. La especificación del modelo *RPC/Literal* define cómo los métodos y sus argumentos (parámetros y valores de retorno) son representados en un cuerpo de mensaje, dentro del elemento `Body`.

Al hablarse de *modos de mensajes* y de *patrones de comunicación* (o formas de interacción) no se deben confundir a los modos de mensajes Documento/Literal y RPC/Literal con los patrones de interacción Orientados a Mensajes en-un-sentido o de petición/respuesta.

El *modo de mensaje* tiene relación con lo que se transporta en el cuerpo de un mensaje SOAP, su carga útil. En el modo Documento/Literal se lleva un fragmento de un documento XML, en tanto que el el modo RPC/Literal se lleva una representación específica para los parámetros y valores de retorno asociados a una RPC.

Por otro lado, los *patrones de interacción* se refieren al flujo de mensajes, no a sus contenidos. El patrón Orientado a Mensajes en-un-sentido es unidireccional; en tanto que el patrón petición/respuesta es bidireccional. El modo Documento/Literal puede usarse para cualquiera de los dos patrones. También sucede lo mismo con el modo RPC/Literal, aunque es más natural su adopción para patrones de petición/respuesta.

Otros modelos de Mensaje

Los otros dos modos de mensajes que pueden ser utilizados en SOAP son: *RPC/Codificado (RPC/Encoded)* y *Documento/Codificado (Document/Encoded)*. Sin embargo, se desestima su uso principalmente por dos razones: la especificación XML Schema los hace obsoletos e introducen cierta cantidad de problemas de interoperatividad.

El modelo RPC/Codificado recibe más atención en la especificación SOAP que cualquier otro modo de mensaje. El modo trata de definir un mapeo entre la sintaxis común de RPC y los tipos de datos de programación, de un lado y XML del otro. El modo se basa en tipos incorporados en esquemas XML, pero con la diferencia que fue diseñado para soportar estructuras de tipo grafo, que son más flexibles que las demás jerárquicas (como las explicadas en la Fig. 5.2).

A pesar que el modo RPC/Codificado fue popular al inicio, la complejidad global y los problemas de interoperatividad causaron que la mayoría de los especialistas de SOAP lo eviten. En especial la organización *Web Services Interoperability Organization (WS-I)* recomienda, en sus norma básicas de buena praxis sobre Servicios Web, no usar el modo RPC/Codificado en dentro de las envolturas SOAP [31]. Los mismos resultados pueden ser alcanzados con los modos RPC/Literal y Documento/Literal, reescribiendo la estructura del mensaje. RPC/Codificado todavía se mantiene en la especificación SOAP para aplicaciones legadas de Servicios Web que usan este modo.

El modo Documento/Codificado aplica codificación a los modelos de mensaje basado en mensajes. Sin embargo, este modo es raramente usado, ya que en la práctica se opta por el modelo Documento/Literal, por su simpleza e interoperatividad.

5.6.4. SOAP y protocolos de transporte

El paso final para lograr de SOAP un protocolo plenamente funcional es definir cómo los mensajes SOAP deben ser transportados sobre la red. La recomendación SOAP no impone ningún protocolo de transporte; sin embargo comúnmente se lo asocia a HTTP y, en menor medida, a SMTP [10].

La especificación denomina ***ligadura*** *(binding)* a esta asociación SOAP-protocolo de transporte; término que es algo confuso por mezcarse con las ligaduras estáticas o dinámicas al descubrir un servicio o servidor. En el contexto de la especificación SOAP, se denomina ***ligadura*** o ***ligadura de transporte*** a la definición de cómo los mensajes SOAP son *"envueltos"* en dentro del protocolo de transporte y cómo el mensaje debe ser tratado según las primitivas del protocolo de transporte.

Como se explicó en la Sec. 4.4.3, HTTP es uno de los protocolos más difundidos de las tecnologías de Web. La mayor cantidad de tráfico que pasa por Internet es de HTTP, y como tal es uno de los protocolos más adoptados y cuenta con la aceptación de la mayoría de los cortafuegos. Los diseñadores de SOAP vieron las posibilidades que brinda HTTP, y en consecuencia promueven la posibilidad de mandar cualquier mensaje de SOAP como carga útil de un mensaje HTTP. Este tuneleo logrado de SOAP sobre HTTP es una de las claves de la rápida aceptación de SOAP [315].

Cuando SOAP es usado sobre HTTP, un mensaje SOAP es envuelto en una *petición HTTP*. Dependiendo que lo que se requiera, SOAP puede usa GET, POST o cualquier otra primitiva HTTP. Por ejemplo, el método POST puede ser utilizado para implementar las invocaciones del ejemplo del modo RPC/Literal de comunicación (Veáse Cód. 5.10 y 5.11). El mensaje de petición SOAP es enviado como carga útil en una petición POST de HTTP. Como parte del protocolo HTTP, el nodo último receptor debe mandar un mensaje de recibido conforme de la petición POST; en el caso de que no se hayan presentado errores a nivel de HTTP. De igual forma, como respuesta al mensaje de petición SOAP, el proveedor del servicio puede incrustar el mensaje de respuesta SOAP en un mensaje POST HTTP (véase Fig. 5.6).

Además de describir cómo se deben transportar los mensajes SOAP sobre el protocolo sub-

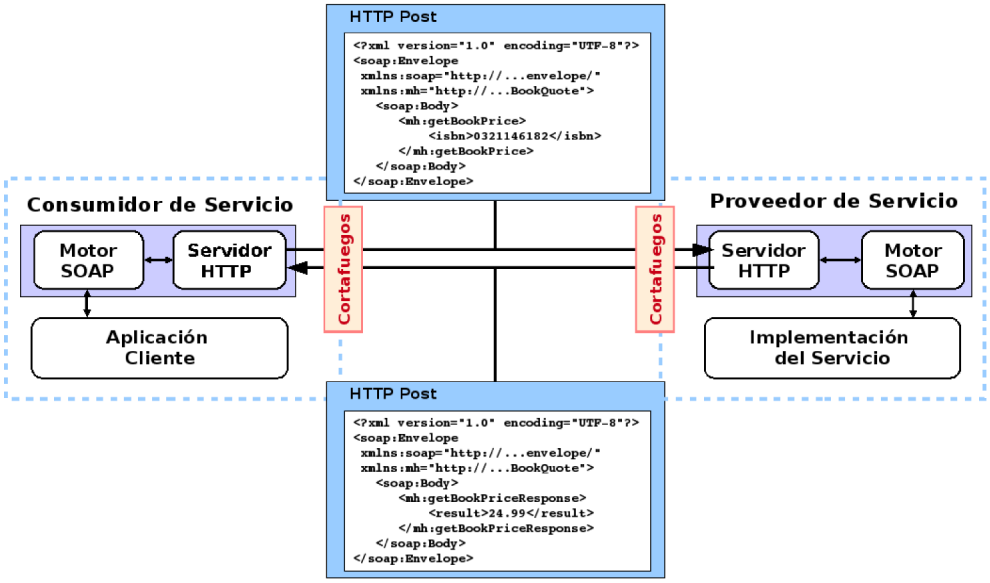

Figura 5.6: Una interacción RPC usando SOAP sobre HTTP.

yacente, una ligadura de transporte SOAP provee otra importante funcionalidad, que en el caso de HTTP está implícita: el hecho de colocar *direcciones a los servicios (adressing)*. En el caso de HTTP, el URL del destinatario del la petición HTTP, es el receptor último del mensaje SOAP.

5.7. Descripción de los Servicios Web: WSDL

Para poder invocar a un servicio web particular, es necesario conocer cómo están estructurados los mensajes SOAP que éste produce y recibe, cuál es el protocolo de transporte utilizado (HTTP o SMTP) y cuál es la dirección por la cual se accede al servicio web. En otras palabras se necesita documentación del servicio [315].

Si tomamos cómo ejemplo el Cód 5.10 y 5.11, usado para describir el modelo RPC/Literal de mensajes, se puede observar que es fácil de entender; sin embargo, no está documentado. Para usar un servicio web nuevo, se debe conocer sus funcionalidades y la manera cómo pedirlas. Si perjuicio de contar con una documentación informal, como manuales de usuario y comentarios, la comunidad de Servicios Web ha decidido contar con un mecanismo de documentación basado en formalismos. El lenguaje *WSDL* sirve para documentar precisamente y sin ambigüedades los servicios web, de forma tal que su documentación sea formal y procesable por aplicaciones informáticas [115].

El lenguaje ***WSDL*** *(Web Services Description Language)* es usado para especificar el formato exacto del mensaje, el protocolo de Internet y la dirección de acceso que una aplicación cliente usará para comunicarse con un servicio web en particular. Una especificación WSDL indica cómo debe usarse el servicio web que la misma describe. WSDL es otro estándar de facto de las tecnologías de Servicios Web, y al igual que SOAP, cuenta con amplia aceptación de los principales vendedores de software y de las organizaciones de estandarización como W3C, WS-I y OASIS. La entidad responsable del estándar es el W3C, y actualmente se cuenta con la Recomendación del WSDL 2.0 [71].

WSDL es adecuado para generadores de código, los cuales leen los documentos WSDL y generan una interfaz de programación para acceder al servicio web. Por ejemplo, los proveedores de JAX-RPC usan WSDL para generar interfaces basadas en Java RMI, y los stubs que son los responsables de establecer el intercambio de mensajes SOAP con el servicio web. Un *proveedor JAX-RPC* es una implementación de la API JAX-RPC [238]. Por ejemplo, BEA WebLogic es un proveedor de JAX-RPC. También todos los servidores de aplicación basados en Java EE son

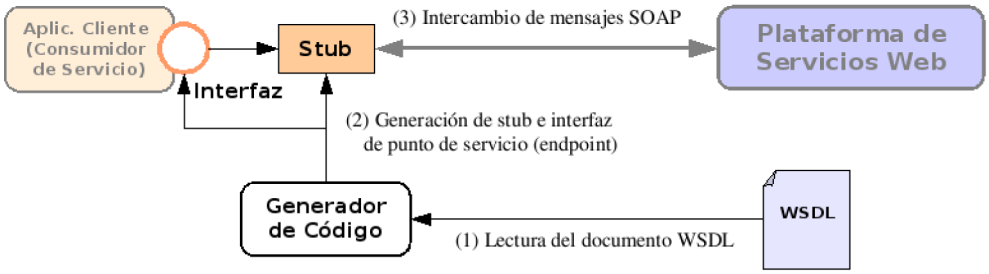

Figura 5.7: Un generación de código a partir de documentos WSDL.

proveedores JAX-RPC.

Hay que aclarar que JAX/RPC no es la única tecnología que puede generar interfaces y stubs a partir de documentos WSDL. Existen herrameintas de IBM WebSphere, Microsoft .Net y Apache Axis para tal tarea. De igual forma, al momento de redactar este trabajo, se ha estandarizado otra API: JAX-WS [239], que sustituye a JAX-RPC, extendiendo su funcionalidad.

La posibilidad de generar automáticamente las interfaces de programación y los stubs a partir de la definición de servicios es una de las principales funcionalidades de los middleware como plataforma de desarrollo. Por tal motivo, en la Fig. 5.7 se ilustra el concepto mostrando los pasos que la plataforma JAX-RPC cumple al usar los documentos WSDL para generar una interfaz basada en Java RMI para acceder a los puntos de atención o puntos finales de servicio *(endpoints)* y los stubs que implementan la interface.

Mientras que WSDL es especialmente usado en generadores de código, es también un recurso a la hora de utilizar otras herramientas o APIs. Muchas aplicaciones clientes consumidoras de servicios utilizan APIs de SOAP en lugar de generar interfaces y stubs. Estas APIs se encargan de modelar la estructura de un mensaje SOAP usando objetos que representan a cada uno de los elementos fundamentales del mensaje SOAP; como ser: objetos de clase `Envelope`, `Header` o `Body`.

Ejemplos de implementaciones de APIs de SOAP son *SAAJ (SOAP with Attachments API for Java)* [240], *Perl::Lite* [442], *Apache SOAP* [488], y otros. Cuando se utiliza una API de SOAP, los documentos WSDL son una guía para definir la estructura de los mensajes a intercambiar con el servicio web.

A pesar que WSDL es considerado un estándar, no es muy simple. Esto es debido a sus altas pretensiones. Su complejidad resulta de la intención de crear un lenguaje de definición de interfaces (IDL) para Servicios Web que no esté atado a ningún protocolo, o lenguaje de programación, o sistema operativo. En si, WSDL tampoco es específico a SOAP; es posible describir servicios web cuya interacción no esté basada en SOAP.

La modularidad también es un logro importante de WSDL. La modularidad del lenguaje permite reusar artefactos para describir más de un servicio web. Si bien en el presente trabajo, se describe la forma de entender y construir documentos WSDL, en la práctica no es necesario realizar tal trabajo mental, ya que la plataforma de desarrollo (Jave EE o .Net, por ejemplo) generan automáticamente tales documentos.

5.7.1. Estructura básica de un documento WSDL

Un documento WSDL es un documento XML que adhiere al esquema WSDL expresado en XML Schema. Como cualquier documento instancia de XML, un documento WSDL debe usar correctamente los elementos para ser bien formado y válido. A continuación se muestra la estructura de un documento WSDL para describir un servicio web en forma adecuada. Si bien WSDL puede describir cualquier tipo de servicio web, los ejemplos y explicaciones se centrarán en servicios basados en SOAP.

Un documento WSDL contiene varios elementos importantes: `types`, `import`, `message` con

`part`, `portType` con `operations`, `binding` y `service` con `port`. Todos estos elementos están encapsulados en el elemento `definitions`, que es el elemento raíz del documento WSDL [95].

En el Cód 5.12 se muestra un documento WSDL que describe el servicios web `BookQuote` (Cotización de Libro) discutido en la Sec. 5.6.3.

El elemento **`definitions`** es el elemento raíz del documento WSDL, el cual encapsula el resto del documento y le brinda un nombre a la definición WSDL. En el elemento `definitions` generalmente contiene un conjunto de declaraciones de espacios de nombres XML, lo cual es normal para un elemento raíz. En el Cód. 5.12 se ve la declaración del espacio de nombres del esquema de `WSDL XML "http://schemas.xmlsoap.org/wsdl/"`, en la cual se lo define para ser utilizado como espacio de nombres por defecto; evitando, de esta forma, que los elementos de WSDL deban ser nombrados en forma calificada (con prefijo).

El primer atributo que se presenta en el elemento `definitions` es **`name`**, el cual es usado para dar un nombre a todo el documento WSDL. Sin embargo, este nombre no es importante, ya que no existen referencias internas o externa a éste.

El elemento **`definitions`** también declara el atributo **`targetNamespace`**, el cual identifica el espacio de nombres al que se asocian los elementos definidos en dicho documento WSDL. Más adelante se verá que a los elementos `message`, `portType` y `binding` se le asigna una etiqueta como nombre; estas etiquetas (indicadas en el atributo `name` de dichos elementos) se asocian al espacio de nombres definido en el atributo `targetNamespace`.

WSDL adopta, como sistema básico de tipos, a los tipos predefinidos e incorporados en XML Schema. El elemento **`types`** sirve como contenedor para definir cualquier tipo de dato que no esté descripto en la especificación XML Schema; por ejemplo, tipos complejo y tipos simples más especializados. Para eso, WSDL permite definir tipos y estructuras de datos en el elemento `types`, que pueden ser utilizadas en las otras definiciones del documento WSDL.

Por ejemplo en el Cód 5.12 (líneas 11 a 17) se define el tipo ISBN, siendo un tipo simple especializado. El tipo definido ISBM es asignado al espacio de nombres definido en el atributo `targetNamespace` cuyo prefijo es **`nh`**. De esta forma, el nombre calificado `nh:ISBN` identifica al tipo definido y es usado en la definición del mensaje `GetBookPriceRequest` (línea 22) como tipo de dato del argumento de entrada.

El elemento **`import`** permite disponer en el presente documento WSDL de definiciones de otro espacio de nombres especificado en otro documento WSDL. Esta característica es muy usada para modularizar los documentos WSDL. Por ejemplo separar las definiciones abstractas (elementos: `types`, `message` y `portType`) de las concretas (elementos: `binding`, `service`, y `port`). O bien, para consolidar en un mismo documento WSDL varias definiciones que se desean modular en forma separada.

El elemento `import` debe declarar dos atributos: **`namespace`** y **`location`**. El valor del primero debe corresponderse con el espacio de nombres declarado en el atributo `targetNamespace` del elemento `definitions` del documento WSDL a importar. El atributo `location` debe hacer referencia a la ubicación, o URL, del documento WSDL a importar (véase Cód. 5.13 más adelante).

Definiciones Abstractas en un documento WSDL

Los elementos **`message`**, **`portType`**, y **`operation`** describen la ***interfaz abstracta*** de un servicio web. También se llaman *"definiciones abstractas"* de un documento WSDL [111, 10].

El elemento **`portType`** combina las definiciones de operaciones y mensajes en una interfaz abstracta; similar a una interfaz de programación que puede describirse en Java o C++. El elemento `portType` describe los tipos de operaciones que soporta el servicio web, sin definir el protocolo de Web que se usa para la interacción ni la URL del servicio.

Varios elementos **`message`** describen la carga útil de los mensajes que utiliza el servicio web. Además, en este elemento se puede describir los bloques de encabezados SOAP y los elementos

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

 <!-- types elements describe specific data types and structures -->
 <types>
  <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <xsd:simpleType name="ISBN">
    <xsd:restriction base="xsd:string">
     <xsd:pattern value="[0-9]{9}[0-9Xx]" />
    </xsd:restriction>
   </xsd:simpleType>
  </xsd:schema>
 </types>

 <!-- message elements describe the input and output parameters -->
 <message name="GetBookPriceRequest">
  <part name="isbn" type="mh:ISBN" />
 </message>
 <message name="GetBookPriceResponse">
  <part name="price" type="xsd:float" />
 </message>

 <!-- portType element describes the abstract interface of a Web service -->
 <portType name="BookQuote">
  <operation name="getBookPrice">
   <input name="isbn" message="mh:GetBookPriceRequest"/>
   <output name="price" message="mh:GetBookPriceResponse"/>
  </operation>
 </portType>

 <!-- binding tells us which protocols and encoding styles are used -->
 <binding name="BookPrice_Binding" type="mh:BookQuote">
  <soapbind:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getBookPrice">
   <soapbind:operation style="rpc"
    soapAction="http://www.Monson-Haefel.com/jwsbook/BookQuote/GetBookPrice"/>
    <input>
     <soapbind:body use="literal"
      namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
    </input>
    <output>
     <soapbind:body use="literal"
      namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
    </output>
  </operation>
 </binding>

 <!-- service tells us the Internet address of a Web service -->
 <service name="BookPriceService">
  <port name="BookPrice_Port" binding="mh:BookPrice_Binding">
   <soapbind:address location=
    "http://www.Monson-Haefel.com/jwsbook/BookQuote" />
  </port>
 </service>
</definitions>
```

Código 5.12: Definición WSDL del servicio BookQuote

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PurchaseOrderWS"
 targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/PO"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <xsd:schema targetNamespace="http://www.Monson-Haefel.com/jwsbook/PO">
      <!-- Import the PurchaseOrder XML schema document -->
      <xsd:import namespace="http://www.Monson-Haefel.com/jwsbook/PO"
       schemaLocation="http://www.Monson-Haefel.com/jwsbook/po.xsd" />
    </xsd:schema>
  </types>
  <!-- message elements describe the input and output parameters -->
  <message name="SubmitPurchaseOrderMessage">
    <part name="order" element="mh:purchaseOrder" />
  </message>
  ...
</definitions>
```

Código 5.13: Uso del esquemas XML previamente definido en el elemento `import`. Definición del mensaje `SubmitPurchaseOrderMessage` a partir de un fragmento XML.

que detallan las fallas. La forma en que se define el mensaje depende si se utilizará el modo de mensaje RPC o basado en documentos.

Cuando el modo de mensajes es basado en el estilo de RPC, los elementos `message` describe la carga útil de los mensajes SOAP de petición y de respuesta. Estos pueden describir parámetros, valores de retorno, bloques de encabezado o fallas. Por ejemplo, el servicio web de ejemplo `BookQuote` usa dos elementos `message` para describir los parámetros y el valor de retorno de tal servicio. En el Cód 5.12 el mensaje `GetBookPriceRequest` (líneas 21 a 23) representa los parámetros de entrada o petición al servicio web, en tanto que el mensaje `GetBookPriceResponse` (líneas 24 a 26) representa la salida o respuesta del servicio web.

Un mensaje no se declara de entrada o de salida por si sólo; esta decisión es determinada cuando se asocien los mensajes a las operaciones del servicio web.

En el estilo RPC de mensajes, cada uno de los mensajes tienen uno o más elementos **`part`**. Cada uno de estos elementos o partes del mensajes representa a un parámetro o resultado.

Cuando se utiliza el estilo de mensaje basado en documentos, la definición de los mensajes hace referencia a las definiciones del elemento `types`. Por ejemplo, en el Cód. 5.13 se describe un servicio web para enviar una orden de compra desde el cliente comercial hacia la editorial `Monson-Haefel Books`. Dicho servicio se llama `SubmitPurchaseOrder` y tiene un modo de comunicación basado en documentos en-un-sentido. Este servicio tiene un único mensaje con el que trata, el cual usa el esquema de orden de compra definido previamente (Purchase Order).

Una parte de mensaje (elemento **`part`**) puede declarar un atributo **`type`** o un atributo **`element`**, pero no ambos. La definición depende del tipo de modo de mensaje que se utilice. Si se utiliza el modo basado en RPC, las partes del mensaje deben llevar el atributo `type`. Por el contrario, si los mensajes responden al modo basado en documentos, los elementos `part` deben usar el atributo `element`. Esta distinción surge por entender que en una RPC se usan tipos para definir los parámetros de una llamada a procedimiento; en tanto que en el modo de mensaje basado en documentos se intercambian fragmentos de documentos XML los cuales hacen referencia a elementos XML que definen esos fragmentos previamente.

El elemento **`portType`** especifica una interfaz abstracta de un servicio web; define un tipo abstracto y la signatura de sus métodos. La implementación de esta interfaz definida en `portType` se "implementa" a través de los elementos subsiguientes `binding` y `service`.

```
1 <portType name="SubmitPurchaseOrder_PortType">
2   <operation name="SubmitPurchaseOrder">
3     <input name="order" message="mh:SubmitPurchaseOrderMessage"/>
4   </operation>
5 </portType>
```

Código 5.14: Definición de la operación SubmitPurchaseOrder para que respete el patrón En-un-sentido.

En una interfaz de servicio web descrita en un elemento `portType` puede tener uno o más elementos **`operation`**, cada uno de los cuales definen un método del servicio web que puede ser del estilo de RPC o basado en documentos. Cada operación definida debe estar compuesta por al menos un elemento **`input`** y/o un elemento **`output`**, que representan los parámetros de entrada y el resultado, y cualquier cantidad de elementos **`fault`** que representan las excepciones y/o errores que pueden producirse en el servicio web. Por ejemplo, en el Cód. 5.12 se puede observar que la interfaz definida en el elemento `portType` de nombre `BookQuote` (líneas 29 a 34), tiene un único método `getBookPrice` que a su vez tiene un argumento de entrada `isbn` y uno de salida `price`.

Patrones de Intercambio de mensajes

Antes de explicar los elementos asociados a las *"definiciones concretas"* en un documento WSDL, es necesario comprender los diferentes patrones de comunicación que puede establecer un servicio web. Hay cuatro *primitivas* o ***patrones de intercambio de mensajes*** usados en Servicios Web: *Petición/Respuesta*, *En-un-sentido*, *Notificación* y *Solicitud/Respuesta*. Un documento WSDL puede dictaminar el patrón de comunicación de una operación según la forma en que se declaren los elementos `input` y `output` [315, 111].

En el patrón de mensaje ***Petición/Respuesta*** *(Request/Response)*, la aplicación cliente (consumidor de servicio) inicia la comunicación mediante el envío de un mensaje de petición al servicio web; luego, el servicio web contesta con un mensaje de respuesta.

Si la operación es declarada con un único elemento `input` seguido de un único elemento `output`, se define como una operaciones de Petición/Respuesta. La orden de las declaraciones es sumamente importante. Colocar el elemento `input` en primer lugar, indica que la comunicación es iniciada por un mensaje de petición el cliente. El `output` en segundo lugar indica que el servicio web contesta al efecto. En el Cód. 5.12, la operacion definida `getBookPrice` (líneas 30 a 33) sigue este patrón.

En el estilo de mensajes ***En-un-sentido*** *(One-Way)*, los clientes mandan mensajes a los servicios web pero no esperan respuesta. Este patrón de comunicación es asociado con la comunicación asincrónica. Si una operación es declarada con un único elemento `input` y sin elementos `output`, se define implícitamente como operación en-un-sentido. El único `input` indica que la operación requiere que el cliente envíe un mensaje, sin exigir respuesta al efecto. Por ejemplo, en el Cód. 5.14 se muestra una operación en-un-sentido de nombre `SubmitPurchaseOrder` (cuyo mensaje se definió en el Cód. 5.13).

El patrón complementario al En-un-sentido es el ***Notificación*** *(Notification)*. En este patrón, un servicio web envía un mensaje a los clientes; respondiendo al esquema de comunicación publicar/suscribir (véase 3.6.3). Las operaciones definidas siguiendo el patrón Notificación, trabajan bajo la suposición que existen clientes suscriptos al servicio web para recibir sus notificaciones acerca de un determinado evento. En el patrón de Notificación, la operación definida contiene un elemento `output` y ningún `input`.

El patrón ***Solicitud/Respuesta*** *(Solicit/Response)* tiene la misma estructura que el patrón Petición/Respuesta, pero la filosofía del Notificación. En el patrón Solicitud/Respuesta es el servicio web quien inicia la interacción enviando un mensaje al cliente, el cual contesta con un mensaje de respuesta. De igual forma que en el patrón Notificación, se supone que el cliente se suscribe al servicio; con la salvedad que el cliente debe contestar a tal notificación. En la operación

que adopta el patrón Solicitud/Respuesta, primero debe declararse el elemento `output` y luego el `input`.

Definiciones de Implementación en un documento WSDL

Conforman las definiciones de implementación o definiciones concretas, los elementos `binding` y `service` con `port`.

El elemento **`binding`** mapea una definición abstracta de **`portType`** a un conjunto concreto de protocolos como ser SOAP y HTTP, a un patrón de mensajes (RPC o orientado a mensajes) y a un estilo de codificación (Literal o Codificado). El elemento `binding`, y sus subelementos, son usados en combinación con los elementos específicos del protocolo (SOAP, por ejemplo). Al respecto, los elementos `binding` especifica las asociaciones o ligaduras entre las interfaces de servicios (elementos `portType`) y operaciones (elementos `operation`) con los elementos propios del protocolo y el estilo de codificación. Cada protocolo, sea SOAP, HTTP o MIME, tiene su propio conjunto de elementos específicos y su propio espacio de nombre.

En el ejemplo usual de referencia (Cód. 5.12), se puede observar la definición del elemento `binding` (líneas 37 a 51) que implementa la definición abstracta del elemento `portType` (líneas 29 a 34), que define una interfaz para el servicio `BookQuote`. La asociación `binding` establece a SOAP como protocolo y RPC como estilo para mensajes; y además, que las operaciones son codificadas en forma literal. Lo primero que hay que notar en estas líneas de código, es que en la definición del elemento `binding` intervienen dos espacios de nombre. Los elementos sin prefijo corresponden al espacio de nombres WSDL `"http://schemas.xmlsoap.org/wsdl/"` el cual es el definido por defecto (línea 7). Los elementos de este espacio de nombres son `binding`, `operation`, `input`, y `output`. Por otro lado, existe un espacio de nombres `"http://schemas.xmlsoap.org/wsdl/soap/"` con prefijo `transport`, destinado a englobar los elementos relacionados con las vinculaciones o ligaduras entre SOAP y WSDL. Sus elementos específicos son `soapbind:binding`, `soapbind:operation`, and `soapbind:body`.

El elemento **`soapbind:binding`** identifica que el protocolo a utilizar para transportar mensajes es SOAP; esto lo realiza a través del atributo **`transport`** y con el valor `"http://schemas.xmlsoap.org/soap/http"`. Además, especifica que el modo de mensajes a utilizar es RPC, mediante el atributo **`style`**. En el caso que se opte por un modo de mensaje orientado a documentos, el valor del atributo será `"document"`. Se debe tener en cuenta que el atributo `transport` se declara explícitamente; en tanto que `style` puede no declararse en este elemento, pero será exigido en `soapbind:operation`.

Por su parte, el elemento **`soapbind:operation`** es requerido. Concreta una operación definida en la interfaz del servicio (subelemento `operation` en `portType`). Especifica el modo de mensajes soportado por la operación (atributo `style`) y define la acción SOAP (atributo `soapAction`) que debe llevarse a cabo en respuesta a la invocación de esa operación.

El atributo **`soapAction`** determina el valor que debe ser colocado en el campo del encabezado `SOAPAction` de una petición HTTP generada por el cliente. El atributo no es requerido; si el atributo se omite, también debe omitirse la inclusión del encabezado `SOAPAction` en el mensaje HTTP. En caso de no omitirse el valor del atributo debe corresponderse con el campo del encabezado. El Cód. 5.15 muestra un mensaje HTTP enviado por un cliente al servicio web definido en el Cód. 5.12.

El estilo o modo de mensaje tiene directa implicación en la forma en que se construye el cuerpo del mensajes SOAP. Si se declara que el modo es RPC (como el caso del Cód. 5.12, línea 40), el cuerpo del mensaje SOAP contendrá un elemento que representa la operación a ser ejecutada. El elemento obtiene su nombre de la operación que fue definida en `portType`. La operación contendrá uno o más parámetros (de entrada o salida), los cuales son derivados de las partes del mensaje (subatributos `part` en los atributos `message`, línea 22 y 25).

Es de notar que los atributos del elemento `soapbind:body` cambian dependiendo si se usa modo de mensaje RPC o basado en documentos. El elemento **`soapbind:body`** tiene cuatro tipos

```
POST 1ed/BookQuote HTTP/1.1
Host: www.Monson-Haefel.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction="http://www.Monson-Haefel.com/jwsbook/BookQuote/GetBookPrice"

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
   <soap:Body>
      <mh:getBookPrice>
         <isbn>0321146182</isbn>
      </mh:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

Código 5.15: Un mensaje HTTP SOAP declarando en su encabezado el campo `SOAPAction`.

de atributos `use`, `namespace`, `part`, y `encodingStyle`.

El atributo **`use`** define en modo de codificación. Las buenas prácticas [31] definen que siempre sea "`Literal`"; por consiguiente el atributo `encodingStyle`, que indica el tipo de codificación, nunca es usado.

El atributo **`part`** sirve para indicar cuáles de los elementos part del mensajes son usados en la definición del cuerpo del mensaje SOAP. Sólo es necesario si un mensaje tiene más de una parte.

En el estilo RPC de los mensajes, el atributo **`namespace`** es requerido con el valor de un URI válido. En el caso del ejemplo, es el mismo espacio de nombres definido en el atributo targetNamespace (véase Cód. 5.12, línea 3, 41 y 44). Por el contrario, el modo de mensaje basado en documento no especifica el atributo `namespace` en el elemento `soapbind:body`; ya que el espacio de nombres del fragmento XML es derivado de su esquema.

En las Secs. 5.6.1 y 5.6.2, se vio la posibilidad de que los mensajes SOAP cuenten con encabezados con diferentes bloques, los cuales podrían ser tratados por varios nodos en la ruta de un mensaje. WSDL puede definir explícitamente los bloques de encabezados a través del elemento **`soapbind:header`**, siendo éste un subelemento del elemento **input** y/o **output**.

A modo de ejemplo se presenta el Cód. 5.16 en el cual se crea un elemento `binding` (líneas 29 a 43) que describe el bloque de mensaje `message-id` (línea 35) (ilustrado en Cód 5.6). Los atributos `message` y `part` usados por el elemento `soapbind:header` (Cód. 5.16 línea 35) se refieren a un mensaje y partes definido previamente (líneas 10 a 12). En este caso, la parte del mensaje referenciada está basada en una definición propia explicitada bajo el elemento `types` (línea 5).

El elemento **`service`** contiene uno o más elementos `port`. Cada uno de los elementos **`port`** representa un *punto de servicio* o punto final de acceso *(EndPoint)* del servicio web que se detallan en el documento WSDL. Puntualmente, el elemento *port* asigna un URI a una ligadura específica o elemento `binding`; como se ve en el Cód. 5.12 (líneas 54 a 59).

Un servicio puede tener más de un elemento **`port`**, cada uno asignando un URI distinto a cada ligadura SOAP específica. Se puede tener varios elementos `port` para el mismo elemento `binding`. De esta forma se logra tener dos puntos de servicios para el mismo servicio web; similar a tener dos personas que atiendan en dos puestos de trabajo diferentes, pero para concretar la misma tarea. Con esta simple configuración, es posible lograr balanceo de carga y alta disponibilidad del servicio web. En Cód. 5.17 se describe un elemento `service` que tiene tres puntos de servicio (elementos `port`), dos de los cuales atienden a la misma ligadura SOAP.

El elemento `soapbind:address` es bastante simple para enterder; simplemente asigna una dirección Internet a una ligadura SOAP, por medio del atributo `location` (su único atributo).

```
....
<types>
  <xsd:schema targetNamespace= "http://www.Monson-Haefel.com/jwsbook/BookQuote"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="message-id" type="string" />
  </xsd:schema>
</types>

<!-- message elements describe the input and output parameters -->
<message name="Headers">
  <part name="message-id" element="mh:message-id" />
</message>
<message name="GetBookPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
<message name="GetBookPriceResponse">
  <part name="price" type="xsd:float" />
</message>

<!-- portType element describes the abstract interface of a Web service -->
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input name="isbn" message="mh:GetBookPriceRequest"/>
    <output name="price" message="mh:GetBookPriceResponse"/>
  </operation>
</portType>

<!-- binding tells us which protocols and encoding styles are used -->
<binding name="BookPrice_Binding" type="mh:BookQuote">
  <soapbind:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getBookPrice">
    <soapbind:operation style="rpc"
     soapAction="http://www.Monson-Haefel.com/jwsbook/BookQuote/GetBookPrice"/>
      <input>
        <soapbind:header message="mh:Headers" part="message-id" use="literal" />
        <soapbind:body use="literal"
         namespace="http://www.Monson-Haefel.com/jwsbook/BookQuote" />
      </input>
      <output>
         ......
      </output>
  </operation>
</binding>
```

Código 5.16: Declaración WSDL del bloque de encabezado `message-id`.

```
<service name="BookPriceService">
  <port name="BookPrice_Port" binding="mh:BookPrice_Binding">
    <soapbind:address location= "http://www.Monson-Haefel.com/jwsbook/BookQuote" />
  </port>
  <port name="BookPrice_Failover_Port" binding="mh:BookPrice_Binding">
    <soapbind:address location= "http://www.monson-haefel.org/jwsbook/BookPrice" />
  </port>
  <port name="SubmitPurchaseOrder_Port" binding="mh:SubmitPurchaseOrder_Binding">
    <soapbind:address location= "https://www.monson-haefel.org/jwsbook/po" />
  </port>
</service>
```

Código 5.17: Definición de un servicio con múltiples elementos `port`.

5.7.2. Implicaciones del modelo WSDL

La estructura plateada por WSDL tiene importantes implicaciones en cómo los servicios web son descritos y cómo interactúan.

Una implicación está relacionada con los diferentes tipos de operaciones. De hecho, los diferentes modos de comunicación y patrones de mensajes pueden asociarse a un elemento `operation` dentro de un elemento `binding` en un documento WSDL. Esto permite, a diferencia de los IDLs en las plataformas de middleware convencionales, que un servicio sea proactivo e inicie una interacción (patrones *Solicitud/Respuesta* y *Notificación*). En otras palabras, puede exponer no sólo operaciones que pueden ser invocadas, sino también operaciones que el mismo servicio puede invocar; significando que el servicio puede comportarse como cliente. Esto conlleva a que las distinciones entre cliente y proveedores de servicio sean borrosas. Esta situación es similar a la presentadas en las plataformas orientadas a mensajes (MOM y Brokers de Mensajes - véase Secs. 3.5.2 y 3.6.3) y en las basadas en Agentes por su predominante arquitectura par-a-par (véase Sec. 3.7.1). En estas plataformas y en Servicios Web las interacciones tienden naturalmente a ser asincrónicas. De hecho, es común en los dominios de Servicios Web modelar a cada parte interactuante como servicio SOAP/WSDL.

Otra implicación de la estructura de datos de la especificación es que WSDL no supone que los intercambios se harán en base a una preestablecida forma comunicación. Por esta razón WSDL puede ser utilizado para describir los dos aspectos de los servicios web. Por un lado es posible describir la interfaz abstracta, a través de las definiciones abstractas (estructuras `message`, `portType`, y `operation`) sin especificar la ubicación del servicio y sin estipular el protocolo que utiliza. En forma separada se especifican las definiciones concretas (estructuras `binding`, `service`, y `port`), la localización real y el protocolo de comunicación.

La gran ventaja de esta separación es que las especificaciones WSDL que describen interfaces abstractas son *reutilizables*: diferentes servicios pueden combinar diferentes interfaces usando diferentes ligaduras; y de esta forma, lograr que estén disponibles en diferentes direcciones o puntos de servicios [111]. También se ve facilitada la reusabilidad de definiciones porque las descripciones WSDL permiten importar otros documentos WSDL. Esto permite modular las especificaciones WSDL, en documentos que concentren definiciones abstractas y otros documentos, que importen a los primeros, para realizar las definiciones concretas mediante la definición de ligaduras y puntos de servicio.

Hay que notar también la correlación que existe entre las descripciones WSDL y SOAP. Si el mensaje definido en WSDL es intercambiado con SOAP, entonces el elemento `InterfaceBindigs` contiene toda la información para inferir y construir automáticamente el mensaje SOAP.

La habilidad de SOAP y WSDL de utilizar múltiples ligaduras de transportes y codificación XML es un aspecto crucial del esfuerzo de estandarización detrás de Servicios Web. SOAP es el "sobre genérico" para "envolver" invocaciones o interacciones que las aplicaciones pueden construir usando otras herramientas de middleware. WSDL es un sistema de descripción genérico de servicios que pueden corresponderse con implementaciones de servicios hechas en diferentes lenguajes de programación. En este contexto, si se piensa en las aplicaciones legadas desarrolladas en base a los sistemas de middleware convencionales existentes, la idea es estandarizar interacciones a través de una capa adicional basada en SOAP; y así, facilitar el desarrollo de herramientas que traducirán las interacciones de los mecanismos existentes a interacciones basadas en servicios web.

Actualmente, las plataformas de middleware permiten usar directamente SOAP y WSDL como mecanismos de interacción y descripción; sin embargo, todavía existen escenarios en los cuales SOAP y WSDL son utilizados para confeccionar envolturas sobre los protocolos y lenguajes de descripción existentes. De aquí la necesidad de WSDL de soportar múltiples protocolos de transporte.

5.8. Búsqueda y Localización de Servicios Web: Registro UDDI

El principal objetivo de ***UDDI*** *(Universal Description, Discovery, and Integration)* es proveer un marco de trabajo estándar para almacenar información acerca de las organizaciones y sus servicios web.

El fundamento de UDDI se concentra en la noción de un registro *de entidades o de negocios (business registry)*, el cual es, esencialmente, un servicio de nombre y directorio. En particular como especificación, UDDI define las estructuras de datos de este registro y las APIs para acceder, publicar, actualizar y consultar la información relacionada a las entidades o servicios web que ellas proveen [10].

Un ***registro UDDI*** es una base de datos que soporta un conjunto estándar de estructuras de datos definidas por la especificación UDDI. Las ***estructuras de datos*** modelan información acerca de organizaciones (empresas, reparticiones públicas, ONGs, etc) y de los requerimientos técnicos para acceder a los servicios web ofrecidos por esas organizaciones. Los registros UDDI son análogos a las Páginas Amarillas en las cuales se pueden localizar una organización según el servicio o rubro que cubre. Siendo estrictos, las APIs de UDDI también son especificadas en WSDL con ligaduras SOAP; por tal motivo, el registro en sí es un servicio web y puede ser accedido con SOAP [315, 334].

Las especificaciones del registro UDDI tiene dos principales objetivos respecto a la búsqueda, localización y descubrimiento de servicios web. En primer lugar, el registro UDDI sirve para encontrar información de los servicios; de esta forma, los desarrolladores pueden construir aplicaciones clientes que se comuniquen con ellos. En una especie de conformación de *ligadura estática* entre aplicaciones interactuantes, en el sentido que se trató en Sec. 3.2.4. En segundo lugar, también permite conformar *ligaduras dinámicas* entre consumidor y productor del servicio web. Por medio de consultas al registro UDDI, una aplicación cliente en ejecución puede obtener las referencias de los servicios de su interés y ligarse a éste automáticamente. Esta dualidad se refleja claramente en UDDI, ya que el registro admite información estructurada (orientada a la interpretación de procesos, propio de ligaduras dinámicas) e información no estructurada (orientadas a profesionales desarrolladores, típica de ligaduras estáticas).

Microsoft, IBM, y Ariba desarrollaron en un principio especificación UDDI, publicando por primera vez en septiembre del 2000. Casi en simultáneo se forma el consorcio *UDDI.org* al cual se invitan a otras doce compañías para participar del desarrollo de las versiones 2.0 y 3.0. En el verado de 2002 UDDI.org entregó la administración de las especificaciones UDDI a OASIS (Organization for the Advancement of Structured Information Standards). Al momento de redacción del presente trabajo, la última versión disponible de la especificación UDDI es la 3.0 [335].

Los productos basados en las especificaciones UDDI son ofrecidos por varias empresas de software, como por ejemplo IBM, Microsoft, Sun, Oracle, Fujitsu, Systinet, y otros. Otro punto importante a destacar es que UDDI soporta la mayoría de los motores de bases de datos; sin embargo, también puede ser implementado en otras tecnologías como ser servidores LDAP y bases de datos XML.

Un registro UDDI debe proveer acceso a sus datos a través de un conjunto de servicios web basados en SOAP. La especificación UDDI describe alrededor de treinta diferentes operaciones SOAP distribuidas en APIs que posibilitan agregar, actualizar, borrar, y encontrar información contenidos en el registro UDDI. La mayoría de los productos UDDI proveen una interfaz web para gestionar la información del registro.

Cualquier organización o consorcio de empresas pueden formar sus propios registros UDDI, para uso privado, o extendiendo su uso a clientes, asociados y proveedores. Además de los registros UDDI privados, se ha consolidado un registro UDDI público y de uso masivo, denominado ***Registro de Negocios UDDI*** *o UBR (UDDI Business Registry)*, el cual administraban en conjunto la empresas IBM, Microsoft, NTT, y SAP bajo un consorcio denominado *UBR Operators Council*. Este registro permitía a cualquier organización registrar sus procesos de negocios

y servicios web y también consultar y utilizar los existentes.

Se considera que la tecnología UDDI es adecuada para catalogar las aplicaciones informáticas de una determinada empresa; con el objetivo de proveer una fuente unificada de activos de software con servicios de documentación y localización. Actualmente UDDI está prestando más atención a los mercados electrónicos de servicios web y a la posibilidad de brindar registros globales, en dónde las compañías puedan organizar y compartir sus servicios web con fines comerciales. A pesar que UBR es gratis, no tiene el mismo impulso inicial; simplemente porque las empresas no han percibido el valor de registrar sus servicios en un registro global UDDI. Esto llevó a desestimar el emprendimiento y a abandonarlo en 2006. Sin embargo, se reconoce que el proyecto UBR tuvo éxito en su objetivo para probar las implementaciones de las especificaciones UDDI y refinar la interoperatividad con tales especificaciones [259, 304].

5.8.1. Información contenida en un registro UDDI

De manera sencilla, se puede puede categorizar la información contenida en un registro UDDI según el uso que a ésta se le de [95]. Se puede entender mejor esta categorización si se plantea la analogía con la guía de teléfonos:

- ***Páginas Blancas***: son los listados de entidades, de su información de contacto (teléfonos, direcciones de correo, etc.) y los servicios que las entidades proveen. Usando el registro UDDI como un catálogo de páginas blancas, los clientes del registro pueden localizar los servicios ubicándolos por la entidad que los provee.

- ***Paginas Amarillas***: son clasificaciones de entidades y de servicios web de acuerdo a una taxonomía que puede ser estandarizada o definida. De forma análoga a la clasificación por rubros comerciales en una guía de teléfonos. A través de este modo, es posible localizar un servicio buscándolo por la(s) categoría(s) a la que pertenece.

- ***Páginas Verdes***: esta información describe cómo se puede acceder e invocar un determinado servicio. Dichos datos son provistos por apuntadores a los documentos que describen el servicio (documentos WSDL), que generalmente están almacenados fuera del registro UDDI.

5.8.2. Estructuras de Datos UDDI

UDDI define cinco estructuras de datos primarias, las cuales son usadas para representar una organización, sus servicios, las tecnologías de implementación de los mismos y su relacion con otros negocios:

- La estructura ***businessEntity*** representa una entidad (empresa u organización) que provee el servicio web.

- La estructura ***businessService*** representa un servicio web u otro servicio electrónico.

- La estructura ***bindingTemplate*** representa una ligadura entre el servicio web, su punto de servicio o punto de acceso (representado a través de un URL) y una estructura `tModels`.

- La estructura ***tModel*** representa un tipo específico de tecnología o sistema estandarizado. El término *"tModel"* es un acrónimo de *"technical model"*; es decir que es un modelo técnico de referencia para indicar o categorizar algo. El uso de `tModel` abarca el ámbito de todas las otras estructuras. Se pueden utilizar estructuras `tModel` para representar sistemas de clasificación, taxonomías, protocolos de interacción, semánticas de operación, etc. Por ejemplo, al indicarse un `tModels` en una ligadura expresada por `bindingTemplate` se hace referencia al tipo de tecnología que será utilizada por el servicio web.

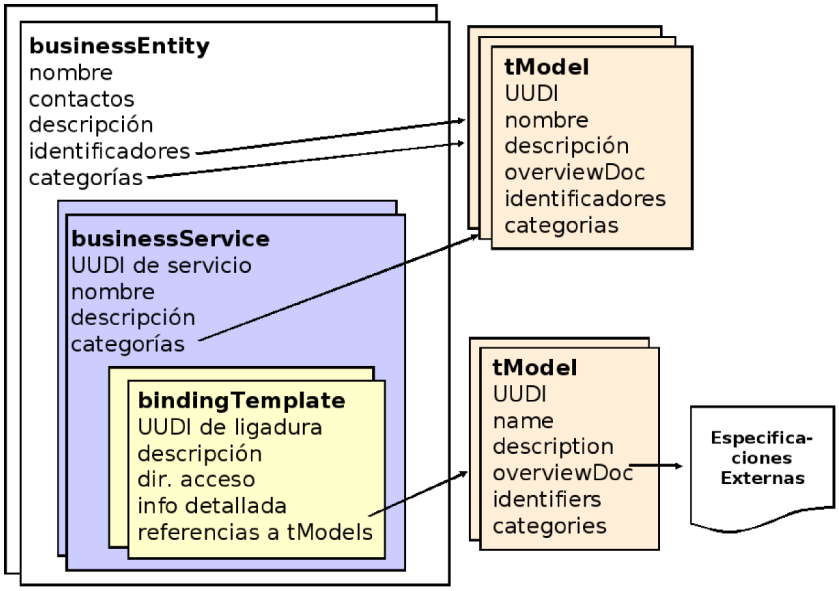

Figura 5.8: Interrelación de las estructuras UDDI.

- La estructura ***publisherAssertion*** representa una relación entre dos entidades u organizaciones.

Existen otras estructuras de datos que representan domicilios, categorías e información de contacto, URLs, y otras informaciones de la empresa y de los servicios web. Sin embargo, estas cinco estructuras son las más importantes, y son las únicas que tiene asignado una única clave de forma tal que puedan ser identificadas aún si son almacenadas en forma separada. Todas las estructuras usadas en UDDI son definidas como tipos complejos en un único documento XML Schema.

En resumen, si se desea publicar uno o más servicios web en un registro UDDI, se deben definir primero el conjunto de modelos técnicos (*tModels*) que describan las direfentes características de la entidad y de los servicios web. Hay que tener presente que existen modelos técnicos preexistentes, definidos por los consorcios de estandarización y que todos los productos UDDI soportan en una única forma a través de un único `tModel`. Luego, se publica la información de la entidad (a través de una estructura `businessEntity`), la información general acerca de los servicios que la entidad ofrece (con un conjunto de `businessService`), y los datos detallados de cada implementación y los puntos de acceso disponibles para los servicios (con `bindingTemplate`). Las descripciones técnicas en cada paso, hacen referencia a los `tModels` previamente publicados [10] (véase Fig. 5.8)

Estructura `businessEntity`

La estructura **`businessEntity`** es una estructura de datos que sirve para representar cualquier personal u organización (empresa, ente estatal, ONG, etc.) que publica servicios web en un registro UDDI. La estructura **businessEntity** provee información general respeto a la *entidad*: nombre, descripciones, información de contacto, los servicios web ofrecidos e información de identidad y categorización. En el Cód. 5.18 se ve el documento XML Schema que define a la estructura. Basada en la definición del Cód. 5.18, una instancia de este esquema sería la presentada por el Cód. 5.19 en dónde se define a la editorial `Monson-Haefel Books`.

El elemento **`businessEntity`** es la raíz de una descripción XML de una entidad. Uno de sus atributos es **`businessKey`** que conlleva un valor de *UUID (Universally Unique Identifier)* [492] (véase Pag. 44). Éste es un valor clave que genera el registro UDDI para la estructura, por única vez y en forma automática.

```
<schema targetNamespace="urn:uddi-org:api_v2"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:uddi="urn:uddi-org:api_v2"
        version="2.03" id="uddi">
 ...
 <element name="businessEntity" type="uddi:businessEntity"/>
 <complexType name="businessEntity">
  <sequence>
     <element ref="uddi:discoveryURLs" minOccurs="0"/>
     <element ref="uddi:name" maxOccurs="unbounded"/>
     <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
     <element ref="uddi:contacts" minOccurs="0"/>
     <element ref="uddi:businessServices" minOccurs="0"/>
     <element ref="uddi:identifierBag" minOccurs="0"/>
     <element ref="uddi:categoryBag" minOccurs="0"/>
  </sequence>
  <attribute name="businessKey" type="uddi:businessKey" use="required"/>
  <attribute name="operator" type="string" use="optional"/>
  <attribute name="authorizedName" type="string" use="optional"/>
 </complexType>
 ...
</schema>
```

Código 5.18: Definición de la estructura businessEntity realizada en XML Schema.

El elemento **discoveryURLs** (en plural) contiene uno o varios elementos **discoveryURL** (en singular). Cada uno de estos elementos contienen un URL a través del cual se accede por HTTP GET a los datos que se encuentran el elemento `businessEntity`.

Cada elemento individual `discoveryURL` tiene un atributo **type**, el cual puede tener el valor `businessEntity` o el valor `businessEntityExt`. Debe existir un único elemento `discoveryURL` con valor `type=businessEntity`. Este elemento contiene una URL que accede a un documento en formato XML. El elemento `discoveryURL` con valor **type=businessEntity** es generado automáticamente por el registro UDDI cuando la entidad es dada de alta o modificada. Por otro lado, los elementos `discoveryURL` con valor **type=businessEntityExt** son opcionales y pueden ser varios. El formato del documento que referencia el URL es información adicional provista por el publicador de la entidad, no es generada por el registro UDDI y puede no ser XML.

El elemento **name** contiene el nombre por el cual se conoce a la entidad representada por el elemento `businessEntity`. Pueden existir varios elementos `name`; por ejemplo, para representar el nombre de la entidad en varios idiomas o si ésta posee alias. En en el ejemplo del Cód. 5.19 se hacen referencia al nombre expresado en ingles y en castellano -español- (líneas 10 y 11). Para esto, se utiliza el atributo opcional `xml:lang` definido en la especificación XML [1].

En el elemento **description** se presenta un resumen de la descripción de la entidad. A diferencia que `name`, el elemento `description` debe llevar obligatoriamente el atributo `xml:lang`, el cual indica el idioma utilizado para redactar la descripción. En el Cód 5.19 (línas 13 y 14) se puede observar descripciones en ingles y castellano.

El elemento **contacts** contiene información de contacto de las personar responsables de mantener la información relacionada con el registro UDDI (véase Cód. 5.19, líneas 16 a 22). El elemento `contacts` es opcional, como se indica en le esquema (Cód. 5.18, línea 11).

El elemento **businessServices** contiene una o más estructuras de datos `businessService` cada uno de las cuales representa una implementación de un servicio web. Por ser de suma importancia, se hace un tratamiento aparte en la Sec. 5.8.2 a continuación.

El elemento **identifierBag** es opcional y puede contener uno o más elementos `keyredReference`. En el contexto de una estructura `businessEntity`, un elemento

[1] La especificación XML dice que los valores legales para `xml:lang` son los especificados por la RFC 1766 [12] y sus sucesoras RFC 3066 [13], RFC 3282 [14], RFC 4646 [385] y RFC 4647 [384].

```xml
<?xml version='1.0' encoding='UTF-8'?>
<businessEntity businessKey="01B1FA80-2A15-11D6-9B59-000629DC0A53"
            xmlns="urn:uddi-org:api_v2">
  <discoveryURLs>
    <discoveryURL useType="buinessEntity">
      http://uddi.ibm.com/registry/uddiget?businessKey=01B1FA80...000629DC0A53
    </discoveryURL>
  </discoveryURLs>

  <name xml:lang="en" >Monson-Haefel Books, Inc.</name>
  <name xml:lang="es" >Libros Monson-Haefel S.A.</name>

  <description xml:lang="en">Technical Book Wholesaler</description>
  <description xml:lang="es">La mayor editorial de Textos Técnicos</description>

  <contacts>
    <contact>
      <description xml:lang="en">Web Service Tech. Support</description>
      <personName>Stanley Kubrick</personName>
      <phone useType="Voice">01-555-222-4000</phone>
    </contact>
  </contacts>

  <businessServices>
    <!-- businessService data goes here -->
  </businessServices>

  <identifierBag>
    <keyedReference keyName="C.U.I.T" keyValue="30-23493556-9"
            tModelKey="uuid:6709D81E-00FA-4C5A-C101-2FA04AD02934"/>
    <keyedReference keyName="D-U-N-S" keyValue="03-892-4499"
            tModelKey="uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823"/>
  </identifierBag>

  <categoryBag>
    <!-- North American Industry Classification System (NAICS) 1997 -->
    <keyedReference
       keyName="Book, Periodical, and Newspaper Wholesalers"
       keyValue="42292"
       tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
    <!-- ISO 3166 Geographic Taxonomy -->
    <keyedReference
       keyName="Minnesota, USA"
       keyValue="US-MN"
       tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88"/>
  </categoryBag>

</businessEntity>
```

Código 5.19: Un ejemplo de la estructura businessEntity.

keyedReference encierra una asociación nombre-valor que declara el identificador o valor asignado a la entidad por alguna organización que provee o regula sistemas de identificación. Por ejemplo, en Argentina existe el C.U.I.T (Código Único de Identificación Tributaria) que es un identificador asignado por la AFIP (Administración Federal de Ingresos Públicos) a cada contribuyente [6]; en el Cód. 5.19 (líneas 29 y 30) está reflejado en el elemento `keyedReference` dicha asignación, indicando el nombre del identificador (atributo `keyName`) y el valor del identificador (atributo `keyValue`).

Además, el elemento `keyredReference` tiene un atributo **`tModelKey`**, que tiene como valor una clave que identifica a un *`tModel`*. Un **`tModel`**, en el contexto de `businessEntity`, hace referencia a un *modelo de taxonomía*; es decir, a una especificación para un sistema de clasificación o identificación específico. Estos modelos de clasificación y/o identificación se homologan en el registro UDDI y reciben, cada uno, una correspondiente clave expresada en un UUDI. Es el valor del UUDI el que se asigna al atributo `tModelKey`. Existen otros sistemas de clasificación o modelos taxonómicos, soportado por la mayoría de los productos UDDI. Ejemplos de esos sistemas son estándares utilizados en EEUU para identificar entidades: el modelo D-U-N-S[2] y el Tomas Register[3]; en el Cód 5.19 lineas 31 y 32 se hace referencia al primero de estos. Estos estándares homologados de clasificación comparten una clave única y conocida para un mismo `tModel` a través de todos los productos que implementan UDDI.

Un elemento **`categoryBag`** tiene la misma estructura que un elemento `identifierBag`, pero su `keyedReference` se refiere a códigos de categorización antes que a identificadores únicos. Por ejemplo, una entidad puede ser categorizada por rubro comercial industrial, por su ubicación geográfica, etc. La especificación UDDI requiere que se soporten varios códigos de categorización estandarizados [333]. Tres ejemplos puntuales son: *NAICS*[4], *UNSPSC*[5] y *ISO 3166*[6]. Para cada uno de estos estándares internacionales, los distintos registros UDDI referencia la misma clave contenida en el `tModelKey`. Esto significa que UDDI tiene un único modelo `tModel` de taxonomía que se comparte con todos los productos UDDI que lo soportan.

En el Cód. de ejemplo 5.19, se presentan las categorías que se correspodonden con la entidad en el sistema NAICS (líneas 36 a 40) y al ISO 3166 (líneas 41 a 45).

A pesar que estos estándares son ampliamente utilizados, se pueden utilizar sistemas de clasificación o taxonomías específicas de un determinado país o sector industrial. También, cada organización puede tener un sistema propio de homologación, para categorizar a las entidades. Para esto se pueden agregar al registro UDDI específico la vinculación con esos sistemas específicos del área/país o con sistemas internos.

Estructuras `businessServices` y `bindingTemplate`

La estructura `businessEntity` declara un elemento denominado `businessServices` (en plural), el cual puede contener uno o más elementos `businessService` (en singular). En concreto,

[2]El número *DUNS (Data Universal Numbering System)* o Sistema Universal de Numeración de Datos pertenece a un sistema desarrollado y regulado por la empresa Dun & Bradstreet (D&B) que asigna un identificador numérico único para cada entidad de negocio (sociedad, consorcio, etc.) [140].

[3]El *Registro Thomas para Fabricantes Americanos (The Thomas Register of American Manufacturers)*, conocido como "Thomas Registry", es un directorio de varios volúmenes con información relacionado con industrias, distribuidores, fabricantes y empresas de servicio de EEUU [497].

[4]NAICS es el Sistema de Clasificación Industrial de Norteamérica (North American Industry Classification System) que provee clasificación para empresas de Canadá, México y EEUU; las nacionaes que participan del Tratado de Libre Comercio de Norte América NAFTA (North American Free Trade Agreement) [507].

[5]UNSPSC es el Estándar Universal para Clasificación de Productor y Servicios (Universal Standard Products and Services Classification). Es un estándar abierto que clasifica productos y servicios, promovido por Naciones Unidas y mantenido por organizaciones voluntarias. Su codificación, basada en notación con punto, representa una ruta en la jerarquía de rubros o clases homologadas [506].

[6]ISO 3166 es un Sistema Geográfico de Ubicación (Geographic Locator System). Fue creado en 1997 por la Organización Internacional de Estándares ISO (International Standards Organization). El sistema de codificación identifica países y regiones (provincias, estádos, ciudades, etc) en el cual la entidad esta ubicada [235].

```
<schema targetNamespace="urn:uddi-org:api_v2"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:uddi="urn:uddi-org:api_v2"
        version="2.03" id="uddi">
  ...
  <element name="businessService" type="uddi:businessService"/>
  <complexType name="businessService">
    <sequence>
      <element ref="uddi:name" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:bindingTemplates" minOccurs="0"/>
      <element ref="uddi:categoryBag" minOccurs="0"/>
    </sequence>
    <attribute name="serviceKey" type="uddi:serviceKey" use="required"/>
    <attribute name="businessKey" type="uddi:businessKey" use="optional"/>
  </complexType>
  ...
</schema>
```

Código 5.20: Definición de la estructura businessService realizada en XML Schema.

el elemento **businessServices** describe el grupo de servicios web ofrecidos por la entidad.

Cada **businessService** contiene una o más entradas `bindingTemplate`. La relación entre `businessService` y `bindingTemplate` es similar a la relación entre el elemento `service` y los diferentes elementos `port` que puede contener en un documento WSDL (Vease Sec. 5.7.1).

El elemento **bindingTemplate** describe la información técnica necesaria para el uso de un servicio web. Esencialmente define la dirección por la cual está disponible el servicio web (punto de acceso, o de servicio (endpoint)); y además, detalla una serie de *tModels* que refieren a documentos técnicos relacionados con la interfaz del servicio y otras propiedades.

Al igual que otras estructuras en UDDI, la estructura `businessService` se encuentra descripta como un tipo complejo en el esquema UDDI XML Schema. En el Cód. 5.20 muestra dicha definición.

Para completar el ejemplo, en el Cód. 5.21 se muestra una instancia del tipo XML `businessService` definido en Cód. 5.20. En este caso, se detallan el servicio `BookQuote` (Cotización de Libro) ofrecido por la editorial `Monson-Haefel Books`. Este código complementa al Cód. 5.19, en el lugar expresado por las líneas 24 a 26.

Existen varios elementos `name` y `description` (véase Cód. 5.21 líneas 7, 8, 9, 13 y 14). Estos elementos pueden aparecer diferentes veces por cada idioma que se desee, haciendo uso del atributo `xml:lang`. La utilización y semántica de estos dos elementos es similar al explicado para la estructura `bussinesEntity` 5.8.2.

El elemento `categoryBag` (véase Cód. 5.20 línea 12) es utilizado en la estructura `businessService` en la misma forma que en `businessEntity` (Sec. 5.8.2). La diferencia es que los tipos de categorías usados sirven para describir el elemento `businessService`.

El elemento **bindingTemplates** (en plural) contiene una o más entradas `bindingTemplate` (en singular). Cada uno de los elementos **bindingTemplate** contendrá información sobre la dirección (URI) para acceder al servicio web y una referencia a los detalles de implementación (elemento `tModelInstanceDetails`). Esta estructura encierra la misma información de las definiciones concretas `port` y `binding` del documento WSDL del servicio (véase Sec. 5.7.1), pero del lado del registro UDDI. En el Cód. 5.22 se muestra la especificación en base a XML Schema de la estructura. Una instancia de esta definición, se presenta en el ejemplo del Cód. 5.21 (líneas 12 a 25).

Al definirse la estructura `bindingTemplate`, se debe optar entre describir el elemento `accessPoint` o el elemento `hostingRedirector`.

El elemento **accessPoint** describe la dirección exacta del servicio; la cual puede ser de

```
<?xml version='1.0' encoding='UTF-8' ?>
<businessEntity businessKey="01B1FA80-2A15-11D6-9B59-000629DC0A53"
            xmlns="urn:uddi-org:api_v2">
 ...
 <businessServices>
  <businessService serviceKey="3902AEE0-3301-11D6-9F18-000629DC0A53">
   <name>BookQuoteService</name>
   <description xml:lang="en">Given an ISBN, this service returns the price
   </description>

   <bindingTemplates>
    <bindingTemplate bindingKey="391E2620-3301-11D6-9F18-000629DC0A53">
     <description xml:lang="en">This service uses a SOAP RPC/Literal Endpoint
     </description>

     <accessPoint URLType="http">
      http://www.Monson-Haefel.com/jwsed1/BookQuote
     </accessPoint>

     <tModelInstanceDetails>
      <tModelInstanceInfo
       tModelKey="uddi:4C9D3FE0-2A16-11D6-9B59-000629DC0A53"/>
     </tModelInstanceDetails>

    </bindingTemplate>
   </bindingTemplates>

  </businessService>
 </businessServices>
 ...
```

Código 5.21: Un ejemplo de la estructura businessService.

```
<schema targetNamespace="urn:uddi-org:api_v2"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:uddi="urn:uddi-org:api_v2"
        version="2.03" id="uddi">
 ...
 <element name="bindingTemplate" type="uddi:bindingTemplate"/>
 <complexType name="bindingTemplate">
  <sequence>
   <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
   <choice>
    <element ref="uddi:accessPoint"/>
    <element ref="uddi:hostingRedirector"/>
   </choice>
   <element ref="uddi:tModelInstanceDetails"/>
  </sequence>
  <attribute name="serviceKey" type="uddi:serviceKey" use="optional"/>
  <attribute name="bindingKey" type="uddi:bindingKey" use="required"/>
 </complexType>
```

Código 5.22: Definición de la estructura bindingTemplate realizada en XML Schema.

Nombre	dnb-com:D-U-N-S
Descripción	Dun & Bradstreet D-U-N-S Number
UUID	uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823
URL	http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm#D-U-N-S
Categoría	identifier

Cuadro 5.1: tModel D-U-N-S

diferentes protocolos "mailto", "http", "https", "ftp", "phone", "other", etc. El tipo de dirección es indicada por el atributo **URLType**. Un elemento bindingTemplate puede contener sólo un elemento accessPoint. Si el servicio web es accesible por más de una URL, debe definirse diferentes bindingTemplate por cada caso. En el Cód. 5.21 (líneas 16 a 18) se ve la dirección de acceso al servicio web BookQuote. Los datos presentados en este elemento de UDDI son coherentes con los descritos en el elemento port WSDL según lo expuesto en la Sec. 5.7.1 en el Cód. 5.12 (línea 57).

Un elemento **hostingRedirector** es usado para indicar que la entrada bindingTemplate es un apuntado o referencia a otra entrada bindingTemplate diferente. Este elemento se utiliza para exponer un mismo servicio web pero con diferentes descripciones. En sí, este elemento no tiene utilización práctica.

Un elemento **tModelInstanceDetails** (en plural) contiene uno o más elementos tModelInstanceInfo; cada uno de los cuales se refiere a un tModel que describe algún aspecto técnico del servicio web. Cuando se utiliza servicios basados en WSDL con UDDI, es convención indicar a un único elemento tModel el cual referencia a un puerto específico de un documento WSDL en particular.

En el caso de ejemplo (Cód 5.21), el elemento tModelInstanceDetails es bastante simple, ya que contiene un sólo elemento vacío tModelInstanceInfo que tiene al atributo tModelKey que referencia al tModel de WSDL registrado en el directorio UDDI (líneas 20 a 23).

Estructura tModel

La estructuras tModels y las forma en que éstas son utilizadas por los elementos bussinesEntity, bussinesService y bindingTemplate son el corazón de la especificación UDDI. A medida que se entiende cómo trabajan, se revela la filosofía de UDDI. De hecho la principal información acerca de un servicio web es especificada como referencia a un elemento tModel [10].

Un **tModel** esencialmente es un espacio de nombres con meta-datos. Representa una simple especificación, una taxonomía, una categoría, un sistema de identificación y cualquier otro concepto técnico. Por ejemplo, la especificación UDDI especifica un tModel estándar para representar el Sistema de Identificación de Empresas D-U-N-S (D-U-N-S Business Identifier System); la información que expone puede verse en Cuad. 5.1. Debido a que el tModel de D-U-N-S es un estándar UDDI, es el mismo en cada uno de los registros UDDI existentes. Existen otros tModels estándares como se vio anteriormente.

Sin embargo también se puede definir un nuevo tModel para identificar cualquier cosa que se requiera. Por ejemplo se puede pensar en un tModel que describa la información requerida para el documento WSDL del servicio web BookQuote de la editorial Monson-Haefel Books (véase Cuad. 5.2). El tModel de servicio web BookQuote incluye un nombre (es común usar como prefijo del nombre un identificador del dominio -Monson-Haefel-), una descripción, un identificador único UUDI, las URL de referencia y una categoría.

Es necesario especificar un documento WSDL porque UDDI está generalizado y en un registro pueden coexistir otros modos de definir servicios web aparte de WSDL. Por tal motivo, un registro UDDI no almacena directamente los documentos WSDL para brindarlos cuando se requiera un servicio. En su lugar, los documentos WSDL se gestionan en forma separada, y existe en el registro UDDI un tModel que los referencia. Gracias a este desacople entre concepto y documentación,

Nombre	Monson-Haefel:BookQuote
Descripción	Given an ISBN, this Web service returns the wholesale price of the book.
UUID	uuid:4C9D3FE0-2A16-11D6-9B59-000629DC0A53
URL	http://www.Monson-Haefel.com/jwsed1/BookQuote.wsdl#xmlns (wsdl=http://schemas.xmlsoap.org/wsdl/) xpointer(/wsdl:definitions/wsdl:portType[@name="BookQuoteBinding"])
Categoría	wsdlSpec

Cuadro 5.2: tModel del servicio Monson-Haefel:BookQuote

es posible adjuntar meta-datos a la referencia WSDL en la forma de categorías. En el caso del servicio web BookQuote, la categoría de su tModel es wsdlSpec, lo que indica que se refiere a un documento WSDL. En tanto que el tModel D-U-M-S se corresponde con la categoría identifier, lo que significa que se refiere a un determinado sistema de identificación o categorización.

Un tModel puede ser asignado a un número arbitrario de categorías. Sin embrgo, es obligación que un tModel tenga asociada alguna de las 16 categorías estándares, denominadas tipos UDDI, o alguno de sus subtipos. Los tipos UDDI son un conjunto de categorías homologado por la especificación UDDI. Alguna de las más importantes son:

- **tModel**: es el ancestro de todos los tipos UDDI. Cada tModel es basado en este tipo o subtipo de éste.
- **identifier**: engloba a los sistemas de identificación, como D-U-N-S.
- **categorization**: engloba a las taxonomías o sistemas de clasificación, como NAICS y UNSPSC.
- **specification**: engloba a los tModels que representan especificaciones de servicios, como ser archivos de CORBA IDL y documentos WSDL.
- **xmlSpec**: es un refinamiento del anterior. Representa a los tModels vinculados con documentos XML. Los tModels de las UDDI APIs son de esta categoría.
- **soapSpec**: es un refinamiento del anterior. Los tModels vinculados a documentos SOAP generalmente se utilizan para estructuras binding Templates para indicar que la interacción de un servicio web se hace vía SOAP.
- **wsdlSpec**: es un refinamiento de xmlSpec y engloba los documentos WSDL.
- **protocol**: engloba los tModels que describen protocolos de cualquier tipo.
- **transport**: es un refinamiento del anterior, y hace referencia a protocolos de transporte como HTTP, FTP, y SMTP.

A pesar que los tModels varían, la estructura es la misma. Ésta incluye los elementos name, description, identifierBag y categoryBag. En el Cód. 5.23 se muestra la definición de tModel basada en XML Schema.

Para mostrar una instancia específicas de tModels, se presentan el Cód. 5.24 que es la definición tModel WSDL del servicio web BookQuote según los descripto en el Cuad. 5.2. Cuando se publica un servicio web en un registro UDDI, el elemento bindingTemplate debe hacer referencia a un tModel de WSDL. Además, el tModel debe ser del tipo UDDI wsdlSpec y puede proveer un indicador *XPointer* a un elemento binding específico presente en un documento WSDL accesible.

```
<schema targetNamespace="urn:uddi-org:api_v2"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:uddi="urn:uddi-org:api_v2"
        version="2.03" id="uddi">
  ...
  <element name="tModel" type="uddi:tModel"/>
  <complexType name="tModel">
    <sequence>
      <element ref="uddi:name"/>
      <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:overviewDoc" minOccurs="0"/>
      <element ref="uddi:identifierBag" minOccurs="0"/>
      <element ref="uddi:categoryBag" minOccurs="0"/>
    </sequence>
    <attribute name="tModelKey" type="uddi:tModelKey" use="required"/>
    <attribute name="operator" type="string" use="optional"/>
    <attribute name="authorizedName" type="string" use="optional"/>
  </complexType>
```

Código 5.23: Definición de la estructura `tModel` realizada en XML Schema.

```
<tModel tModelKey="UUID:4C9D3FE0-2A16-11D6-9B59-000629DC0A53">
  <name>Monson-Haefel:BookQuote</name>
  <description xml:lang="en">
    Provides a wholesale price given for a ISBN number.
  </description>
  <overviewDoc>
    <description xml:lang="en">
      This URL points to the BookQuote WSDL document.
    </description>
    <overviewURL>
      http://www.Monson-Haefel.com/jwsed1/BookQuote.wsdl
      #xmlns(wsdl=http://schemas.xmlsoap.org/wsdl/)
      xpointer(/wsdl:definitions/wsdl:portType[graphics/ccc.gif @name="BookQuoteBinding"])
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
          keyName="types" keyValue="wsdlSpec"/>
  </categoryBag>
</tModel>
```

Código 5.24: Un ejemplo de `tModel` describiendo un documento WSDL.

El elemento **name** es, por convención, basado en un URI y es usado para identificar un `tModel` sucintamente. El nombre no tiene por que ser único en un registro UDDI, ya que el verdadero identificador es el UUID indicado en el atributo **tModelKey** del elemento raíz `tModel`. El hecho de basar el nombre en URIs significa que serán calificados, por ejemplo el nombre del `tModel D-U-N-S` es `dnb-com:D-U-N-S` (véase Cuad. 5.1) y el nombre dado para el `tModel NAICS` es `ntis-gov:naics:1997`; en ambos casos el nombre empieza con un prefijo de organización, el identificador de la taxonomía y, en el caso de NAICS, la versión. También se puede utilizar el mismo modelo para nombrar a `tModels` no estandarizados; como se muestra en el Cód. 5.24 el nombre dado para el WSDL del servicio es `Monson-Haefel:BookQuote`.

El elemento **description** aparece varias veces por cada lenguaje que se requiera. Su significado y manejo es igual al expresado para la estructura `businessEntity` 5.8.2.

El elemento **overviewDoc** tiene un elemento opcional `description` y un elemento obligatorio `overviewURL`. El elemento **overviewURL** puede ser cualquier URL válida, pero comúnmente se indica un archivo que puede obtenerse por una operación HTTP GET (es decir, puede ser bajado por un navegador web). Por ejemplo, en el `tModel D-U-N-S` apunta al documento que describe todas las taxonomías estándares de UDDI (véase Cuad. 5.1).

De particular importancia es el elemento **overviewURL** el cual apunta al elemento `binding` del documento WSDL del servicio, como el ejemplo del Cód. 5.24. Si se lo vincula con los otros ejemplos, el documento WSDL que referencia el `tModel` es el presentado en la Sec. 5.7.1 en el Cód. 5.12 y el elemento `binding` se describe en las líneas 37 a 51 de dicho código. La referencia al elemento `binding` se realiza a través del sistema de referencias *XPointer*. *XPointer* es una tecnología basada en XML [275], y proporciona una forma de identificar unívocamente a fragmentos de un documento XML. La especificación XPointer ofrece un mecanismo para direccionamiento de documentos XML en función de su estructura interna, lo que permite examinar su estructura jerárquica, posibilitando conocer sus elementos y contenidos, valores de atributos y posiciones relativas.

El elemento **identifierBag** no es empleado usualmente en `tModel`, salvo si se lo utiliza con el identificador `isReplacedBy`. El identificador **isReplacedBy** indica que un `tModel` ha sido reemplazado por otro; por ejemplo, si ha surgido una nueva versión de la tecnología representada por el `tModel`.

El elemento **categoryBag** por otro lado, es usado recurrentemente en los `tModels` y básicamente sirven para asignar categorías al `tModel`. En el Cód. 5.24 se indica que el `tModel` pertenece a la categoría o tipo UDDI `wsdlSpec`.

En resumen, la estructura `tModel` es muy versátil; sirve para identificar o categorizar otras estructuras UDDI. En ella se encierra el propósito central de UDDI, que es categorizar o identificar los servicios web y las organizaciones que lo proveen, de forma que estos puedan ser localizados usando consultas al registro UDDI. Sin duda, `tModel` es la estructura más importante en UDDI.

Estructura `publisherAssertion`

En grandes organizaciones o empresas multinacionales, es usual que cada sucursal o división sean autónomas para crear sus propias entradas en el registro UDDI para los servicios que éstas ofrecen; pero sin perder la vinculación que se tiene con la casa matriz. Este objetivo puede ser logrado a través de la estructura `publisherAssertion`. Una estructura **publisherAssertion** define las relaciones entre dos organizaciones o entidades. En el Cód. 5.25 se muestra su especificación basada en XML. Una instancia del esquema de `publisherAssertion` se ejemplifica en el Cód. 5.26.

La estructura identifica a ambos participantes a través de los elementos **fromKey** y **toKey**. En una instancia de esta estructura, los elementos **fromKey** y **toKey** contendrán un único identificador UDDI de `businessEntity`.

El elemento **keyedReference** apuntará el `tModel` que representa la especificación de la relación entre las organizaciones y clasifica la relación con los atributos `keyName` y `keyValue`.

```
<schema targetNamespace="urn:uddi-org:api_v2"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:uddi="urn:uddi-org:api_v2"
        version="2.03" id="uddi">
 ...
 <element name="publisherAssertion" type="uddi:publisherAssertion"/>
 <complexType name="publisherAssertion">
   <sequence>
     <element ref="uddi:fromKey"/>
     <element ref="uddi:toKey"/>
     <element ref="uddi:keyedReference"/>
   </sequence>
 </complexType>
```

Código 5.25: Definición de la estructura publisherAssertion realizada en XML Schema.

```
<publisherAssertion>
  <fromKey>0207DE98-9C61-4138-A121-4B9E636B7649</fromKey>
  <toKey>1EE48BF0-9356-11D5-8838-002035229C64</toKey>
  <keyedReference keyName="subsidiary"
                  keyValue="parent-child"
                  tModelKey="uuid:807A2C62-EE22-470D-ADC6-E0424A337C03"/>
</publisherAssertion>
```

Código 5.26: Ejemplo de publisherAssertion.

Para que sea válida, y por ende visible, en un registro UDDI, ambas entidades deben remitir las entradas `publisherAssertion` complementarias. Esta restricción evita distorsiones; ya que si una entidad publica una relación que su contraparte no reconoce, no será válida la relación.

5.8.3. APIs del registro UDDI

Los registros UDDI tienen tres tipos de usuarios que utilizan sus APIs: los proveedores de servicios que publican servicios; los consumidores que buscan servicios y otros registros UDDI que necesitan intercambiar información.

Las aplicaciones clientes pueden acceder a seis tipos de APIs diferentes en un registro UDDI [10, 101]:

- La ***API de Consulta*** *(UDDI Inquiry API)*: incluye operaciones para encontrar entradas que satisfacen un determinado criterio y retornan un conjunto de resultado. Algunas de esas operaciones son: `find_business`, `find_service`, `find_binding` y `find_tModel`. También provee operaciones para obtener información detallada de una determinada entrada; como por ejemplo: `get_businessDetail`, `get_serviceDetail`, `get_bindingDetail` y `get_tModelDetail`. Esta API está pensada para que sea utilizada con herramientas de navegación UDDI; las cuales permiten a los desarrolladores hallar información, o son adecuadas para utilizarlas en aplicaciones clientes para establecer ligaduras dinámicas.

- La ***API de Publicación*** *(UDDI Publishers API)*: está orientada a entidades proveedoras de servicios. Permite agregar, modificar, y borrar entidades en un registro. Algunas operaciones que soportan son: `save_business`, `save_service`, `save_binding` y `save_tModel` para crear o modificar estructuras. También tiene las operaciones `delete_business`, `delete_service`, `delete_binding` y `delete_tModel` para eliminar estructuras. Luego de la publicación, el registro UDDI asigna claves únicas a las estructuras agregadas.

- La ***API de Seguridad*** *(UDDI Security API)*: permite a los usuarios obtener y descartar *tokens* o credenciales de autenticación para ser utilizadas en las comunicaciones con el

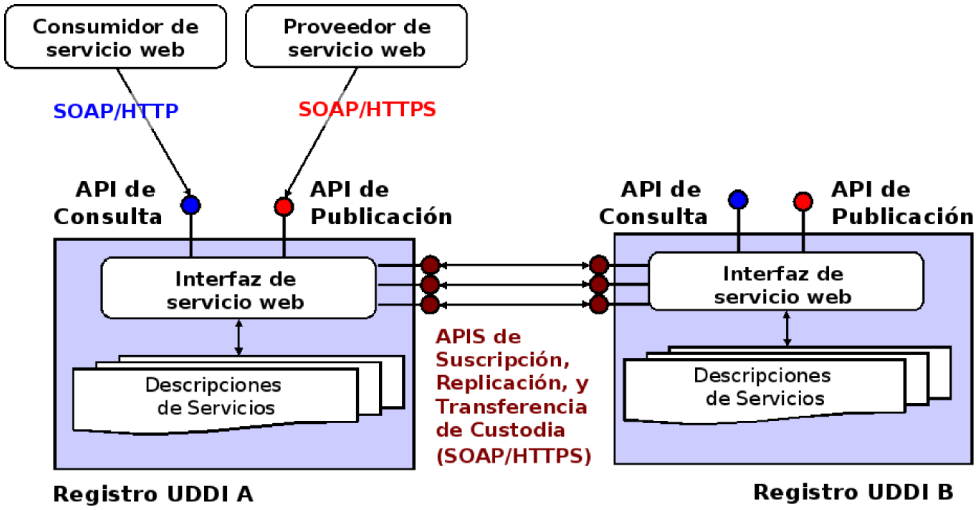

Figura 5.9: APIs de UDDI.

registro UDDI. Las principales operaciones son: `get_authToken` y `discard_authToken`.

- La ***API de Transferencia de Custodia y Propiedad*** *(UDDI Custody and Ownership Transfer API)*: permite a los usuarios registradores transferir la custodia o propiedad de las entidades o servicios entre sí. Las operaciones principales son: `get_transferToken`, `transfer_entities` y `transfer_custody`.
- La ***API de Suscripción*** *(UDDI Subscription API)*: permite monitorear los cambios en un registro mediante la suscripción, para obtener información de eventos relacionados con el alta, baja o modificaciones de entradas. La API incluye las operaciones: `save_subscription`, `delete_subscription`, `get_subscriptionResults` y `get_subscriptions`.
- ***La API de Replicación*** *(UDDI Replication API)* soporta la replicación de información entre registros UDDI, de forma tal que puedan quedar sincronizados.

Un registro UDDI mantiene diferentes *puntos de acceso* (URIs) para solicitantes de información, publicadores de información y otros registros UDDI. Por ejemplo, en la Fig. 5.9 se pueden observar dos registros UDDI. Cada uno de ellos expone sus APIs de Publicación y de Consulta a sus consumidores y publicadores respectivamente. Las APIs de Suscripción, Replicación y Transferencia de Custodia son expuestas en los registros UDDI entre sí. Una de las razones para mantener puntos de acceso separados es que una de las APIs (por ejemplo la de Consulta) no requiere autenticación; en tanto que otras (por ejemplo la de Publicación) exige algún mecanismo de validación y autenticación [101].

Una de las cosas que hace que UDDI sea interesante desde el punto de vista de los Servicios Web es que sus APIs requieren soporte SOAP sobre HTTP. Estas APIs utilizan el modo Documento/Literal para sus mensajes SOAP; las APIs son descriptas por documentos WSDL localizados en el sitio web de `UDDI.org`. Estas características hacen que los registros UDDI sean verdaderos servicios web [315].

La interacción con un registro UDDI se realiza a través de una serie de intercambios de mensajes UDDI SOAP. UDDI especifica los posibles mensajes a intercambiar en términos de peticiones y respuestas, especificados en documentos XML Schemas. Estos detalles son ocultos al desarrollador o usuario del servicio web, ya que la mayoría de las plataformas de middleware de servicios web, puntualmente los servidores de aplicaciones, proveen mecanismos para publicar y buscar entradas en un registro UDDI a través de primitivas de sistema.

La interacción entre registros UDDI es esencial por dos principales razones: para transferir la *custodia* de una entrada, y para replicar información. Las tareas de replicación se suceden

```
POST /someVerbHere HTTP/1.1
Host: www.someoperator.org
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <some-uddi-element generic="2.0" xmlns="urn:uddi-org:api_v2">
      ...
    </some-uddi-element>
  </Body>
</Envelope>
```

Código 5.27: Estructura general de un mensaje UDDI SOAP

de acuerdo al protocolo definido por UDDI, y son utilizadas para mantener registros UDDI sincronizados ya sea en escala local o global, dependiendo de la naturaleza del registro.

A pesar que la información es replicada, cada entrada es *"propiedad"* de un único registro UDDI, el cual tiene la *"custodia"* de la misma. Un usuario publicador, si desea modificar una entrada, debe acceder al registro *custodio* de la entrada para cambiarla. En algunos casos, los publicadores pueden requerir en cambio de custodia a otro registro UDDI; esto puede darse porque el registro custodio previo dejó de operar, por razones de eficiencia, o por razones contractuales. Estas situaciones exigen comunicación inter-registros usando la API de Transferencia de Custodia y Propiedad.

Búsqueda y Consulta en el registro UDDI

Todas las operaciones de Consulta y Publicación usan el modo Documento/Literal para los mensajes SOAP y el patrón Solicitud/Respuesta; lo que significa que el registro UDDI siempre contesta con un mensaje o con un error SOAP. La estructura básica de un mensaje UDDI SOAP y su encabezado HTTP se muestra en el Cód. 5.27.

Debido a que UDDI no soporta el uso del elemento `Header` en los mensajes SOAP y todos los mensajes son Documento/Literal, todo lo que se requiere conocer es la estructura del fragmento del documento XML en el elemento `Body`.

Las invocación de las operaciones de las APIs UDDI se realizan con la inserción de un tipo UDDI XML en el cuerpo del mensaje SOAP. Teniendo en cuenta el Cód. 5.27, para invocar cualquier operación de una API solamente hay que reemplazar el pseudoelemento *`some-uddi-element`* con el apropiado tipo UDDI XML (nombre de la operación) y se obtendrá un mensaje UDDI SOAP. El mensaje de respuesta sigue las mismas características. Los tipos UDDI XML son parte el espacio de nombres **`urn:uddi-org:api_v2`** para UDDI 2.0. Hay que incluir el atributo **`generic`** con el valor 2.0 para significar que el mensaje SOAP conforma con el conjunto de APIs de Programación UDDI 2.0.

Todos los mensajes SOAP son envueltos y transportados en mensajes HTTP POST. El mensaje HTTP POST debe declarar el encabezado `SOAPAction`, pero el valor puede ser la cadena vacía o cualquier valor.

Un registro UDDI provee puntos de acceso diferentes para exponer las APIs de Consulta y de Publicación. En el caso de los registros UDDI privados, estas URLs no son hechas públicas. En el caso de los registros públicos los puntos de acceso a las APIs son de dominio público y de libre acceso.

Las operaciones de la API de Consulta UDDI están divididas en dos grupos. Un grupo orientado a obtener grupos de resultados de las entradas que se corresponden con un determinado

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
      <categoryBag>
       <!-- North American Industry Classification System (NAICS) 1997 -->
       <keyedReference keyName="Book, Periodical, and Newspaper Wholesalers"
          keyValue="42292"
          tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
       <!-- ISO 3166 Geographic Taxonomy -->
       <keyedReference keyName="Minnesota, USA"
          keyValue="US-MN"
          tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88"/>
      </categoryBag>
    </find_business>
  </Body>
</Envelope>
```

Código 5.28: Búsqueda de entradas a través de categoryBag

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
      <identifierBag>
        <!-- D-U-N-S -->
        <keyedReference keyName="Monson-Haefel Books, Inc."
               keyValue="038924499"
               tModelKey="uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823" />
      </identifierBag>
    </find_business>
  </Body>
</Envelope>
```

Código 5.29: Búsqueda de entradas a través de identifierBag

criterio de búsqueda: las consultas *find_XXX*. En el otro grupo están las consultas para recuperar el detalle de una determinada entrada: las consultas `get_XXX`.

El criterio utilizado para las operaciones *find_XXX* son basados en los valores de los `tModels`, valores de campos y modificadores a las reglas por defectos establecidas por UDDI. Una variedad de elementos pueden ser utilizados para las operaciones `find_XXX`, incluyendo `identifierBag`, `categoryBag`, `name`, `tModelBag` y `findQualifiers`.

En el Cód. 5.28 se ilustra el uso de la operación `find_business` con el elemento `categoryBag`. Esta consulta retornará un conjunto de entidades que se correspondan con los valores en los elementos `keyedReference` correspondientes. En el caso de utilizar **`categoryBag`** como elemento de consulta, en comportamiento por defecto es una condición AND; es decir que una entrada será devuelta en el conjunto de resultado, debe contener todos los valores `keyedReference` de búsqueda.

Las búsquedas por el elemento `categoryBag` puede aplicarse a las operaciones `find_business`, `find_service` y `find_tModel`.

En forma similar, en el Cód 5.29 se ilustra el uso de la operación `find_business` con el elemento `identifierBag` buscando correspondencia con los valores de `keyedReference`. El comportamiento por defecto de las búsquedas usando **`identifierBag`** es con condición OR; es decir conforman el conjunto de resultado aquellas entradas (Entidades, en este caso) que contengan al menos un elemento `keyedReference` que se corresponda con la búsqueda.

Las búsquedas por el elemento `identifierBag` puede aplicarse a las operaciones `find_business`

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
      <name>Monson-Haefel</name>
      <name xml:lang="en">Addison</name>
    </find_business>
  </Body>
</Envelope>
```

Código 5.30: Búsqueda de entradas a través de name

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_service generic="2.0" xmlns="urn:uddi-org:api_v2">
      <tModelBag>
         <tModelKey>uddi:4C9D3FE0-2A16-11D7-9B59-000629DC0A53</tModelKey>
         <tModelKey>uddi:2B4C3DE0-23B4-84FE-7B22-000438FE0C22</tModelKey>
      </tModelBag>
    </find_service>
  </Body>
</Envelope>
```

Código 5.31: Búsqueda de entradas a través de tModel

y `find_tModel`.

Es posible también realizar búsquedas a través del atributo **`name`** de las entradas UDDI, con las operaciones `find_business`, `find_service` y `find_tModel`. Por ejemplo, el Cód. 5.30 muestra un mensaje SOAP para buscar entradas `businessEntity` que comiencen con algunos de los dos valores determinados.

En la búsqueda por el elemento `name`, se pueden especificar múltiples nombre; dicha consulta será realizada con una condición OR. En el caso del Cód. 5.30 obtendrá aquellas entidades cuyo nombre comience con algunos de los dos valores de búsqueda: "`Monson-Haefel`" o "`Addison`".

También es admisible utilizar el carácter comodín (`%`) en la búsqueda por nombres. Por defecto, el comodín está tácitamente colocado al final del valor a buscar; es por eso que tomará todos los nombres que comiencen con la cadena de caracteres en cuestión. De la misma forma, se puede explícitamente colocar el comodín, en cualquier lugar del texto, indicando "cualquier cantidad variable de cualquier carácter".

También es posible restringir más la búsqueda por nombre, indicando el atributo `xml:lang` para buscar entradas de se correspondan con un determinado idioma.

Las operaciones `find_business`, `find_service` y `find_binding` pueden utilizar el elemento de búsqueda **`tModelBag`** en los mensajes de consulta. El elemento `tModelBag` de búsqueda tiene de uno o más elementos `tModelKey` los cuales encierran el valor del identificador único de los `tModels` a buscar.

Puede darse que se haga referencia a un `tModel` cuyo identificador sea valor en un atributo `InstanceDetails` presente en una estructura `bindingTemplate` subelemento de un `businessService` subelemento de `businessEntity`.

Una búsqueda en base a un elemento `tModelBag` listará aquellas estructuras (incluidas las ancestras) que contengan a todos los `tModelKeys` especificados; es decir, mediante una condición AND. En el Cód. 5.31 se muestra un ejemplo de una consulta basa en `tModel`.

Utilizar el elemento **`findQualifiers`** en un búsqueda a través de las operaciones `find_XXX` permite cambiar la semántica de las condiciones de las operaciones; es decir, permite cambiar el comportamiento de las comparaciones para generar el conjunto resultado.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
 <Body>
  <find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
    <findQualifiers>
      <findQualifier>orAllKeys</findQualifier>
    </findQualifiers>
    <categoryBag>
      <keyedReference keyName="Minnesota, USA"
                keyValue="US-MN"
                tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88" />
      <keyedReference keyName="Book, Periodical, and Newspaper Wholesalers"
                keyValue="42292"
                tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2" />
    </categoryBag>
  </find_business>
 </Body>
</Envelope>
```

Código 5.32: Búsqueda usando el elemento `findQualifiers`

En un mensaje SOAP UDDI de consulta, un elemento **`findQualifiers`** (en plural) puede contener uno o más elementos **`findQualifier`** (en singular), cada uno de los cuales puede especificar un diferente calificador para modificar el comportamiento de las operaciones. Por ejemplo, el Cód. 5.32 usa elementos `findQualifiers` para cambiar el comportamiento de la operación `find_bussines` de una condición AND a una OR.

Otros modificadores posibles que puede usar `findQualifier` son:

- `caseSensitiveMatch`: establece que la búsqueda serán sensible a mayúsculas y minúsculas.
- `sortByNameAsc`: establece que el conjunto resultado esté ordenado por nombre.
- `exactNameMatch`: establece que deben buscarse las entradas cuyos valores se corresponda exactamente con el valor de búsqueda; anulando el comportamiento por defecto que utiliza el comodín tácito.

A modo de síntesis, se puede decir que desde la perspectiva de un usuario consumidor de servicios web, la API más importante es la de Consulta. Los diseñadores de UDDI intencionalmente han diseñado esta API para que sea de fácil uso, acotándola a pocos tipos de consultas. Esto se produce bajo la suposición que estándares simples tienden a ser más aceptados que los complejos. Por otro lado, consultas de mayor complejidad, requieren implementaciones complejas. Por ende, una API simple es un buen balance entre funcionalidad y simplicidad de implementación.

Este enfoque "simplista" de alguna forma, permite a los implementadores de registros UDDI brindar facilidades de consulta adicionales, más complejas. Dicho de otra forma, se promueve la diferenciación y competitividad mientras la estandarización es la base.

5.9. Resumen

En este capítulo se discutieron XML, SOAP, WSDL y UDDI como las tecnologías fundacionales de Servicios Web.

El rol básico que juega XML se manifiesta por ser el medio para expresar las estructuras de datos y las especificaciones de los demás estándares.

La interacción entre servicios web se realiza a través de SOAP. SOAP especifica los mensajes como documentos descriptos en XML y define ligaduras con los protocolos de transportes subyacentes. Una vinculación de los patrones de mensajes con los protocolos de transporte se especifica

como una ligadura; SOAP permite las ligaduras con los protocolos estándares de Internet como HTTP y SMTP. De hecho, SOAP puede ser entendido como una especificación de un protocolo que envuelve los mensajes de interacciones antes que como un protocolo de comunicación en sí. Su principal propósito es proveer una manera común de implementar diferentes mecanismos de interacción dentro de documentos XML. Por tal motivo, cada mecanismo concreto de interacción necesita una especificación SOAP; por ejemplo, las especificación para RPC sobre HTTP define cómo insertar una invocación RPC en un documento XML y cómo transmitir dicho documento por HTTP.

Las interfaces a los servicios web son definidas utilizando WSDL. Este lenguaje le permite a los diseñadores especificar el tipo de datos usado en el documento de la definición, especificar las operaciones ofrecidas por el servicio y los mensajes necesarios para invocarlas, y especificar también, las ligaduras o vinculaciones con el protocolo de interacción usado para realizar las invocaciones, por ejemplo SOAP. Lo que se conoce como servicio en WSDL es una unidad lógica que dispone de varias interfaces, todas pensadas para proveer el mismo servicio lógico, pero a través de diferentes mecanismos de acceso o paradigmas de interacción, por ejemplo interacciones basadas en documentos o RPCs.

El descubrimiento y la ubicación de servicios web está basada en la especificación UDDI. UDDI define la estructura de un registro de servicios web y un conjunto de APIs para interactuar con dicho registro. Las entradas de un registro UDDI contienen información acerca del proveedor del servicio, información de categorización y los datos relacionados con la interfaz del servicio y los puntos de acceso al proveedor del servicio en dónde se encuentra la definición WSDL del mismo. Las APIs brindan a los proveedores de servicios los medios de publicación para nuevas descripciones de servicios y, a los consumidores, las herramientas para consulta y recuperación de información de servicios de interés.

Capítulo 6

Segunda Generación de Tecnologías de Servicios Web

El rol de los servicios electrónicos dentro de una empresa es simplificar la integración de funciones y aplicaciones empresariales a través de distintos dominios tecnológicos y organizacionales. El advenimiento de los Servicios Web y sus tecnologías fundacionales (XML, SOAP, WSDL y UDDI) ha ayudado a la evolución del pensamiento relacionado con la forma en que los sistemas distribuidos pueden conectarse e interoperar en una manera cada vez más autónoma, siendo también un marco propicio para ambientes empresariales dinámicos [440].

Ninguna de las especificaciones que conforman el núcleo de Servicios Web (SOAP, WSDL, UDDI), también denominadas *especificaciones de Primera Generación o Núcleo de especificaciones de Servicios Web* [150], fueron diseñadas para proveer mecanismos que describan cómo varios servicios web puedan ser interconectados para crear soluciones informáticas, interdependientes y confiables, para los procesos de negocios con el nivel adecuado de complejidad.

En el Capítulo 3, al definirse los middlewares basados en RPCs (Sec. 3.2) se mostró que una de las limitaciones del modelo original de RPC es la falta de soporte para encarar interacciones de múltiples partes, puntos de acceso o *endpoints*. RPC es un paradigma orientado a interacciones de una invocación a la vez, entre un determinado cliente y un determinado servidor. Cuando se necesita realizar una secuencia de interacciones coordinadas, es necesario establecer mecanismos que garanticen fiabilidad de esas intertacciones más complejas; tales como el soporte a transacciones y los protocolos que puedan adaptarse a tales semánticas (como el protocolo 2PC - véase Pag. 49), e infraestructuras que implementen estas funcionalidades (como los Monitores TP - véase 3.3).

En el contexto de los Servicios Web sucede algo similar. Las tecnologías y estándares tratados en el capítulo anterior, sólo soportan interacciones simples, en las cuales una aplicación consumidora de servicio realiza una invocación a una operación residente en un servicio web. Si se desea ir más allá de estas prestaciones, se requieren nuevos protocolos, abstracciones e infraestructuras [10].

En respuesta a este desafío, surgen nuevas especificaciones que complementan el núcleo de especificaciones de Servicios Web original. A este grupo de especificaciones, se las suele denominar *especificaciones de Segunda Generación* o *especificaciones WS-**[1]. Las extensiones y características ofrecidas por las *especificaciones de Segunda Generación*, no solamente replican esas características asociadas con los middleware tradicionales y con la Integración de Aplicaciones Empresariales (EAI), sino que se construyen sobre los fundamentos de la orientación a servicios. La mayoría de los vendedores de productos tradicionales, exponen las funcionalidades a través de un conjunto de servicios web, extendiéndolas al terreno de tecnologías compatibles

[1]El término *WS-** ha llegado a ser una abreviatura usada comúnmente para referirse a las especificaciones de Segunda Generación de Servicios Web. El término WS-* se hizo popular porque los nombres de las especificaciones de Segunda Generación han sido creados usando el prefijo `WS-`.

con arquitecturas SOA. Estas especificaciones amplían el marco de trabajo establecido por las especificaciones fundacionales de Servicios Web [150, 148], explicadas en el capítulo 5.

En este nuevo escenario de requerimientos, la *composición* y *coordinación* de servicios web tiene su principal campo de acción en la *integración de aplicaciones empresariales (EAI)*, como medio para el soporte a los procesos de negocios y las actividades que ellos componen. El concepto de *actividad* y *proceso de negocio* están íntimamente relacionados. Un ***proceso de negocios*** es una colección de *actividades* o tareas las cuales logran, en conjunto, un determinado objetivo [119]. Por ende, la necesidad es proveer marcos de trabajo y de desarrollo para poder soportar, no solamente interacciones complejas y coordinadas (como las transacciones), sino también para poder describir y desarrollar procesos de negocios completos que surjan de la composición de un conjunto de actividades, implementadas como servicios web o coordinaciones entre varios servicios web, residentes en diferentes organizaciones, implementados en diferentes plataformas. Las tecnologías de Primera Generación de Servicios Web carecen de la habilidad de soportar, en forma estructural, el mantenimiento de información de contexto de una determinada actividad o conversación entre servicios web. Si no se cuenta con un contexto activo que preserve el estado de una interacción, no es posible soportar transacciones distribuidas, procesos de negocios o cualquier otro tipo de iteracción compleja.

Entendiendo la necesidad de lograr estándares para definir e implementar procesos de negocios a partir de Servicios Web, surge el *Lenguaje de Servicios Web para la Ejecución Procesos de Negocios WS-BPEL (Web Services Business Process Execution Language)* o simplemente *WS-BPEL*, denomominación que surge de su última especificación [336] gestionada por OASIS. Actualmente La especificación WS-BPEL es un lenguaje de definición de flujos de trabajo basado en XML que permite a las organizaciones definir procesos de negocios (funciones empresariales) sofisticados que pueden consumir o proveer servicios web.

WS-BPEL tiene importantes especificaciones complementarias *WS-Coordination* y las especificaciones *WS-Transaction*. Conforman a las especificaciones WS-Transaction, la especificación *WS-AtomicTransaction* y la especificación *WS-BusinessActivity*; estas dos últimas especificaciones son una posterior división de una especificación original primera [149]. En conjunto estas especificaciones tratan los detalles de cómo coordinar el resultado integral dependiente de las actividades de negocios de corta y larga vida de ejecución respectivamente. Esta cuestión es central en el éxito de una correcta implementación de sistemas informáticos distribuidas que soporten los procesos de negocios integrados en varias organizaciones; en especial para los entornos de integración de aplicaciones empresariales (EAI) y de interconección B2B, descriptos en la Sec. 3.6.

Estas especificaciones establecen una serie de conceptos, principios y modelos para la composición y gestión de actividades, cada una de las cuales con diferente alcance y propósito, pero todas relacionadas dentro del contexto de arquitecturas orientadas a servicios componibles. Estas especificaciones, y los marcos de trabajo que definen, son complementadas con extensiones o especificaciones WS-* que gobiernan áreas especificas del marco de trabajo para mensajes SOAP (*WS-Addressing* y *WS-ReliableMessaging*), la creación y el intercambio de metadatos (*WS-Policy* y *WS-MetadataExchange*), y la introducción de niveles de seguridad (*WS-Security*) [149, 150]. En la Fig. 6.1 se muestra las interrelaciones entre las diferentes especificaciones de Servicios Web.

6.1. Un escenario para la especificaciones WS-*

Para ilustrar las funcionalidades y beneficios de las especificaciones de Segunda Generación, especificamente de WS-BPEL, WS-Coordination y las WS-Transaction, se presentará una aplicación de estas tecnologías en el escenario del mundo de los negocios.

Una agencia de viajes ficticia *Acme Travel* decidió ofrecer a sus clientes comerciales el beneficio de planificar y hacer reservas de servicios de itinerarios de viaje a través de una aplicación

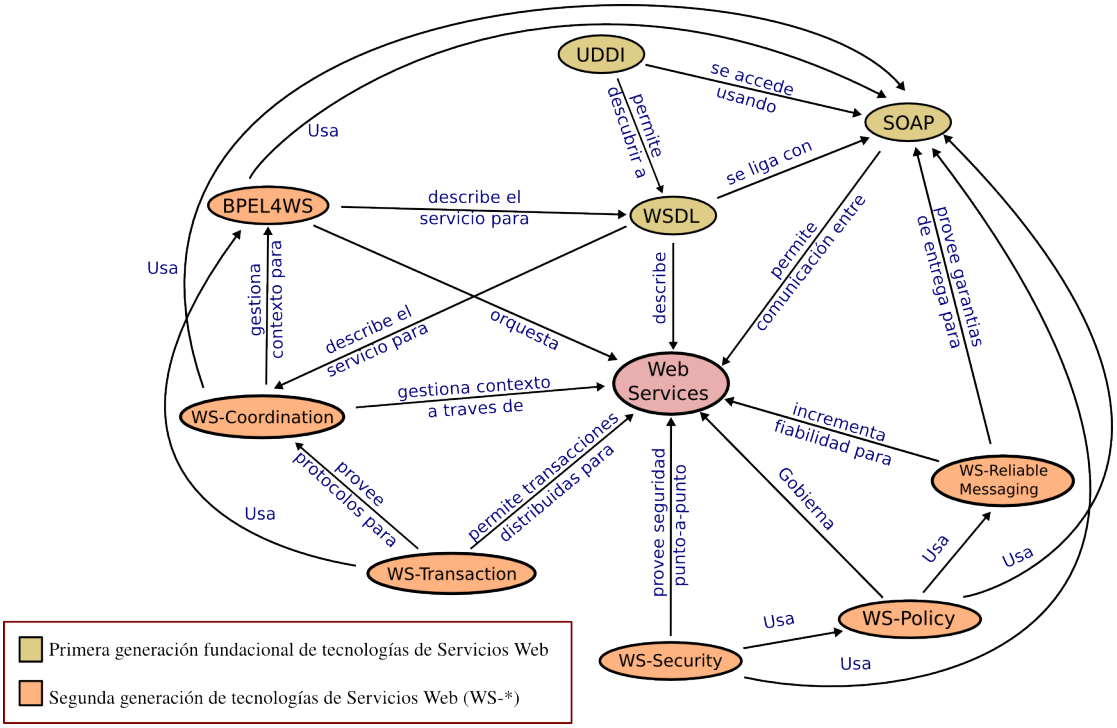

Figura 6.1: Relaciones entre las especificaciones de Servicios Web [149].

web. Como parte de esta aplicación empresarial está la posibilidad de permitir, a las empresas asociadas y a los clientes comerciales, acceso a sus servicios y aplicaciones informáticas comerciales a través de la interfaz a un servicio web. Esta opción tecnológica, le brindará a la agencia de viajes ventajas estratégicas significativas y ayudará a lograr eficiencia en ese proceso de negocio [440].

La agencia de viajes *Acme Travel* tratará de lograr tres importantes objetivos a través del desarrollo de sus servicios web:

1. permitir a los clientes comerciales remitir un itinerario de viaje,
2. procurar automáticamente las reservas apropiadas en las empresas asociadas (aerolínea, hotel y alquiler de vehículos), y
3. retornar automáticamente las confirmaciones de todas la reservas al cliente una vez que el intinerario haya sido procesado completamente.

Se entiende que para que la solución funcione, la agencia de viajes *Acme Travel* debe encontrar una forma de integrar el flujo de trabajo (encerrado en su proceso de conformación de itinerario) con las empresas de aerolíneas, hoteles y de alquiler de autos que se eligieron para cerrar el negocio; a las cuales se las denominará *empresas asociadas*, o simplemente *asociados* (*partners*). Este hecho, propone un conjunto interesante de desafíos:

- La agencia de viaje debe conectar el servicio web expuesto al cliente, con el proceso de negocios que se desea automatizar e integrar con sus compañías asociadas.
- Cada compañía asociada debe externalizar un medio que permita que *Acme Travel* pueda directamente integrar su proceso de negocios con el sistema de reservación propio de cada asociado.
- La agencia *Acme Travel* debe ser capaz de asegurar disponibilidad, confiabilidad y dependencia en el proceso integral.

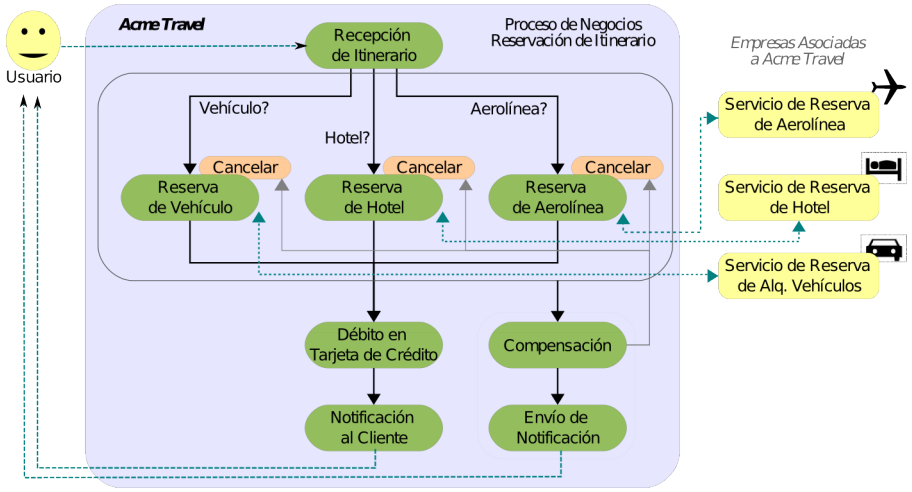

Figura 6.2: Proceso de negocios de reservas de itinerario de un agencia de viajes.

- La agencia *Acme Travel* debe ser capaz de coordinar actividades con cada asociado con el propósito de asegurar que el itinerario del cliente sea satisfactoriamente procesado.

El proceso de negocios básico de *Acme Travel* a implementar se muestra en la Fig. 6.2. Dicho proceso de negocios contiene los siguientes pasos:

1. La agencia de viajes recibe un itinerario de un cliente comercial.
2. Después de corroborar la consistencia y coherencia del itinerario, el proceso de la agencia determina qué reservaciones deben realizarse; y en consecuencia, envía peticiones a las agencias asociadas (aerolínea, hotel, alquiler de autos) para realizar las reservaciones correspondientes.
3. Si alguna de las tres tareas de reservación falla, el itinerario es cancelado mediante una actividad de *"compensación"* y el cliente es notificado del problema.
4. En caso contario, *Acme Travel* espera por la confirmación de las tres reservas.
5. Luego de recibir la confimación, la agencia de viajes confirma al cliente del éxito del proceso y le remite los números de confirmaciones y los detalles de las reservas.
6. Una vez que el cliente fue notificado del éxito o fracaso de su requerimiento, puede realizar otra petición al sistema.

El proceso en si, no es muy complejo. Los desafíos no están en el diseño de la función empresarial, sino en su implementación; en particular cuando se trata de varios asociados independientes, cada uno de los cuales puede implementar su parte del proceso integral global con diferentes requerimientos y tecnologías distintas. Tecnologías como WS-BPEL, WS-Coordination y las especificaciones WS-Transaction permiten que las soluciones a estos escenarios sean posible. En si, las tecnologías nombradas permiten alcanzar los objetivos impuestos para el proceso de negocios porque brindan los medios para:

- Definir cómo integrar los servicios de reserva de los asociados en un proceso de negocios único.
- Definir cómo las actividades especificas del proceso de negocio pueden ser publicadas y expuestas como servicios web.
- Coordinar la actividades de múltiples servicios web dentro de una transacción integral de negocios.

- Vincular dinámicamente los servicios de múltiples proveedores en tiempo de ejecución en base a las preferencias del cliente o datos derivados del proceso de flujo de trabajo (por ejemplo, elegir una determinada aerolínea, etc.).

La suposición fundamental para que la solución sea factible es que cada agencia asociada (aerolíneas, hoteles y agencias de alquiler de autos) expongan su servicio de reserva cómo un servicio web.

Adecuar el marco de trabajo de los Servicios Web para que puedan soportar secuencias de operaciones que constituyan actividades dentro de un proceso de negocios tiene importantes implicaciones. Por un lado, internamente la aplicación cliente (la aplicacion web de *Acme Travel*) debe ser capaz de ejecutar procedimientos complejos para llevar a cabo operaciones en un orden preestablecido. También debe mantener el contexto de la interacción para lograr propagar la información de ejecución de una operación a otra. Por tales razones, la lógica interna de los clientes es necesariamente más compleja. Esta lógica puede ser desarrollada en base a *composición de servicios*, similar a un sistema de flujo de trabajo adaptado a Servicios Web [10].

Paso 1: Definir el proceso de negocios.

El primer paso para *Acme Travel* es definir y documentar el proceso de negocios que se va a implementar. El *Lenguaje de Servicios Web para la Ejecución Procesos de Negocios WS-BPEL* provee una sintaxis XML que permite crear descripciones abstractas del proceso de reservaciones de itinerario.

Una de las principales diferencias de WS-BPEL sobre otras especificaciones de coordinación es que los documentos WS-BPEL son scripts *ejecutables* que pueden ser interpretados por un *motor de proceso de negocios*. Cada actividad es implementada como una interacción entre el servicio web provisto por la agencia de viajes y un servicio web provisto por algunas de sus empresas asociadas. Hay algunas porciones de scripts que necesitan ser externalizadas para permitir que la información fluya dentro del proceso y dispare varias acciones; como por ejemplo la recepción del itinerario del cliente. Estas porciones del script son expuestas como servicios web. Un motor de ejecución de procesos de negocios que comprenda WS-BPEL proveerá los recursos para estas funcionalidades. Los desarrolladores de *Acme Travel* sólo necesitan definir el proceso, proveer la lógica de negocios para cada actividad e indicar al motor cómo localizar los servicios web de cada empresa asociada que interectuarán con el proceso.

Paso 2: coordinaciones y transacciones.

Una vez que el proceso de negocios y las interacciones con las compañías asociadas han sido definidas, el próximo paso es proveer un mecanismo para coordinar todas las actividades involucradas en el proceso de negocio, de forma tal que se logre un resultado integral, producto de la composición de esas actividades. Existen varias tareas diferentes que deben ser exitosamente completadas para lograr el resultado integral del proceso de negocios; algunas de éstas pueden ejecutarse en forma simultánea, y desde el punto de vista de la agencia *Acme Travel* debe asegurarse que todas las actividades se completen exitosamente o fallen en conjunto.

La transacciones de tareas distribuidas y de tiempo de ejecución extenso siempre tienen dificultades a solucionar. Además de las cuestiones técnicas, la duración prolongada y la latencia en la comunicación introducen complejos requerimientos por intentar integrar una colección de tareas llevadas a cabo por diferentes organizaciones ejecutándose en infraestructura de software incompatibles. Las transacciones requeridas en el proceso de negocios de la agencia *Acme Travel* deben ser controladas tarea-por-tarea, con una infraestructura que monitoree y maneje los resultados de cada actividad individual como lo exige el proceso.

Otro desafío es poder conectar directamente los servicios de transacciones propios de cada organización. La mayoría de estos productos no trabajan juntos por fuera de su ámbito. Los

Servicios Web ayudan a vincular estos productos proveyendo un marco de trabajo común; a través del cual plataformas divergentes pueden ser integradas.

Las especificaciones *WS-Coordination*, *WS-AtomicTransaction* y *WS-BusinessActivity* complementan WS-BPEL al proveer un enfoque basado en Servicios Web para mejorar la interdependencia de transacciones automatizadas y de largo tiempo de ejecución en una forma extensible e interoperable.

La forma de trabajo es bastante simple. El proceso de negocios de *Acme Travel* involucra un número de servicios web trabajando juntos para proveer una solución común. Cada uno de estos servicios necesitan estar disponibles para coordinar sus actividades con el propósito que el proceso integral tenga éxito. La coordinación ocurre gracias a que cada servicio web participante comparte un poco de información que puede ser usada para vincular las actividades individuales en el proceso global. La especificación *WS-Coordination* define un marco de trabajo a través del cual estos servicios pueden trabajar en un contexto de coordinación compartido. Este contexto contiene toda la información necesaria para vincular las operaciones y soportar las iteracciones que conforman una *actividad*. Conceptualmente, una ***actividad*** es una unidad de ejecución en un proceso de negocios [118]. En el contexto de integración de aplicaciones a través de Servicios Web, una *actividad* o *actividad de servicio* está conformada por varias operaciones que son antendidas por diferentes servicios web, los cuales interactuan entre si mancomunadamente para el logro de esa tarea o actividad [150]. Es por esto, que en este contexto de coordinación y desde el punto de vista de los servicios web participantes, es usual referirse a las actividades como *conversaciones* entre servicios web [10]. En forma complementaria, las *especificaciones WS-Transaction* relacionadas con transacciones (*WS-AtomicTransaction* y *WS-BusinessActivity*), proveen un marco de trabajo para permitir a *Acme Travel* monitorear el éxito o el fracaso de cada actividad/conversación individual y coordinada. Brinda un medio para que la agencia dueña del proceso de negocios pueda monitorear el proceso global de reserva y cancelar en forma fiable todas las reservas parciales en el caso de que un asociado no pueda cumplir con su parte de la tareas. El hecho que las especificaciones WS-Transaction sean basadas en Servicios Web permite soportar transacciones que pueden interoperar traspasando los límites de los gestores de transacciones propios de cada organización participante.

Cada una define un determinado tipo de coordinación apuntado a las necesidades de clases complementarias de actividades. Ambas ayudan y conjuntamente proveen una infraestructura para la coordinación de actividades. Por un lado, *WS-AtomicTransaction* soporta a las conversaciones que requieren las propiedades tradicionales de atomicidad, consistencia, aislamiento y perdurabilidad (ACID) para transacciones; es decir, la especificación está orientada a la coordinación de actividades que son altamente acopladas. Por el contrario, las actividades que soporta *WS-BusinessActivity* son aquellas conversaciones de tiempo prolongado de ejecución, que pueden requerir que los resultados de las operaciones parciales sean visibles antes del cierre de la transacción global. La naturaleza de estas conversaciones, no admiten que exista atomicidad en la transacción, por entender que las operaciones involucradas no pueden permanecer bloqueadas hasta que se termine la ejecución global [495].

Todo este marco de trabajo, es nutrido y completado por otras especificaciones de aplicación horizontal a la estructura orientada a servicios propuesta como solución. Tales especificaciones son de suma importancia y se aplican en determinadas situaciones complementando la funcionalidad de las especificaciones orientadas a la composición y coordinación de servicios. Por ejemplo, es necesario identificar correctamente el destinatario de los mensajes que se intercambian entre participantes de la coordinación; no basta con identificar la operación del servicio web destinatario, sino que es necesario indicar una instancia de ejecución particular de ese servicio, identificando la interacción en la que la instancia participa. Una solución a este problema de direccionamiento es provista por WS-Addressing.

Por otro lado, es importante tener en cuenta que una actividad, encierra una secuencia de mensajes que establecen un diálogo entre partes (servicios web). Es deseable que esa secuencia

se respete, y fundamentalmente se determine en forma confiable si los mensajes llegaron o no a destino. WS-RealiableMessaging es una especificación que abarca tal problemática y brinda al marco de trabajo de coordinación autenticidad en la entrega de mensajes.

Otras especificaciones WS-* pueden ser empleadas. Por ejemplo, para proveer seguridad a los mensajes (WS-Security) y, para lograr expresividad en la semántica de los servicios web y en el intercambio de mensajes más alla que la expresada con WSDL (WS-Policy y WS-MetadataExchange).

El modelo de gestión del proceso de negocios, de coordinación y de transacción propuesto ayuda a reconocer algunos beneficios claves:

- El modelo está sustentado en la infraestructura de Servicios Web. La tecnología de Servicios Web permite a las aplicaciones ejecutarse en diferentes plataformas, en algunos casos totalmente incompatibles, y lograr integración e interacción en una forma mucho más flexible que las propuestas por los sistemas de middleware convencionales.

- El modelo es extensible. WS-BPEL, WS-Coordination y las especificaciones WS-Transaction, en su núcleo proveen un marco general para definir la forma en que los procesos de negocios pueden ser implementados, pero deja plena libertad para extender y personalizar los detalles para evolucionar la arquitectura SOA a medida que los procesos de negocios evolucionen.

- El modelo es flexible, provee soporte para un amplio espectro de procesos de negocios transaccionales y no-transaccionales.

- El modelo brinda procesamiento duradero y confiable. La especificaciones WS-Transaction provee los medio para monitorear la certeza de que cada tarea individual debe ser completada en el proceso. La especificación WS-BPEL, por su parte, provee los medios para definir cómo las tareas falladas deben ser compensadas.

- A nivel de negocios, WS-BPEL, WS-Coordination, y WS-Transaction ayuda a la agencia de viajes *Acme Travel* en todos sus objetivos en el sentido que permite fácilmente integrar sus actividades de negocios con aquellas de sus agencias asociadas y clientes, sin importar la plataforma específica de desarrollo o de ejecución de cada parte. Esta posibilidad, le da oportunidad a la agencia *Acme Travel* y a sus asociadas en enfocarse en lo que realmente cuenta: la solución del negocio.

Se ha presentado un ejemplo simple para brindar una idea de lo que proveen las tres especificaciones. En los negocios de la realidad, hay mucho más detalle y opciones que generan mayor complejidad a cualquier proceso de negocios. WS-BPEL, WS-Coordination, y las especificaciones WS-Transaction están diseñadas para manejar todos los niveles de complejidad en una forma consistente basada en las tecnologías fundacionales de los Servicios Web.

Este capítulo se centra en la presentación de tales tecnologías necesarias para Servicios Web. En primer lugar se tratan las extensiones necesarias para dar soporte a la coordinación entre servicios web. Se explican cómo se pueden modelar interacciones más complejas, cuáles son las abstracciones que se pueden proveer para simplificar el desarrollo y cómo la plataforma de middleware soporta tales abstracciones. En particular se explica la especificación WS-Coordination.

Se trata, a continuación, los modelos y especificaciones de transacciones, basados en el marco de coordinación anterior. Se hace un análisis de los dos modelos transaccionales: Transacción Atómica y Actividad de Negocios, provistos por las tecnologías: WS-AtomicTransaction y WS-BusinessActivity, respectivamente.

Se abarcan los estándares y especificaciones orientados a la composición de servicios web intraempresariales. Se tratan en particular el modelo de orquestación de servicios web, tomando como ejemplo la especificación WS-BPEL para descripción de procesos de negocios. Luego se

Figura 6.3: Una conversación simple entre cliente y servicio. Las lógicas internas de consumidor y proveedor de servicio deben soportar la conversación; siendo responsables de mantener un estado que se preserve entre las invocaciones de la misma.

abarca el modelo de coreografía de servicios web, enunciando los principios sustentados por WS-CDL.

Para finalizar el capítulo, se describen otras tecnologías de Segunda Generación que proveen caracteristicas y funcionalidades puntuales y esenciales para la realización del marco de trabajo de composición, coordinación y transacciones en Servicios Web.

6.2. Coordinación en Servicios Web

Para comenzar el tratamiento del capítulo, se expondrá en primer lugar las problemáticas y características relacionadas a la coordinación en Servicios Web.

En aplicaciones reales, las interacciones son típicamente más complejas que una simple invocación aislada a un servicio web. Generalmente involucran un conjunto de interacciones que casi siempre debe ser ejecutadas respetando una secuencia. Por ejemplo, puede existir un servicio web que sirva para realizar compras de algún producto; y las funcionalidades requeridas son: solicitar un presupuesto, ordenar los productos a adquirir y realizar pago de la compra. Cada uno de estos pasos pueden ser modelados como operaciones expuestas por un servicio web que cumple la función de ***proveedor*** *(Supplier)* de tales operaciones (véase Fig. 6.3).

Como se explicó anteriormente, si se parte del punto de vista del proceso de negocio, esta secuencia de interacciones conforman una *actividad* [92]. Por otro lado, desde el punto de vista de los servicios web, como se considera que éstos "dialogan". Por ende, al conjunto de interacciones que deben ejecutarse en secuencia también se lo denomina *conversación* [10].

La naturaleza de bajo acoplamiento de los servicios web requiere que los principios para mantener la persistencia de la información contextual de las actividades sean diferentes respecto a la forma de trabajo de los ambientes distribuidos tradicionales. Con el objetivo de preservar la integridad de una actividad, es esencial la existencia de un servicio que gestione el contexto y el estado de las actividades. Para poder aplicar este contexto a la gestión de transacciones, se requieren protocolos estructurados que sean capaces de dictar los aspectos de comportamiento de los servicios que participan en una actividad [149].

Surgen así dos problemáticas a abordar: por un lado el diseño y la implementación de la *actividad/conversación* refiriéndose a la secuencia de operaciones que debe establecerse entre cliente y servicio; y en segundo lugar, los *protocolos de coordinación* para definir un conjunto de conversaciones correctas y aceptadas. Ambos aspectos son complementarios, presentan determinados problemas y requieren diferentes abstracciones.

Al tratarse en este contexto de estudio, con *protocolos de coordinación* se hace referencia la definición de roles y posibles mensajes que conforman la especificación de una actividad particular. No hay que entender en este contexto la idea de protocolo como estándar tecnológico o especificación (como puede ser el "protocolo" HTTP). Un ***protocolo de coordinación*** establece las reglas de interacción entre participantes de un tipo de actividad en si; y una conversación es una instrancia de ese protocolo. Por otro lado, el estándar WS-Coordination sirve como marco de trabajo para definir protocolos de coordinación.

En la cuestión de la interacción coordinada, el problema es cómo los servicios web pueden establecer los protocolos de coordinación y de qué forma esta información estará disponible para los clientes. De hecho, al igual que las interfaces WSDL están publicadas, se deberían poder subir a un registro los protocolos de coordinación que poseen los servicios web, para que dichos protocolos puedan ser descubiertos y recuperados por las aplicaciones clientes.

Las operaciones de una conversación (o actividad) entre servicios web que son comunes para explicitar coordinación son definidas mediante la especificación *WS-Coordination*. Las especificaciones *WS-AtomicTransaction* y *WS-BusinessActivity* definen cada una, un determinado tipo de coordinación que trata las necesidades de clases distintas de actividades. Ambas ayudan y conjuntamente proveen una infraestructura para la coordinación de actividades que son altamente o bajamente acopladas respectivamente. Por su parte, WS-AtomicTransaction soporta a las conversaciones que requieren las propiedades tradicionales de atomicidad, consistencia, aislamiento y perdurabilidad (ACID) para transacciones. Aquellas conversaciones que requieren que las operaciones parciales sean visibles antes del cierre de la transacción global, están dentro de las que soporta la conclusión final WS-BusinessActivity [92].

En una conversación entre servicios web, estos servicios pueden pertenecer a diferentes empresas; en este caso, existe la necesidad de soportar relaciones de cercanía y confianza entre las organizaciones. Cada relación entre servicios pares debe ser definida para incluir estrictamente sólo la información necesaria para que ambas partes puedan interoperar y lograr el resultado final [92]. Estas especificaciones logran este objetivo asumiendo que:

- Las operaciones entre participantes son asincrónicas.
- Existe un registro explícito en las actividades cuyo comportamiento está predefinido.
- Toda la comunicación entre participantes está basada en una colección de mensajes estipulados y estos mensajes manejan patrones de coordinación.
- Es factible la composición con otras especificaciones WS-*, como ser entrega fiable de mensajes (WS-Reliable Messaging) o intercambio de mensajes seguros (WS-Security) y otros.

6.2.1. WS-Coordination

Cuando un grupo de servicios web interactúan colectivamente para ejecutar una unidad de la lógica de negocios o actividad, generalmente se requiere que estos servicios puedan compartir un contexto común. Gracias a este enfoque, la actividad recibe una identidad que es propagada a cada participante de la conversación.

En este contexto, no sólo se define la existencia en tiempo de ejecución de la actividad, sino que también se establece un nivel de control sobre la forma en la cual la actividad debe ser ejecutada. Cada participante de una actividad, como servicio, introduce un nivel de contexto en un ambiente de ejecución de aplicación. Cada cosa que ocurre o se ejecuta tiene significación en el tiempo de vida de ese contexto, y las descripción de esa significación (y otras características relativas a su existencia) pueden ser clasificadas como información de contexto [150].

El tener mayor complejidad en la actividad, trae aparejado mayor información de contexto. La complejidad de una actividad está relacionada a varios factores, en los cuales incluye:

- la cantidad de servicios que participan en la actividad,
- la duración de la actividad,
- la frecuencia en la cual la naturaleza de la actividad cambia,
- si pueden o no coexistir concurrentemente múltiples instancias de una actividad.

Se requiere de un marco de trabajo para mantener la información de contexto y estado en actividades complejas, que permita gestionar, preservar, actualizar y distribuir esta información entre los participantes de una actividad. El principal objetivo de ***WS-Coordination*** o ***WS-C*** es brindar dicho marco de trabajo [150].

WS-Coordination es una especificación inicialmente propuesta por IBM, Microsoft y BEA en 2002 [91]. Después de un tiempo, la administración y gestión de la especificación fue transferida a *OASIS (Organization for the Advancement of Structured Information Standards)*; actualmente se ha lanzado la versión 1.1 [343].

Desde el punto de vista funcional, *WS-Coordination* propicia un ambiente para soportar los protocolos de coordinación usados en una actividad/conversación [10]. En este sentido, es una meta-especificación que posibilita definir especificaciones para declarar formas concretas de coordinación. Como tal, provee métodos para:

- Gestionar el contexto de coordinación para lograr administrar y propagar el identificador único de conversación entre los servicios web intervinientes.
- Registrar en un *manejador de protocolos* o *Servicio Coordinador* el endpoint de un servicio web que participa en la conversación. En si, WS-Coordination provee la forma para que los participantes de una actividad/conversación puedan registrarse en un manejador de protocolo asociándose a un determinado protocolo de coordinación.
- Registrar los roles de los participantes de una conversación en el Servicio Coordinador.

Componentes en una coordinación

Las entidades básicas de marco de trabajo que define WS-Coordination son los *servicios web coordinadores* y los *servicios web participantes*. Un ***Coordinador*** es un servicio web que es un endpoint para una instancia de un protocolo de coordinación y posee el rol de **coordinador** de esa conversación. Si bien la definición de la semántica de los roles de coordinador y participante es parte del protocolo específico de coordinación, se puede decir que un servicio coordinador controla la composición de otros tres tipos de servicios (servicios de activación, servicio de registro y servicios de protocolos especificos) los cuales juegan su rol en la gestión de la información del contexto de una actividad [150]. Por otro lado, un ***Participante*** es un endpoint para una instancia de un protocolo de coordinación, cuyo servicio web que cumple el rol de **participante** [92] (véase Fig. 6.4).

WS-Coordination usa tres abstracciones para describir las interacciones entre los coordinadores y los participantes: *tipo de coordinación*, *protocolo de coordinación* y *contexto de coordinación*.

- Un ***protocolo de coordinación*** es un conjunto de reglas que gobiernan las conversaciones entre el coordinador y los participantes, lo que incluye el formato de mensajes y las reglas de secuenciado. El protocolo 2PC (véase Sec. 3.3.2) es un ejemplo de protocolo de coordinación. En el caso de Servicios Web, el protocolo *WS-AtomicTransaction two-phase commit* y *WS-BusinessActivity BusinessAgreementWithCoordinatorCompletion* son ejemplos de protocolos. Un *protocolo de coordinación* es mejor visto como un conjunto de reglas que son impuestas a una actividad y las cuales todos los participantes registrados deben seguir.
- Un ***tipo de coordinación*** representa un conjunto o familia de protocolos de coordinación, lógicamente relacionados que tratan una clase particular de coordinación. Una instancia de un tipo de coordinación puede involucrar a la ejecución de varias instancias del mismo o de distintos protocolos. Cada coordinación está basada en un determinado tipo, el cual especifica la naturaleza y la lógica subyacente de cada contexto de información a ser gestionado. Los tipos de coordinaciones son definidos en diferentes especificaciones. El marco de trabajo de WS-Coordination es extensible y puede ser utilizable con una variedad de protocolos de

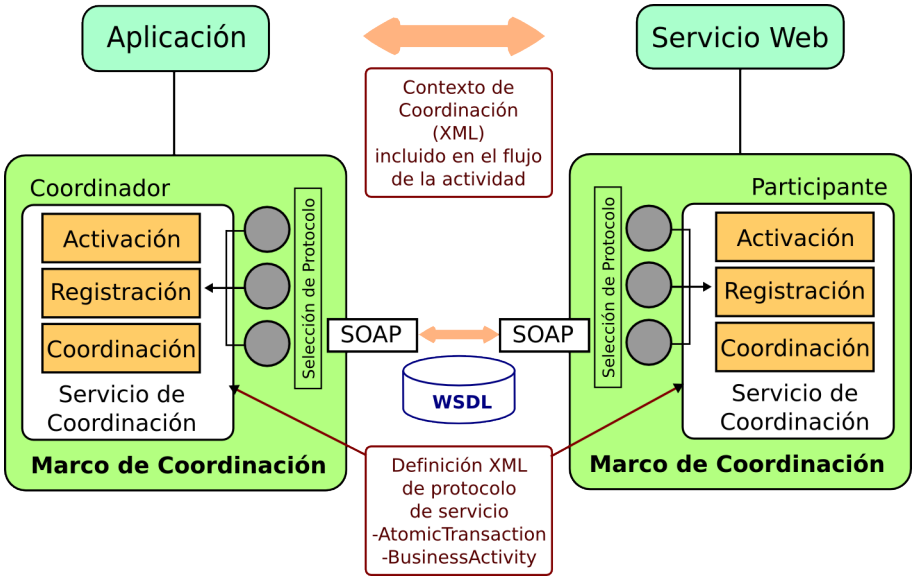

Figura 6.4: Una visión general de WS-Coordination [495].

coordinación, incluyendo aquellos específicamente desarrollados. Sin embargo, existen dos tipos de protocolos de coordinación, WS-AtomicTransaction y WS-BusinessActivity, que son los más comúnmente asociados con WS-Coordination. Las extensiones de los tipos de coordinación proveen un conjunto de protocolos de coordinación, el cual representa únicas variaciones de tipos de coordinación, y consistente en una colección de comportamientos específicos y reglas.

- Un ***contexto de coordinación*** es una estructura de datos, representada en XML, usada para marcar los mensajes que pertenecen a la misma actividad o "coordinación", como lo denomina la especificación WS-Coordination. Todo mensaje SOAP puede ser intercambiado entre participantes de una única conversación mediante la inclusión de la correspondiente estructura del contexto de conversación, la cual contiene el identificador único de la instancia del tipo de coordinación. Al insertar la información de contexto en un mensaje, se invita a participar de una actividad o conversación al servicio web destinatario del mensaje. La información de contexto incluye los datos necesarios para registrarse y participar de la conversación.

Contexto de Coordinación

El elemento **`CoordinationContext`** es usado por los servicios web que interactuan en una actividad para pasar la información del contexto de una coordinación. El elemento `CoordinationContext`, una vez creado, es propagado a los otros servicios web que serán partes de la conversación. Estos servicios deben registrarse como *Participantes* de la actividad, cuyo contexto está descripto en el elemento `CoordinationContext` propagado. Generalmente el contexto de una actividad se comparte a través de mecanismos definidos por la aplicación distribuida; es común hacerlo como un bloque de encabezado en los mensajes SOAP que intercambian las participantes de la coordinación que conforma la aplicación distribuida [343].

Para especificar contextos de coordinaciones se utiliza la definición XML-Schema `http://www.w3.org/2003/05/soap-envelope`. Un elemento `CoordinationContext` incluye la siguiente información [343, 92] (véase Cod. 6.1) :

- Un **identificador** de actividad/conversación (líneas 9 a 11).
- El **tipo de la coordinación**. Con este tipo se especifica un conjunto de protocolos de

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
3    <soap:Header>
4      <wscoor:CoordinationContext
5          xmlns:wsa="http://www.w3.org/2005/08/addressing"
6          xmlns:wscoor="http://docs.oasis-open.org/ws-tx/wscoor/2006/06"
7          xmlns:myApp="http://www.example.com/myApp"
8          soap:mustUnderstand="true">
9        <wscoor:Identifier>
10           http://Fabrikam123.com/SS/1234
11       </wscoor:Identifier>
12       <wscoor:Expires>3000</wscoor:Expires>
13       <wscoor:CoordinationType>
14          http://docs.oasis-open.org/ws-tx/wsat/2006/06
15       </wscoor:CoordinationType>
16       <wscoor:RegistrationService>
17          <wsa:Address>
18            http://Business456.com/mycoordinationservice/registration
19          </wsa:Address>
20          <wsa:ReferenceParameters>
21            <myApp:BetaMark> ... </myApp:BetaMark>
22            <myApp:EBDCode> ... </myApp:EBDCode>
23          </wsa:ReferenceParameters>
24       </wscoor:RegistrationService>
25       <myApp:IsolationLevel>
26           RepeatableRead
27       </myApp:IsolationLevel>
28     </wscoor:CoordinationContext>
29       . . .
30   </soap:Header>
31   </soap:Body>
32      . . .
33   </soap:Body >
34 </soap:Envelope>
```

Código 6.1: Ejemplo de contexto de coordinación propagado en un encabezado SOAP. La coordinación encierra una transacción atómica.

coordinación que describen los comportamientos mostrados por las partes en una actividad. En el ejemplo es una `AtomicTransaction` (líneas 13 a 15).

- Una **referencia al endpoint Servicio de Registro** (EndpointReference). Es decir, la dirección del servicio que es usado para registrar la participación en una actividad. Generalmente la referencia se describe siguiendo la especificación WS-Addressing [201] (líneas 5 y 16 a 24).
- Un **tiempo de expiración** (opcional) de la actividad (línea 12).
- **Elementos adicionales** para extender la información que puede ser comunicada, generalmente relacionada de las extensiones específicas implementadas (líneas 7, 21, 22 y 25 a 27).

Servicio de Coordinación

La especificación WS-Coordination establece un marco de trabajo que introduce un servicio general basado en el modelo de coordinador de servicio. Para lograr coordinación, WS-Coordination define al ***Servicio de Coordinación*** o ***Coordinador*** como una composición que consiste en los siguiente servicios web [150, 343]:

- ***Servicio de Activación***: que es el responsable de la creación de un nuevo contexto y la asociación de este contexto con una actividad particular. El contexto es creado a partir de la nueva instancia de un tipo de coordinación.

- ***Servicio de Registro***: permite a los servicios participantes usar la información de contexto recibida del Servicio de Activación para registrarse en un protocolo específico de la coordinación, soportado por el Coordinador o manejador de conversación. A través del registro, un servicio web declara que será participante en una ejecución de las instancias del tipo de protocolo, y que debe ser notificado cuando los pasos correspondientes son ejecutados.

- ***Servicios específicos de protocolos***: estos protocolos representan los tipos de coordinaciones soportadas por el Coordinador.

La Fig. 6.5 ilustra un ejemplo de cómo dos servicios de aplicaciones (*App1* y *App2*) con sendos coordinadores (*CoodinadorA* y *CoordinadorB*) interactúan a medida que la actividad se propaga entre ellos. El protocolo **`Y`** es específico al tipo de coordinación, y es tratado por los endpoints **`Ya`** e **`Yb`** en los respectivos servicios. El protocolo **`Y`** no está definido en la especificación WS-Coordination.

1. (a) *App1* envía un mensaje `CreateCoordinationContext` para una coordinación de tipo *Q*, al Servicio de Activación `ASa` del Servicio de Coordinación `CoordinatorA`. (b) En respuesta, el Servicio de Activación `ASa` crea el contexo `Ca` de tipo `Q` y manda un mensaje `CreateCoordinationContextResponse` a *App1* con el contexto `Ca`; el cual contiene el identificador de actividad `A1`, el tipo de coordinación `Q` y la referencia al endpoint del Servicio de Registración `RSa` de `CoordinadorA`.

2. *App1* envía un mensaje de aplicación a *App2* conteniendo como bloque de encabezado SOAP al contexto `Ca`, a modo de invitación para participar de la actividad usando uno de los protocolos (en este caso `Y`) del tipo de coordinación (`Q`).

3. *App2*, al aceptar la invitación, prefiere utilizar su propio Coordinador `CoordinatorB` (en lugar de `CoordinatorA`), razón por la cual *importa* al contexto `Ca` en un mensaje `CreateCoordinationContext` para interponer a `CoordinatorB`. El Servicio de Activación `ASb` de `CoordinatorB` crea su propio contexto `Cb` que contiene el mismo identificador de actividad y tipo de coordinación que `Ca`, pero con una referencia al endpoint de su propio Servicio de Registración `RSb`.

4. *App2* determina los protocolos de coordinación soportados por el tipo de coordinación `Q` y se registra para un protocolo de coordinación `Y` en el Servicio de Registración `RSb` de `CoordinatorB`, intercambiando referencias a los endpoints de *App2* y del servicio específico del protocolo `Yb`.

5. La acción de registro genera que `CoordinatorB` decida reenviar la información de registro al Servicio de Registro de `RSa` de `CoordinatorA`, intercambiando las referencias a los endpoints de los servicios `Yb` e `Ya` del protocolo específicos `Y`. Esto conforma una conexión lógica entre aquellas referencias a endpoints que el protocolo `Y` puede usar.

Las interacciones de los participantes con los Servicios de Activación y de Registración son independientes del tipo de coordinación; es decir, son horizontales y no cambian de un tipo de coordinación a otro. Por esto, las interfaces que deben implementar los participantes para estos dos tipos de interacciones son provistas por la especificación WS-Coordination. Por otro lado, las interfaces necesarias para las interacciones de protocolos específicos varían (`Y` en el ejemplo de la Fig. 6.5); por ende, la especificación WS-Coordination no las define [10].

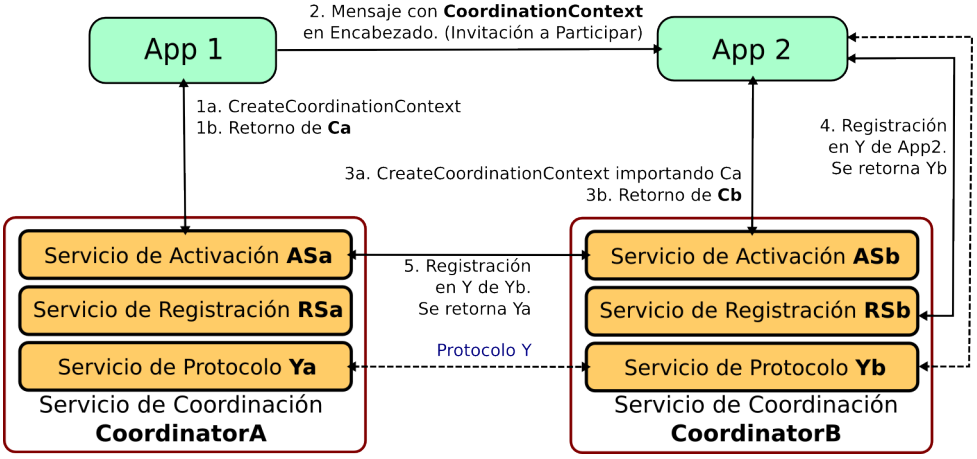

Figura 6.5: Conversación entre dos aplicaciones y sus dos servicios de coordinación [343].

Servicio de Activación

El ***Servicio de Activación*** crea una nueva actividad y retorna su contexto de coordinación. La especificación WS-Coordination define un *protocolo de activación* que respeta el patrón petición-respuesta y que tiene dos pasos [343, 92]:

1. Un servicio web, envía un mensaje **`CreateCoordinationContext`** a un *Servicio de Activación*. (Teniendo en cuenta el ejemplo de la Fig. 6.5, el proceso de activación inicial se efectúa cuando el servicio web de aplicación *App1* envía dicho mensaje al Servicio de Activación `ASa` del coordinador *CoordinatorA*, paso 1.a).

2. En respuesta, el Servicio de Activación del coordinador envía un mensaje **`CreateCoordinationContextResponse`** que contiene información del nuevo contexto de coordinación creado, elemento **`CoordinationContext`**. (En la Fig. 6.5 paso 1b., `ASa` retorna el contexto `Ca` a *App1*)

Tanto la aplicación que inicia la coordinación (*App1*), cómo el Servicio de Coordinación (*CoordinatorA*) son servicios web. Para lograr la comunicación entre ambos se utilizan mensajes SOAP, respetando la definición de las operaciones descriptas en sus definiciones WSDL respectivas (véase Sec. 5.7.1). Es por esto, que dos interfaces de servicios (`portTypes`) son definidos por la activación de una coordinación (véase Fig. 6.6). Por un lado, el servicio web participante tiene una interfaz `ActivationRequestorPortType` cuyo endpoint genera el mensaje `CreateCoordinationContextRequest`, que es enviado al endpoint de la interfaz `ActivationCoordinatorPortType` del Servicio de Activación del Coordinador. Como parte de su petición, el servidor web especifica qué tipo de coordinación quiere iniciar (en el ejemplo un tipo de coordinación `Q`) y también la referencia a su propio puerto `ActivationRequestorPortType`. En respuesta el coordinador envía el mensaje `CreateCoordinationContextRequest` al endpoint de la operación `ActivationRequestorPortType` de servicio participante [10].

Un mensaje `CreateCoordinationContext` puede contener un parámetro opcional ***`CurrentContext`*** que es el identificador de un contexto de coordinación de una actividad existente. Este parámetro causa que la actividad nueva sea una réplica de una actividad cuyo contexto existe en otro Servicio de Coordinación. En el caso del ejemplo de la Fig 6.5, en el paso 3.a y 3.b el servicio web `App2` crea en el Servicio de Activación `SAb` de su propio coordinador `CoordinatorB` un contexto de coordinación `Cb` a partir de un contexto `Ca` creado en otro Servicio de Activación. Esta opción sirve para lograr coordinación distribuida en la cual interactúan varios Servicios de Coordinación, y replicación de contextos de coordinación en forma local por razones de eficiencia.

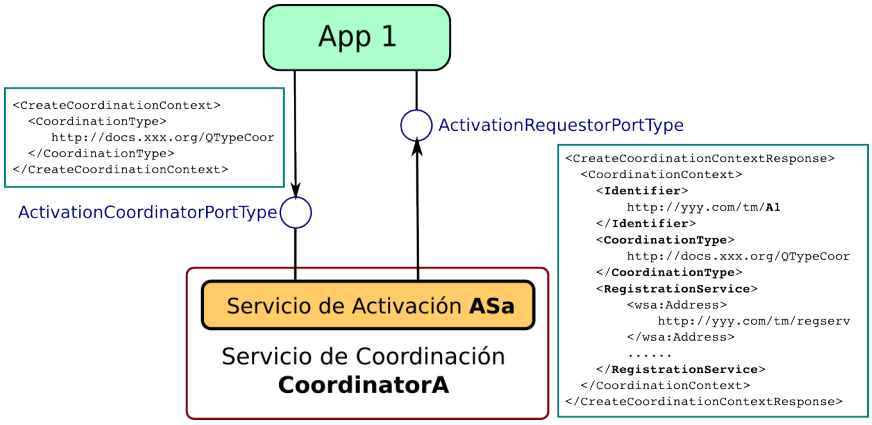

Figura 6.6: Activación o creación de un contexto de coordinación en WS-Coordination.

Servicio de Registro

Una vez que una aplicación tiene un contexto de coordinación obtenido de un Servicio de Activación de un Coordinador elegido, puede registrarse para participar de un protocolo específico de una actividad.

La fase de ***registro***, es necesaria para que el coordinador y los participantes puedan intercambiar información sobre las interfaces del protocolo específico que emplearán. Cada participante que quiere tomar parte en una conversación (es decir, ejecutar uno de los protocolos del tipo de coordinación) debe registrarse ante el Servicio de Registro del Coordinador.

En el paso previo de activación, el Servicio Coordinador provee una referencia al endpoint del Servicio de Registro en el elemento `CoordinationContext` en respuesta a la invocación `CreateCoordinationContext`. Si se toma el ejemplo de la Fig. 6.5, la activación corresponde a los pasos 1(a) y 1(b). Un servicio web, recibe este `CoordinationContext` agregado en el encabezado de un mensaje enviado, a modo de invitación, por el servicio web que originó la activación (paso 2, Fig. 6.5). Luego de esto el primer servicio web puede registrarse como participante del protocolo específico de la coordinación (paso 4, Fig. 6.5), de la siguiente forma [343, 10]:

- El servicio web que requiere la participación en la coordinación utiliza la referencia al endpoint del Servicio de Registro presente en `CoordinationContext`. Envía a ese endpoint un mensaje **`Register`**, en el cual está la referencia a su endpoint como parámetro. Este mensaje es remitido al puerto o endpoint que implementa la interfaz `RegistrationCoordinatorPortType` en el Servicio de Registro del Coordinador (véase Fig. 6.7). El mensaje `Register` contiene una referencia al endpoint del participante (`RegistrationRequestorPortType`), el nombre del protocolo específico (para el ejemplo de la Fig. 6.5, el protocolo `Y`), y una referencia a su propia interfaz de participante de dicho protocolo específico.

- En respuesta, el Servicio de Registro envia un mensaje **`RegisterResponse`** en el que se incluye una referencia al endpoint del puerto del protocolo específico en el coordinador (según la Fig. 6.5, el endpoint `Yb`). El mensaje es enviado al endpoint del participante que implementa la interfaz `RegistrationRequestorPortType`.

Luego de este proceso, ambas partes tienen las referencias a los endpoint propios del protocolo específico de la coordinación (según la Fig. 6.5, el protocolo `Y`); y por consiguiente, pueden cambiar mensajes en base al mismo. Estas referencias a endpoints pueden contener un elemento opaco **`wsa:ReferenceParameters`** para completar la calificación del endpoint del servicio que atiende al protocolo específico. Esta referencia debe hacerse de acuerdo a las reglas de la especificación WS-Addressing [201], en base al esquema `wsa="http://www.w3.org/2005/08/addressing"`. La especificación WS-Addressing se explíca más adelante en el capítulo (Sec. 6.6.1).

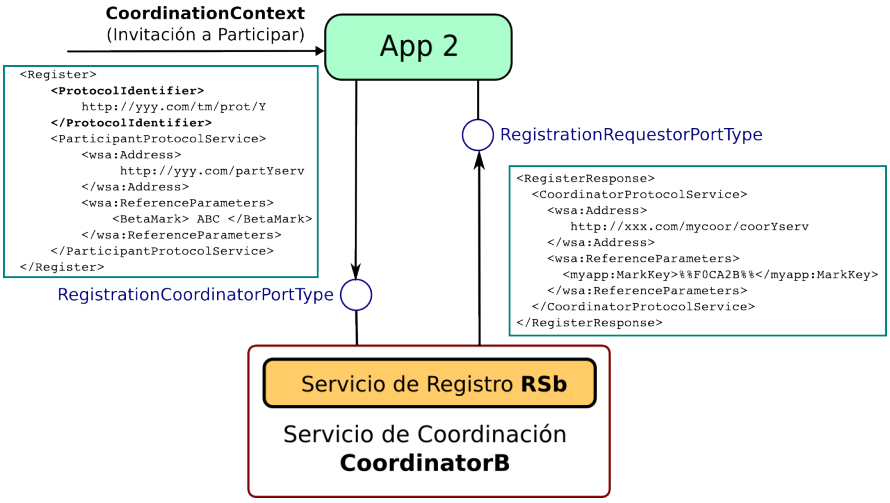

Figura 6.7: Registro en WS-Coordination.

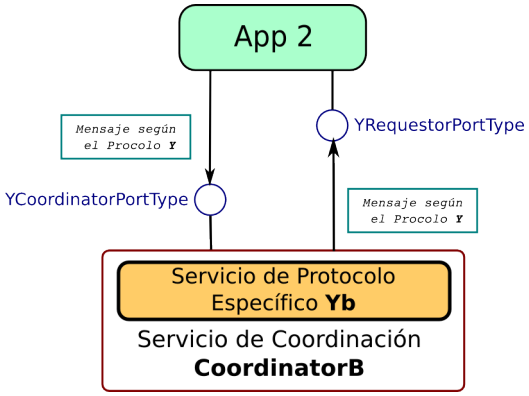

Figura 6.8: Registro en WS-Coordination.

Servicio de Protocolos Específicos

A pesar que las interfaces del protocolo específico no son definidas a través de WS-Coordination, se puede asumir que vienen de a pares. Generalmente existirá una interfaz a implementar en un puerto por cada uno de los participantes y una interfaz para un puerto en el Coordinador. Por cada protocolo específico. Según la Fig. 6.5, el protocolo específico es `Y`; por ende, existirá un puerto por cada participante (`YparticipantPortType`), y de la misma forma un puerto para el Servicio Coordinador (`YCoordinatorPortType`). Los endpoints de ambos puertos son intercambiadas entre los participantes y el coordinador en el proceso de registro (véase Fig. 6.8).

La especificación WS-Coordination no define un método o protocolo para terminar una instancia de un protocolo específico de coordinación. La terminación es específica al protocolo de coordinación, y por ende es parte de la definición de dicho protocolo. Por ejemplo, como se verá mas adelante, tanto WS-AtomicTransaction como WS-BusinessActivity plantean diferentes requerimientos a los participantes para terminar una coordinación.

6.2.2. Importancia de la Coordinación en SOA

En resumen, la especificación WS-Coordination trata cuestiones fundamentales relacionadas con la coordinación en Servicios Web en arquitecturas orientadas a servicios:

- Define *extensiones* para SOAP que son necesarias para lograr coordinación.
- Define *meta-protocolos* para crear contextos de coordinación (a través de la función de

Activación) y para ligar coordinadores con participantes de una actividad entre si (proceso de Registro)

- Define un conjunto básico de componentes de middleware y sus interfaces para implementar coordinación centralizada o distribuida.

Por esta razón, WS-Coordination incluye algunos de los ingredientes necesarios para una plataforma de Servicios Web que controle y gestione actividades o conversaciones entre servicios web; como por ejemplo el mecanismo para propagar los identificadores de conversación. Sin embargo algunos aspectos de coordinación todavía están pendientes de tratamiento. Específicamente, WS-Coordination no es un lenguaje para describir protocolos de coordinación. A pesar que asume que tales protocolos son implementados por un par de interfaces de coordinador-participante (descriptas en WSDL), no hay forma de describir, usando WS-Coordination, una secuencia de mensajes en un protocolo de coordinación [10].

Formas similares de coordinación son especificadas en las plataformas de middleware tradicionales, como CORBA. Por ejemplo, el modelo de coordinación de WS-Coordination es similar al descripto en OTS de CORBA, el cual puede verse como un coordinador de transacciones. Esto no se da por accidente, ya que este servicio de CORBA es el predecesor directo de WS-Coordination [411]. Sin embargo, OTM asume que existe un único gestor de transacciones de objetos (OTM) y todos los servicios interactuarán con éste. WS-Coordination, por su parte, permite que cada servicio web tenga su propio coordinador local, dando lugar a la adopción de los mecanismos par-a-par para lograr coordinación.

En los middleware convencionales no se trata a la coordinación como un bloque genérico de construcción. En su lugar, específicas formas de coordinación, como ser transacciones y fiabilidad, fueron tratadas en forma directa sin considerar definir e instaurar un marco común, tal como el contexto de coordinación y la funciones de activación y registro propuestas por WS-Coordination.

Un marco de trabajo para arquitecturas orientadas a servicios basados en coordinación, como el provisto por WS-Coordination, introduce una capa de control de composiciones de servicios. Se estandariza la gestión y el intercambio de la información de contexto dentro de una variedad de protocolos orientados a los negocios, como ser WS-BussinesActivity o WS-BPEL. La coordinación de servicios también alivia la necesidad de los servicios de retener información de estado; provee dicha responsabilidad a través de la gestión de la información de contexto de conversación [150].

6.3. Transacciones en Servicios Web

Como se vio en la sección anterior, WS-Coordination provee un marco de trabajo para implementar protocolos para Servicios Web. De hecho, es lógico suponer que la automatización de las interacciones entre servicios web requieran diferentes protocolos. Como se explicó en capítulos anteriores (véase Sec. 3.3), uno de los principales protocolos requeridos en las plataformas de middleware convencionales son aquellos que posibilitan el soporte a transacciones. En le contexto de Servicios Web, tales protocolos son definidos por un grupo de especificaciones WS-*.

En lo orígenes del tratamiento de esta problemática en 2002, IBM, Microsoft y BEA generaron la especificación *WS-Transaction* [90]. Luego, por la creciente complejidad que presentaba la especificación, se dividió en el dos: *WS-AtomicTransaction* [341] y *WS-BusinessActivity* [342], y se conformó en 2005 un comité responsable de su mantenimiento y gestión denominado *Committee to Advance Web Services Transaction (WS-Tx Committee)*, dentro de OASIS [340].

Si se compara el soporte de transacciones provisto por las plataformas de middleware convencionales con su contraparte en Servicios Web, se ve que los protocolos deben ser rediseñados por las ausencia de una plataforma centralizada y adaptados para trabajar con entidades de coordinación de transacciones distribuidas. Sin embargo, ésta no es la principal diferencia. De hecho, las transacciones implementadas a través de Servicios Web son generalmente de largo tiempo de ejecución. En el ejemplo de escenario explicado en la Sec. 6.1, los servicios web implementan

aplicaciones empresariales; y las transacciones que estos procesos requieren pueden exigir intervenciones humanas, y procesos de negocios completos. Por ejemplo: en ocasiones no siempre es posible, o incluso deseable, preservar las propiedades ACID; de la misma forma y rigurosidad como las exige el protocolo 2PC (véase Sec. 3.3.2).

Otra importante diferencia es la falta de un modelo prefijadom o definición, para *recursos* y para *operaciones*. Por ejemplo, en las transacciones en bases de datos hay una clara definición de los conceptos referidos a *"recurso"*, *"bloqueo"*, *"commit"* y *"roolback"*. Existe una estructura de tipos de datos bien definida y un conjunto acotado de operaciones para manipular los datos, esencialmente la creación, la inserción, actualización y eliminación de registros. Por otro lado, en Servicios Web no existe tal definición; esencialmente las operaciones descriptas en WSDL pueden representar cualquier cosa, desde la actualización a una base de datos hasta en envío de una carta a un cliente. En este caso, las operaciones de rollback pueden significar diferentes contramedidas. Por ejemplo, si se desea volver atrás el envío de una carta, posiblemente se deba remitir otra carta, solicitando que se desestime la primera. Esto hace que sea difícil caracterizar la noción de recursos, bloqueos, commits y rollbacks [10].

El enfoque de Servicios Web para resolver tales cuestiones, es flexibilizar las propiedades "rígidas" ACID promoviendo mecanismos de ***compensación*** [186, 185]. La idea básica subyacente de este enfoque consiste en permitir que los servicios web participantes pueda actualizar sus respectivos estados (o almacenamientos) persistentes después de un paso de transacción, aún si ésta no fue confirmada o acometida. Es decir, los resultados de la operación individual pueden ser visibles antes de la finalización completa de la transacción. Si por alguna razón se requiere abortar la transacción, tales resultados visibles deben ser anulados con una *operación de compensación* cuya semántica anule los efectos de la ejecución (parcial) de la transacción.

La tarea de *cancelación* o **anulación** de los efectos de un paso previo, se considera parte de la lógica de aplicación. Cada operación puede tener una diferente lógica de compensación, cada proveedor de servicio web puede entender una noción diferente acerca de una operación de anulación de una operación determinada. Por ejemplo, en el ejemplo presentado en la Sec. 6.1, compensar una operación de reserva de hotel puede involucrar la cancelación de la reserva; y en determinada situación, también puede involucrar el pago de una multa por exceder un tiempo límite de cancelación.

Sin importar la forma en que los servicios web implementen internamente las transacciones y los mecanismos de compensación, se requieren protocolos estandarizados que les permitan a cada uno de los servicios web participantes determinar el resultado de una transacción. Al efecto, se definen dos importantes tipos de protocolos para soportar transacciones en Servicios Web apoyándose en el marco de trabajo de WS-Coordination. Por un lado, para tratar a las transacciones de tiempo prolongado, las denominadas *actividades de negocios*, se establece la especificación ***WS-BussinesActivity***. Por otro lado, si se desea preservar las propiedades ACID, se estable el tipo de protocolo relacionado con las transacciones atómicas, especificado en ***WS-AtomicTransaction*** [495].

6.3.1. Relación entre las especificaciones WS-Transaction y WS-Coordination

Las especificaciones WS-Transaction definen un conjunto de protocolos que requieren coordnación entre múltiples participantes; por consiguiente, es natural basar el marco de trabajo de las transacciones en WS-Coordination. En particular, las especificaciones WS-Transaction asumen la exitencia de un conjutno de servicios web que participan de una determinada transacción y uno o más Servicios de Coordinación o Coordinadores para coordinar la transacción, ya sea en forma centralizada o descentralizada. De igual forma, las especificaciones WS-Transaction también definen la estructura del contexto de coordinación a emplearse en una transacción atómica o actividad de negocios, también define las interfaces WSDL a ser implementadas en los endpoints de los participantes o coordinadores. En definitiva, la semántica transaccional es lograda por una combinación de WS-Coordination y de las especificaciones WS-Transaction, y es ejecutada con

el soporte de los Servicios de Coordinación [10].

En el contexto de transacciones, los protocolos o servicios establecidos por WS-Coordination son utilizados en las siguientes situaciones:

- Por medio del servicio web que inicia la transacción, para crear un nuevo contexto de coordinación relacionado a una transacción.
- Por un servicio web en el rol de participante, para invitar y pasar el contexto de coordinación a otro participante.
- Por un servicio web, para registrarse en el rol de participante ante un Coordinador central de transacciones o ante su propio Servicio de Coordinación.
- Por un Coordinador intermediario o *proxy*, para vincularse con el Coordinador principal.

6.3.2. WS-AtomicTransaction

WS-Coordination es un marco de trabajo extensible para definir tipos de coordinaciones. La especificación ***WS-AtomicTransaction*** provee una definición de un tipo de coordinación que se centra en el concepto de ***transacción atómica*** *(atomic transaction)*. Si una actividad adopta este tipo de coordinación, asimila la propiedad de *"todo-o-nada"* [341]. Es decir, los protocolos de la especificación WS-AtomicTransaction definen las funcionalidades que posibilitan las transacciones a través de servicios web, compatibles con las propiedades ACID encontradas en las mayorías de plataformas de middleware tradicionales como los Monitores TP. ***ACID*** es un acrónimo que representan las características esenciales que deben preservar las transacciones "tradicionales" [150]:

- ***Atomicidad*** *(Atomicity)*: todos los cambios propuestos por la transacción son producidos o ninguno se produce. Esta característica introduce la necesidad de la operación de *rollback* que es la responsable de dehacer cualquier cambio realizado como parte de una transacción fallida, y restaurar el estado previo inicial.
- ***Consistencia*** *(Consistancy)*: ninguno de los cambios a los datos hechos como resultado de una transacción pueden violar la validez de cualquier modelo de datos relacionado. Cualquier violación genera un rollback automático en la transacción.
- ***Aislamiento*** *(Isolation)*: si varias transacciones ocurren en forma concurrente, no se deben interferir entre si. A cada transacción se le debe garantizar un ambiente de ejecución aislado.
- ***Perdurabilidad*** *(Durablility)*: luego de la terminación exitosa de una transacción, los resultados hechos por la transacción serán perdurables y no se podrán deshacer, aún si el sistema eventualmente falla.

La transacciónes atómicas al cumplir con las características ACID, significa que las acciones de la transacción, que son llevadas a cabo por distintos participantes, se consideran tentativas antes de realizar el *commit*. Es decir que los efectos de tales acciones no son persistentes ni son visibles por fuera de la transacción; sólo lo serán cuando se logre el *commit*. Cuando una aplicación termina el trabajo de una transacción, requiere que el Coordinador determine el resultado final de la transacción. El Coordinador determina si existieron fallas en la transacción solicitando a los participantes confirmaciones. Si todos los participantes confirman el éxito de la parte de la transacción que les tocaba, entonces el Coordinador confirma la persistencia de las acciones (*commit*) realizadas por los participantes. Si alguno de los participantes solicita abortar la transacción o no responde, el Coordinador solicita abortar (*abort*) todas las acciones (tentativas) llevadas a cabo por el resto de los participantes. Solicitar el `commit` de una transacción, involucra que las acciones tentativas de los participantes sean finales, de forma tal que sus efectos sean

persistentes y visibles. Por otro lado, la primitiva ***abort***, obliga a los participantes que todas sus acciones tentativas fuesen desechadas, y consideren como si nunca ocurrieron.

Las transacciones atómicas comúnmente requieren un alto grado de confianza entre los participantes de la actividad, y son generalmente de corta duración. WS-AtomicTransaction define protocolos que permite crear *"envolturas"* para los protocolos utilizados en los sistemas de procesamiento de transacciones preexistentes en una organización, y posibilita también poder interoperar a través de diferentes plataformas de software y hardware [341].

Las transacciones atómicas han probado ser extremadamente útiles para varias aplicaciones. Estas proveen consistencia ante fallas y semántica de recuperación para las mismas; de forma tal que las aplicaciones no necesitan lidiar más con los mecanismos para determinar la completitud de una serie de acciones o para eliminar los efectos colaterales de acciones no consistentes.

La especificación define protocolos que gobiernan el resultado final de transacciones atómicas. WS-AtomicTransaction tiene como objetivo, también, es un medio para desarrollar envolturas a mecanismos propietarios de transacciones y para interoperar a través de implementaciones de gestores de transacciones de diferentes empresas de software.

Contexto de WS-AtomicTransaction

WS-AtomicTransaction se construye sobre WS-Coordination, aprovechando sus Servicios de Activación y de Registro. En si, las coordinación de transacciones atómicas son un tipo de protocolo específico de coordinación y, como tal, manejan un determinado tipo de contexto; es decir, tiene un tipo especial de elemento `CoordinationContext`.

La especificación WS-AtomicTransaction agrega la siguiente semántica específica a la operación `CreateCoordinationContext` atendida por el endpoint `ActivationCoordinatorPortType` del Servicio de Activación del Coordinador (véase Fíg. 6.6):

- si el participante que solicita la activación incluye el elemento `CurrentContext`, el Coordinador receptor del mensaje es interpuesto como subordinado al Coordinador estipulado en el elemento `CurrentContext`. Es decir, el Coordinador receptor funcionará como intermediario o `proxy` hacia el Coordinador del contexto existente; de esta forma, se asegura que sólo un Servicio de Coordinación gestiona la transacción;
- si la solicitud de activación no incluye el elemento `CurrentContext`, el Coordinador receptor del mensaje crea una nueva transacción, y se establece como gestor de la misma.

El contexto de coordinación puede contener el elemento **`Expires`**. Este elemento especifica el período de tiempo, medido desde el momento en el cual el contexto de coordinación fue creado (o recibido), después del cual una transacción puede ser terminada debido únicamente a su lapso de operación. Después de cumplido este período, el Coordinador puede elegir realizar un *rollback* u omitir la remisión de *commit*.

El tipo de protocolo Transacción Atómica está definido mediante el tipo de coordinación indicado por **`http://docs.oasis-open.org/ws-tx/wsat/2006/06`**.

Protocolos de WS-AtomicTransaction

En si, WS-AtomicTransaction define un tipo de protocolo para WS-Coordination; y como se explicó en la Sec. 6.2.1, todo tipo de protocolo está compuesto por un conjunto de protocolos de coordinación. Un participante puede registrarse para uno o más de estos protocolos. En este caso, los protocolos de Transacción Atómica son:

- ***Completion***: el protocolo ***completion*** inicia el proceso de *commit*, en base al protocolo elegido por los participantes. El Coordinador que gestiona el protocolo *completion* comienza el proceso *commit* (o *abort*) con los participantes registrados para el protocolo *Volatile 2PC* y sigue con aquellos registrados para *Durable 2PC*. El resultado final es informado al participante o servicio web inicializador de la transacción.

- ***Two-Phase Commit*** *(2PC)*: El protocolo ***2PC*** coordina a los participantes registrados a alcanzar una decisión *commit* o *abort*, y asegura que todos los participantes sean informados del resultado de la decisión. En el contexto de WS-AtomicTransaction, el protocolo 2PC tiene dos variantes:
  - ***Volatile 2PC***: en este caso, los participantes manejas recursos "volátiles" como por ejemplo registros de cache o similares.
  - ***Durable 2PC***: en este caso, por el contrario, los participantes involucran recurso "durables" como registros de bases de datos.

Protocolo *Completion* Este protocolo es usado por un servicio web participante para informar al Coordinador que se desea acometer o abortar una transacción atómica. Luego que una transacción fue concluida, el resultado es retornado a la aplicación.

Si uno de los participantes de una transacción inicia este protocolo, debe registrarse usando el identificador de protocolo `http://docs.oasis-open.org/ws-tx/wsat/2006/06/Completion`. Este identificador debe ser expresado en el elemento `ProtocolIdentifier` del mensaje `Register` enviado al endpoint `RegistrationCoordinationPortType` del Coordinador (véase Fig. 6.7).

WS-AtomicTrasaction especifica que el Coordinador de una instancia del protocolo *Completion* debe ser el coordinador principal de la transacción atómica correspondiente; la tarea no puede ser llevada a cabo por ningún Coordinador *proxy* o subordinado.

En la ejecución del protocolo *Completion*, el Servicio Coordinador y el servicio iniciador del protocolo intercambian mensajes (véase Fig. 6.9). Por un lado, el Coordinador acepta los mensajes:

- **Commit**: al recibir este mensaje desde un participante, iniciador del protocolo *Completion*, el Coordinador reconoce que el iniciador ha completado el procesamiento en su aplicación; por consiguiente, el Coordinador debe iniciar el *commit* de la transacción.
- **Rollback**: al recibir este mensaje, el Coordinador reconoce que el iniciador ha terminado el proceso y debe abortar la transacción.

Como se explicará luego, el Coordinador, al recibir el mensaje *Commit* o *Rollback*, inicia las fases del protocolo 2PC, interactuando primero con los participantes registrados para el protocolo *Volatile 2PC* y luego con los registrados para *Durable 2PC*.

Por su parte, el iniciador acepta los siguientes mensajes, en base al resultado del protocolo *2PC*:

- **Committed**: al recibir este mensaje el iniciador de la instancia del protocolo, reconoce que el Coordinador ha logrado el *commit*.
- **Aborted**: al recibir este mensaje el iniciador de la instancia del protocolo, reconoce que el Coordinador ha abortado la transacción.

WS-AtomicTransaction especifica que todo Servicio de Coordinacion que soporta un Servicio de Activación debe soportar el protocolo *Completion*.

Protocolo *Two-Phase Commit (2PC)* El protocolo *Two-Phase Commit (2PC)* es un protocolo de coordinación que define la forma en la cual múltiples participantes alcanzan un acuerdo para lograr el resultado global de una transacción atómica. Como se explicó, el protocolo tiene dos variantes: *Volatile 2PC* y *Durable 2PC*.

El protocolo 2PC lo comienza el Coordinador al recibir la petición `Commit` o `Rollback` del servicio web iniciador del protocolo *Completion*. El Coordinador principal comienza la fase *prepare* para todos los participantes registrados para el protocolo *Volatile 2PC*. Todos los participantes de *Volatile 2PC* deben contestar, antes que el Coordinador comience con los mensajes `Prepare` para

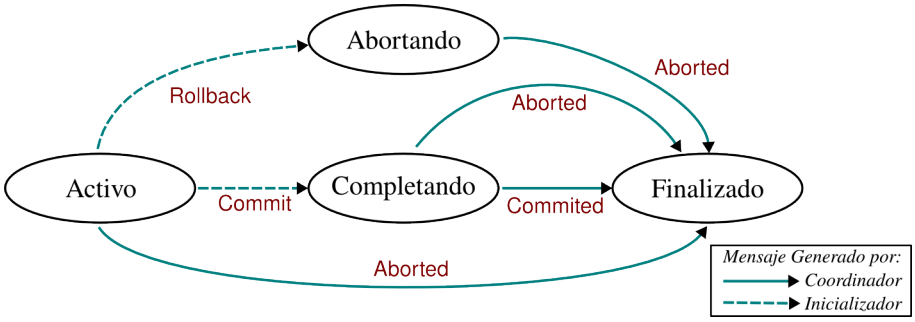

Figura 6.9: Intercambio de mensajes y estados del protocolo *Completion*.

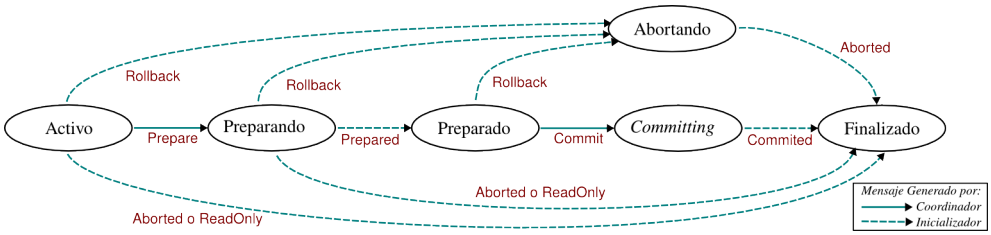

Figura 6.10: Intercambio de mensajes y estados del protocolo *2PC*.

el resto de los participantes (de *Durable 2PC*). Cuando los participantes (volátiles y durables) hayan recibido el mensaje, pueden contestar con un mensaje `Prepared`, `ReadOnly` o `Aborted` hacia el Coordinador.

Si todos los participantes han enviado el mensaje `Prepared`, el Coordinador envía el mensaje `Committed` al participante del protocolo *Completion* y el mensaje `Commit` a cada uno de los participantes de *Volatile 2PC* y de *Durable 2PC*. En la Fig 6.10 se detalla el diagrama del protocolo 2PC y de las interacciones.

Mientras que el protocolo tradicional 2PC, bastante conocido y entendido, no ha cambiado durante décadas; sí lo han hecho los ambientes para el procesamiento de actividades transacciones. En la actualidad es muy común que la infraestructura informática de una organización cuente con sistemas que son de múltiples capas o servicios. Por ejemplo, en aplicaciones web típicas, los servidores que instauran el *front-end* o nivel de presentación están separados de los servidores que implementan la capa intermedia de lógica de aplicación. A su vez, los servidores middleware *cachean* en forma adecuada el estado de las aplicaciones que soportan y acceden a los servidores *back-end* o de gestión de bases de datos, que soportan el nivel de gestión de recursos en el cual realmente se mantiene el verdadero estado de datos manejado por las aplicaciones. Para que un protocolo de transacciones soporte estas configuraciones de sistemas distribuidos, se requieren que los protocolos de coordinación cuenten con características adicionales.

Por ejemplo, para permitir "fijar" el estado actualizado de sus caches en lo servidores de base de datos en forma permanente, los servidores de capas intermedias, que soportan la lógica de aplicación, deben se notificados antes de que el protocolo 2PC comience. En la especificación WS-AtomicTrasaction esto se logra mediante el protocolo *Volatile 2PC*.

Una segunda extensión propuesta por WS-AtomicTransaction a la versión tradicional 2PC surge de la necesidad de iniciar el protocolo por fuera de los Coordinadores originales. Hecho muy común cuando varios participantes, que no comparten el mismo Coordinador y la misma base de datos, deban saber el momento en el cual el protocolo 2PC comienza. Por tal razón, se propuso el protocolo *Completion*.

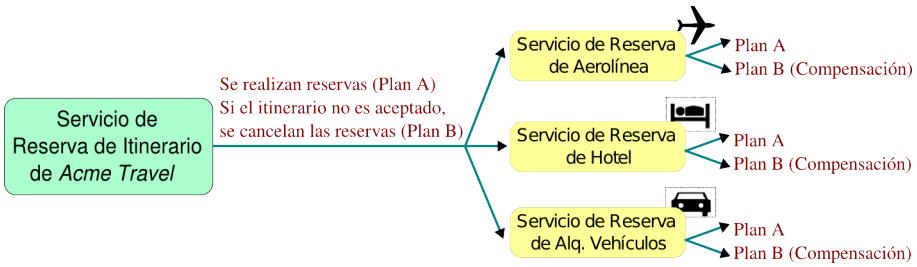

Figura 6.11: La integridad de la actividad de negocios se preserva con operaciones de *compensación*.

6.3.3. WS-BussinesActivity

Las denominadas ***actividades de negocios*** *(business activities)* son aquellas coordinaciones complejas entre servicios web, de largo tiempo de duración. Pueden pasar horas, días, y hasta incluso semanas antes que una *actividad de negocio* pueda completarse. Durante este período, la actividad puede realizar diferentes tareas que involucran a varios servicios web participantes [150, 496].

Lo que distingue a las actividades de negocios de cualquier otra actividad no-transaccional es el hecho de que sus participantes siguen reglas especificas definidas en protocolos. La principal diferencia entre los protocolos de actividades de negocios y los protocolos basados en transacciones atómicas está en la forma en que tratan las excepciones y en la naturaleza de las restricciones introducidas por las reglas de protocolos. Por ejemplo, los protocolos de Actividades de Negocios no tiene la capacidad de *rollback*; debido a la naturaleza de larga duración de las actividades de negocios, no resulta adecuado el enfoque sugerido por las transacciones de tipo ACID. En su lugar, las actividades de negocios proveen un proceso opcional de *compensación* que, puede entenderse como un "Plan B", el cual puede invocarse cuando condiciones de excepción son generadas, buscando anular o *compensar* los efectos del "Plan A" (véase Fig. 6.11).

Es importante destacar que el uso de actividades de negocios no excluye la utilización de transacciones atómicas. De hecho, puede darse que una actividad de negocios de larga duración, contenga varias transacciones atómicas durante su ciclo de vida.

Las actividades de negocios tiene las siguientes características [342]:

- Una actividad de negocios puede consumir varios recursos en un intervalo largo de tiempo.
- Pueden contener un número significativo de transacciones atómicas involucradas.
- Las tareas individuales dentro de una actividad de negocios pueden ser visibles previo a la finalización de la actividad de negocios; el resultado de dicha tarea puede impactar por fuera de los participantes de la actividad.
- El hecho de contestar a una solicitud puede tomar un tiempo prolongado. Esta situaciones se dan, por ejemplo cuando se requiere la aprobación de una persona responsable, cuando se debe completar la manufactura o la entrega de bienes; todas estas actividades deben ser completadas antes de que se logre remitir un mensaje de respuesta a la primera petición.
- En el caso que se produzca una excepción y se requiera que una actividad sea lógicamente deshecha, no es suficiente abortar la tarea o tareas que han generado la excepción. Los mecanismos de manejo de excepción pueden requerir cierta lógica de negocios. En el ejemplo explicado en la Sec. 6.1 y representado por la Fig. 6.11, cada servicio de reserva, ante la excepción de cancelación del itinerario, compensa los efectos de una reserva.
- Los participantes en una actividad de negocios pueden ser de diferentes dominios de confianza, para lo cual se necesita que todas las relaciones sean establecidas explícitamente.

La especificación ***WS-BussinesActivity*** provee la definición para dos tipos de coordinaciones relacionadas con actividades de negocios, que aplican lógica de negocios para manejar excepciones que ocurren durante la ejecución de actividades de un proceso de negocio. Las acciones en una actividad de negocios son aplicadas inmediatamente y son permanentes. Acciones de compensación pueden ser invocadas en eventos de error. WS-BusinessActivity define protocolos que permiten a los proceso de negocios existentes y a los sistemas de flujo "enmascarar" sus mecanismos propietarios e interoperar a través de los límites de confianza y de diferentes implementaciones [342].

Los protocolos de actividades de negocios definidos en la especificación WS-BusinessActivity se han centrado en los siguientes puntos de diseño:

- Todas las transiciones de estados son fielmente registradas, incluyendo el estado relacionado a la aplicación distribuida y a los metadatos del coordinador.
- Todas la notificaciones no-terminales son reconocidas por el protocolo para asegurarse una vista consistente de estado entre coordinador y participantes. Un coordinador o participante puede solicitar el estado de un participante o reenviar notificaciones para obtenerlo.
- Cada notificación está definida como un mensaje individual. El nivel de transporte petición/respuesta con sus mecanismos de reintento y tiempo agotado (*timeout*) no son suficientes para lograr el acuerdo de coordinación a nivel de aplicación (*end-to-end*) para actividades de larga latencia.

Contexto de WS-BussinesActivity

Al igual que WS-AtomicTransaction, la coordinación especificada por WS-BusinessActivity define un tipo de contexto de coordinación a través de un elemento `CoordinationContext` de WS-Coordination. Los mensajes de aplicación relacionados a la actividad de negocios deben propagarse usando este contexto de coordinación, insertado dentro de los encabezados SOAP.

El espacio de nombres XML que se debe usar para implementar interacciones basadas en WS-BusinessActivity es **`http://docs.oasis-open.org/ws-tx/wsba/2006/06`**

Protocolos de WS-BussinesActivity

Cuando algunos de los protocolos de WS-BussinessActivity es usado, el controlador de WS-Coordination asume el rol específico de Coordinador de la actividad de negocios. De igual forma que en las transacciones atómicas, el Coordinador tiene distintos tipos de control sobre la actividad, basándose en el protocolo específico usado por los participantes.

Las actividades de negocios soportan dos tipos de coordinaciones y dos tipos de protocolo. Cada uno de los tipos de coordinación pueden ser usado con cualquiera de los tipos de protocolos [342].

En el elemento `CoordinationContext` de una actividad de negocios se debe especificar si el tipo de coordinación es:

- ***AtomicOutcome*** (`http://docs.oasis-open.org/ws-tx/wsba/2006/06/AtomicOutcome`), o
- ***MixedOutcome*** (`http://docs.oasis-open.org/ws-tx/wsba/2006/06/MixedOutcome`).

Un Servicio de Coordinación para un tipo de coordinación *AtomicOutcome* debe dirigir a todos los participantes hacia el cierre o compensación de una actividad de negocios. En tanto, cuando un Servicio de Coordinación implementa el tipo de protocolo *MixedOutcome*, el Coordinador dirige a todos los participantes hacia el resultado, pero puede delegar a cada participante la posibilidad de cerrar o compensar la actividad.

Los protocolos de coordinación para un actividad de negocios pueden ser los siguientes:

- **BusinessAgreementWithParticipantCompletion**: un participante se registra para este protocolo ante su Coordinador, de forma tal que este último pueda gerenciar al primero. Este protocolo permite a cada participante determinar si se ha completado su parte de la actividad de negocio.
- **BusinessAgreementWithCoordinatorCompletion**: al igual que el anterior, un participante se registra para este protocolo ante su Servicio Coordinador; con la salvedad que el participante delega en su Coordinador la responsabilidad de decidir cuándo se han completado el trabajo exigido para la actividad de negocios.

Los participantes interactúa con el Servicio de Coordinación definido por WS-Coordination para registrarse en los protocolos determinados, de la misma forma en que se explicó en la Sec. 6.3.2.

Protocolo *BusinessAgreementWithParticipantCompletion* Durante el ciclo de vida de una actividad de negocios, el Coordinador y los servicios web participantes transitan por una serie de estados. El punto real de una transición (de un estado a otro) ocurre cuando un mensaje especial de notificación es propagado entre los servicios [150].

Por ejemplo, un participante puede indicar que ha completado un determinado procesamiento que le fue encomendado como parte de la actividad mediante la generación de una notificación de completitud. Esto cambia a los participantes de un estado *"activo"* a un estado *"completado"*. El Coordinador puede responder con un mensaje "cierre" para indicarle al participante que la actividad ha sido exitosamente completada.

Sin embargo, si la actividad de negocio no siguió el curso deseado, porque una de sus operaciones o tareas fracasó o no se logró llevar a cabo, los participantes entran en el estado de *"compensación"* durante el cual pueden tomar alguna medida para manejar la excepción. Para esto generalmente se invoca a un proceso separado de compensación que puede involucrar una serie de pasos adicionales. Una *compensación* difiere de una transacción atómica en el sentido de no pretender que los cambios realizados por los servicios participantes se vuelvan atrás (*rollback*) como que si nunca se produjeron; el propósito de una compensación es ejecutar un plan alternativo o contingente cuándo un plan original falló.

Alternativamente, se puede entrar a un estado de *"cancelación"*. Este genera la terminación de cualquier proceso futuro por fuera de las notificaciones de cancelación.

Existe otra distinción entre actividades de negocios y transacciones atómicas, es el hecho de que no se requiere que los servicios permanezcan ligados como participantes durante la ejecución de toda la actividad de negocios. Debido a que no existe un control estricto sobre los cambios realizados por los servicios participantes, estos pueden dejar la actividad de negocios inmediatamente después de haber realizado su contribución a la actividad. Cuando esto ocurre, los participantes entran en un estado de *"salida"* enviando un mensaje de notificación *"exit"* al Coordinador.

Estos y otros estados son definidos en la especificación WS-BusinessActivity en una serie de tablas. Estas tablas documentan las reglas fundamentales de los protocolos de las actividades de negocios para determinar las secuencias y condiciones permitidas de estados.

El diagrama de estado presentado en la Fig. 6.12 ilustra, en forma abstracta, el comportamiento del protocolo en las interacciones entre Coordinador y participantes. La figura demuestra los distintos estados por los cuales pueden pasar los participantes y el Coordinador de la actividad. Los participantes que se registran en el protocolo deben especificar el identificador **`http://docs.oasis-open.org/ws-tx/wsba/2006/06/ParticipantCompletion`** en el elemento `ProtocolIdentifier` en el momento de Registro [343].

El Servicio Coordinador puede aceptar los siguientes mensajes:

- **Completed**: al recibir este mensaje, el Coordinador conoce que el participante ha completado todos los procesamientos relacionados con la instancia del protocolo. Como siguiente

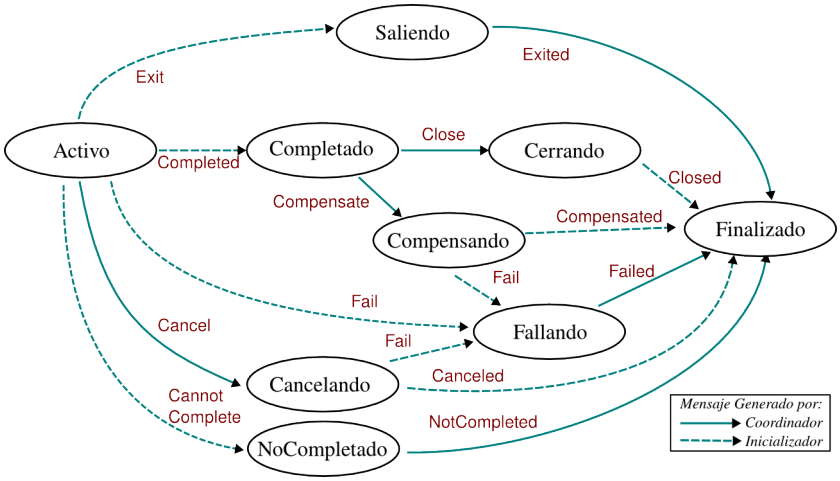

Figura 6.12: Intercambio de mensajes y estados del protocolo `BusinessAgreementWithParticipantCompletion`.

paso, el Coordinador debe enviar el mensaje `Close` o `Compensate` para indicar el resultado final de la instancia del protocolo. Luego de enviar la notificación **`Completed`**, el participante no puede participar en ninguna otra tarea de la misma actividad.

- **Fail**: al contrario del anterior, al recibir este mensaje el Coordinador conoce que el participante ha fallado; quedando indeterminado el estado del mismo. Como paso siguiente, el Coordinador debe generar la notificación `Failed`.
- **Compensated**: este mensaje, al ser generado por el participante, indica que éste se ha "olvidado" de la actividad de negocios realizada, anulado sus efectos con una compensación. Al recibir este mensaje, el Coordinador reconoce que el participante ha realizado exitosamente el procedimiento compensatorio.
- **Closed**: al igual que el anterior, el mensaje se genera por el participante, indicándole al Coordinador que se ha finalizado el proceso de coordinación exitosamente.
- **Canceled**: el participante genera esta notificación en respuesta a la solicitud de cancelación del Coordinador. De esta forma, se le indica al Coordinador que se ha cancelado todo lo relacionado con la instancia del protocolo.
- **Exit**: luego de la recepción de esta notificación, el Coordinador reconoce que el participante que la generó no participará más de la actividad; por consiguiente, toda tarea pendiente del participante fue descartada y todo trabajo realizado por éste fue debidamente cancelado.
- **CannotComplete**: luego de recibir esta notificación, el Coordinador conoce que el participante ha determinado que no puede terminar exitosamente todo el procesamiento relacionado con la instancia del protocolo. Todo trabajo pendiente a realizar por el participante fue descartado y todo trabajo realizado por el participante, cancelado. Luego del envío de este mensaje, un participante no puede intervenir en cualquier actividad relacionada con la actividad.

Por su parte, el participante puede recibir los siguientes mensajes:

- **Close**: luego de recibir esta notificación, el participante reconoce que la instancia de un protocolo está lista para ser finalizada exitosamente. Como siguiente paso, el participante debe enviar el mensaje `Closed` para finalizar definitivamente la interacción.
- **Cancel**: cuando se recibe este mensaje, un participante reconoce que el trabajo a realizarse ha sido cancelado. Como siguiente paso en el protocolo, el participante debe remitir el

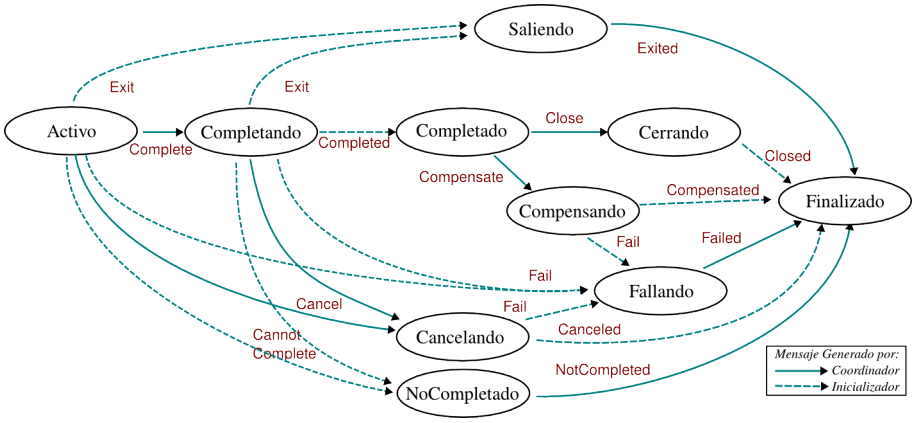

Figura 6.13: Intercambio de mensajes y estados del protocolo `BusinessAgreementWithCoordinationCompletion`.

mensaje `Canceled` o `Fail`, para indicar si le fue posible cancelar la operación o falló en el intento.

- **Compensate**: en este caso, el participante reconoce que el trabajo a realizarse debe compensarse. Como siguiente paso en el protocolo, el participante debe responder con el mensaje `Compensated` o `Fail`, indicando el éxito o no de la operación de compensación.
- **Failed**: al recibir esta notificación el participante conoce que el Coordinador es consciente de una falla en la actividad y, por ende, no se requiere más acciones por parte del participante. A partir de este momento, el participante entre en estado `Finalizado` y debe "olvidar" el contexto de la actividad.
- **Exited**: esta notificación significa que el Coordinador ha desvinculado al participante de la actividad y, por ende, no se requiere más la participación en la actividad.
- **NotCompleted**: esta notificación significa que el Coordinador ha reconocido que el participante no puede completar el procesamiento de las tareas solicitadas para la actividad y, por ende, no se requiere más la participación en la actividad.

También existen una serie de mensajes que conforman el protocolo `BusinessAgreementWithParticipantCompletion` que pueden ser aceptados por el Coordinador y los participantes, indistintamente. Por un lado el mensaje **GetStatus** requiere el estado actual del Servicio de Coordinación o de un participante. En respuesta, se genera el mensaje **Status** en el cual se indica el estado actual del servicio web que interviene en la actividad de negocios (Coordinador o participante).

Protocolo *BusinessAgreementWithCoordinatorCompletion* Este protocolo es similar al protocolo anterior, con la salvedad que los participantes confían que el Coordinador les informe cuándo recibe todas las peticiones para realizar el trabajo de las actividades de negocios [343].

Los participantes que desean registrarse utilizando el identificador de protocolo **`http://docs.oasis-open.org/ws-tx/wsba/2006/06/CoordinatorCompletion`**. Los posibles estados y mensajes partes del protocolo son descriptos en la Fig. 6.13.

6.3.4. Transacciones en SOA

Mucha de la funcionalidad transaccional implementada en las soluciones orientadas a servicios es hecha entre los componentes que ejecutan una actividad como parte de un único servicio. Sin embargo, a medida que más servicios emergen dentro de una organización y los escenarios de composiciones de servicios comienzan a ser más comunes, la necesidad de mover los límites del

manejo de transacciones hacia la interacción interservicios se ha hecho cada vez más notoria. Ser capaz de garantizar un resultado integral de una actividad, es una clave principal de la computación a nivel empresarial, y las transacciones juegan un rol importante en asegurar la calidad de dicho servicio.

Las capacidades de transacciones atómicas, no solamente conducen a la robustez en la ejecución de actividades para ambientes SOA, sino que promueven la interoperatividad cuando deben ser extendida a ambientes de integración. Esto permite que el alcance de una actividad abarque diferentes soluciones contruidas con diferentes plataformas de software, mientras aún se asegura garantiza el resultado del tipo "todo-o-nada". Asumiendo por su puesto, que la especificación WS-AtomicTransaction es soportada por las aplicaciones afectadas, esta opción amplia las aplicaciones del protocolo 2PC más allá de los límites de las plataformas de middleware tradicionales.

Las actividades de negocios complementan en forma adecuada la naturaleza composicional de las arquitecturas SOA, por la gestión y la regulación de actividades complejas; para las cuales es posible administrarlas en sus ejecuciones de largos períodos de tiempos, complementando los esenarios no abarcados por las transacciones atómicas. La autonomía e integridad de los servicios participantes es preservada al exigir la participación de los mismos en la actividad solamente por el tiempo que se los requiriere. Esto brinda la posibilidad de diseñar actividades de negocios altamente adaptables, en las cuales los participante pueden incrementar su actividad o lógica de proceso para acomodarse a los cambios en la tarea de negocios que se automatiza. Gracias al uso de los proceso de compensación, las actividades de negocios incrementan la calidad de servicio de la arquitectura SOA, incorporando una lógica para manejo de fallas.

Al igual que la especificación WS-AtomicTransaction, el soporte de múltiples productos para la extensión WS-BusinessActivity promueve la inherente interoperatividad y simplifica la integración de arquitecturas informáticas empresariales orientadas a servicios. Las actividades de negocios van un poco mas allá de los contextos transaccionales de servicios web intraempresariales, resultando una tecnología más adecuada para transacciones que abarquen servicios web provistos por empresas asociadas externas [149, 150].

6.4. Orquestación en Servicios Web

Las organizaciones que ya han realizado el paso de adquirir productos de middleware para desarrollo y soporte de Integración de Aplicaciones Empresariales (EAI) para automatizar sus procesos de negocios o integrar varios ambientes de sistemas legados, conocen muy bien el concepto de ***Orquestación*** *(Orchestration)*. En estos sistemas, un conjunto controlado y centralizado de lógicas de flujos de trabajos facilitan la interoperatividad entre dos o más aplicaciones. Una implementación común para los escenarios de orquestación son los modelos de mensajes centralizados (*hub-and-spoke* similar a los presentados en los brokers de mensajes, véase Sec. 3.6) que permite que múltiples participantes externos puedan interactuar a través de interfaces públicas con el motor central de orquestación [150].

Uno de los requerimientos esenciales que fundamenta la creación de tales soluciones es la posibilidad de contener la fusión de grandes procesos de negocios. Con la orquestación, diferentes procesos pueden ser conectados e integrados sin la necesidad de rediseñar la solución individual, que originalmente, los automatizó. La orquestación soluciona dicha problemática introduciendo una nueva capa relacionada con la lógica de flujo de trabajo. De allí, se puede concluir que el uso de orquestación puede reducir significativamente la complejidad de los ambientes de integración; por considerar que la lógica de flujos de trabajos es más fácil de mantener en forma separada que cuando se encuentra inserta en cada uno de los procesos o aplicaciones individuales.

El rol de la orquestación se amplia en los ambientes orientados a servicios. A través del uso de extensiones que permiten a la lógica del proceso de negocios ser expresada mediante servicios, la orquestación puede describir su lógica de aplicación y representarse mediante una forma

estandarizada y basada en servicios. Cuando se construyen soluciones orientadas a servicios, éstas proveen un medio atractivo para albergar o controlar la lógica que representa el proceso de negocios a ser automatizado.

La orquestación mejora la interoperatividad intrínseca buscada en los diseños orientados a servicios gracias al hecho de proveer potenciales endpoints de integración en el proceso. Un aspecto clave es entender que las orquestaciones, en el contexto de arquitecturas orientadas a servicios, son en si mismas servicios. Por consiguiente, es posible construir sobre una lógica de orquestación estandarizada las representaciones de procesos a través de toda la organización; logrando además, alcanzar un contexto de federación empresarial y promoción de la orientación a servicios.

6.4.1. WS-BPEL

Entendiendo la necesidad de lograr estándares para definir e implementar procesos de negocios a partir de Servicios Web, varias empresas de software, entre las cuales están IBM, Bea y Microsoft, desarrollaron y propusieron el *Lenguaje para Ejecución de Procesos de Negocios BPEL4WS (Business Process Execution Language for Web Services)* o simplemente *BPEL* [17]. Esta es una especificación que suplanta esfuerzos individuales previos [433] como IBM Web Services Flow Language (WSFL) [272] y la Gramática Microsoft XLANG [487], ofreciendo mayor funcionalidad y flexibilidad. La primera versión 1.0, fue liberada en julio de 2002. Porteriormente, con el aporte de otras empresas como SAP y Siebel Systems, la versión 1.1 de BPEL4WS estuvo disponible en mayo de 2003. Esta versión recibió más atención y soporte por parte de los vendedores, lográndose producir y comercializar un grupo de motores de orquestación compatibles.

Subsecuentemente, OASIS anunció formalmente el soporte a la especificación, para lo cual creó un comité técnico especial para gestionarla. A partir de allí, comenzó a llamarse ***Lenguaje de Servicios Web para la Ejecución Procesos de Negocios*** *o* ***WS-BPEL*** *(Web Services Business Process Execution Language)*, y en 2007 se publicó la versión 2.0 de la especificación [336]. WS-BPEL es la principal especificación industrial que estandariza la orquestación en Servicios Web [150].

La lógica de flujo de trabajo que es compatible con una orquestación puede consistir de un conjunto numeroso de reglas de negocios, condiciones y eventos. Colectivamente, estas partes de una orquestación establece un ***protocolo de negocios*** que define cómo los participantes pueden interoperar para lograr la culminación de una ejecución de un proceso de negocios. Los detalles de la lógica de flujo de trabajo, encapsulados y expresados por una orquestación, son contenidos dentro de la ***definición del proceso***, hecha en WS-BPEL.

Los servicios web involucrados deben ser identificados y descriptos dentro de la definición del proceso. Se debe tener en cuenta primeramente, que el proceso en si es representado como un servicio, al que se lo llamará ***Servicio de Proceso*** *(process service)*. Los demás servicios que están permitidos interactuar con el Servicio de Proceso son identificados como ***servicios asociados*** *(partner services)*. Dependiendo de la lógica de flujo de trabajo, el Servicios de Proceso puede ser invocado o invocar a otros servicios asociados externos.

WS-BPEL divide la lógica de flujo de trabajo en una serie de primitivas de actividades predefinidas. Las actividades básicas (`receive`, `invoke`, `reply`, `throw`, `wait`) representan las acciones fundamentales del flujo de trabajo, las cuales pueden ser ensambladas con la lógica proporcionada por actividades estructuradas (`sequence`, `switch`, `while`, `flow`, `pick`).

Estructura de una definición de proceso de negocios

Existen cuatro principales secciones en la definición de un proceso:

- La sección **`partnerLinks`** la cual define diferentes servicios asociados que interactúan con el proceso de negocio durante su ejecución. Cada servicio definido con un elemento **`partnerLink`** está caracterizado con un atributo `partnerLinkType` y con uno o más roles.

Esta información identifica la funcionalidad que debe ser provista por el Servicio de Proceso y por los servicios asociados para que la relación tenga éxito; es decir, los elementos `portTypes` de la definición WSDL que el proceso debe acceder y que los servicios asociados deben implementar.

- La sección **`variables`** define las variables de datos usadas en el proceso, proveyendo su definición en términos de tipos de mensajes WSDL, tipos XML Schema (simples o complejos), o elementos XML Schema. Las variables posibilitan que el proceso de negocios mantenga la información de estado entre intercambios de mensajes.

- La sección **`faultHandlers`** contiene los manejadores de fallas definiendo las actividades que deben ejecutarse en respuesta a una falla resultante de la invocación a un servicio asociado. En WS-BPEL, todas las fallas, sean internas o resultantes de la invocación a un servicio, son identificadas por un nombre calificado.

- Las *actividades* que encierran la lógica del proceso de negocio. El resto de la definición `process` contiene la descripción del comportamiento normal (exceptuando las excepciones y fallas) para el manejo del proceso de negocio. Los elementos fundamentales de ésta descripción son diferentes constructores de *actividad*.

Para explicar las principales parte de la definición de un proceso, se dá la definición WS-BPEL del ejemplo del proceso de negocios generador de itinierarios de la empresa *Acme Travel*, descripto en Sec. 6.1 (véase Cód. 6.2 a continuación). En el ejemplo, la definición describe el proceso de negocio de la empresa *Acme Travel* expuesto como servicio web. Dicho proceso de negocio invoca a tres servicios web propiedad de distintas empresas asociadas a *Acme Travel*; para el caso, las empresas son AirlineX, VehicleY y HotelZ.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Copyright (c) 2007, Sun Microsystems, Inc. All rights reserved. -->
3  <process name="TravelReservationService" targetNamespace="tres"
4          xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
5          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6          xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
7          xmlns:tns="tres" xmlns:tres="http://acmetravel.com/TravelReservationService"
8          xmlns:ares="http://www.airlineX.com/AirlineReservationService"
9          xmlns:vres="http://www.vehicleY.com/VehicleReservationService"
10         xmlns:hres="http://www.hotelZ.com/HotelReservationService"
11         xmlns:ota="http://www.opentravel.org/OTA/2003/05">
12
13    <documentation>
14       The Travel Reservation Service is an example of real-life ...
15    </documentation>
16
17    <import namespace="http://www.opentravel.org/OTA/2003/05"
18          location="OTA_TravelItinerary.xsd"
19          importType="http://www.w3.org/2001/XMLSchema"/>
20    <import namespace="http://acmetravel.com/TravelReservationService"
21          location="TravelReservationService.wsdl"
22          importType="http://schemas.xmlsoap.org/wsdl/"/>
23    <import ... />
24      ...
25    <partnerLinks>
26       <partnerLink name="Travel"
27             partnerLinkType="tres:TravelReservationPartnerLinkType"
28             myRole="TravelReservationServiceRole">
29             <documentation>
30                The partnerlink represents a client who sends an itinerarary.
31             </documentation>
32       </partnerLink>
33       <partnerLink name="Airline"
```

```
34              partnerLinkType="ares:AirlineReservationPartnerLinkType"
35              partnerRole="AirlineReservationServiceRole"
36              myRole="AirlineReservationCallbackServiceRole">
37        </partnerLink>
38        <partnerLink name="Vehicle"
39              partnerLinkType="vres:VehicleReservationPartnerLinkType"
40              partnerRole="VehicleReservationServiceRole"
41              myRole="VehicleReservationCallbackServiceRole">
42        </partnerLink>
43        <partnerLink name="Hotel"
44              partnerLinkType="hres:HotelReservationPartnerLinkType"
45              partnerRole="HotelReservationServiceRole"
46              myRole="HotelReservationCallbackServiceRole">
47        </partnerLink>
48     </partnerLinks>
49
50     <variables>
51        <variable name="CancelAirlineOut" messageType="ares:CancelAirlineOut"/>
52        <variable name="CancelAirlineIn" messageType="ares:CancelAirlineIn"/>
53        <variable name="ItineraryIn" messageType="tres:ItineraryIn"/>
54        <variable name="ItineraryOut" messageType="tres:ItineraryOut"/>
55        <variable name="ItineraryFault" messageType="tres:ItineraryFault"/>
56        <variable name="ReserveAirlineIn" messageType="ares:ReserveAirlineIn"/>
57        <variable name="AirlineReservedIn" messageType="ares:AirlineReservedIn"/>
58        . . .
59     </variables>
60
61     <correlationSets>
62        <correlationSet name="ItineraryCorrelator" properties="tres:ItineraryRefId"/>
63     </correlationSets>
64
65     <sequence name="Main">
66        <receive name="ReceiveItinerary" partnerLink="Travel"
67           portType="tres:TravelReservationPortType"
68           operation="buildItinerary" createInstance="yes" variable="ItineraryIn">
69           <correlations>
70              <correlation set="ItineraryCorrelator" initiate="yes"/>
71           </correlations>
72        </receive>
73
74        <assign name="CopyItineraryIn">
75           <copy>
76              <from variable="ItineraryIn" part="itinerary"/>
77              <to variable="ItineraryOut" part="itinerary"/>
78           </copy>
79           <copy>
80              <from variable="ItineraryIn" part="itinerary"/>
81              <to variable="ReserveAirlineIn" part="itinerary"/>
82           </copy>
83                ...
84        </assign>
85
86        <flow name="Flow1">
87           <if name="HasAirline">
88              <condition>
89               not($ItineraryIn.itinerary/ota:ItineraryInfo/ota:ReservationItems/ota:Item/ota:Air)
90              </condition>
91              <sequence>
92                 <invoke name="ReserveAirline" partnerLink="Airline"
93                    portType="ares:AirlineReservationPortType"
94                    operation="reserveAirline" inputVariable="ReserveAirlineIn">
95                    <correlations>
96                       <correlation set="ItineraryCorrelator"></correlation>
```

```
97                </correlations>
98             </invoke>
99             <pick name="Pick1">
100               <onMessage partnerLink="Airline" operation="airlineReserved"
101                       portType="ares:AirlineReservationCallbackPortType"
102                       variable="AirlineReservedIn">
103                  <correlations>
104                     <correlation set="ItineraryCorrelator" initiate="no">
105                     </correlation>
106                  </correlations>
107                  <assign name="CopyAirlineReservation">
108                     <copy>
109                        <from variable="AirlineReservedIn" part="itinerary"/>
110                        <to variable="ItineraryOut" part="itinerary"/>
111                     </copy>
112                     <copy>
113                        <from variable="AirlineReservedIn" part="itinerary"/>
114                        <to variable="ReserveVehicleIn" part="itinerary"/>
115                     </copy>
116                  </assign>
117               </onMessage>
118               <onAlarm>
119                  <for>'POYOMODTOHOM2OS'</for>
120                  <sequence name="AirlineCancelSequence">
121                     <assign name="CopyAirlineCancellation">
122                        <copy>
123                           <from>$ReserveAirlineIn.itinerary/ota:ItineraryRef</from>
124                           <to variable="CancelAirlineIn" part="itinerary"/>
125                        </copy>
126                     </assign>
127                     <invoke name="CancelAirline"
128                           partnerLink="Airline" operation="cancelAirline"
129                           portType="ares:AirlineReservationPortType"
130                           inputVariable="CancelAirlineIn"
131                           outputVariable="CancelAirlineOut"/>
132                  </sequence>
133               </onAlarm>
134            </pick>
135         </sequence>
136      </if>
137
138      <if name="HasVehicle">
139         <condition>
140        not($ItineraryIn.itinerary/ota:ItineraryInfo/ota:ReservationItems/ota:Item/
               ota:Vehicle)
141         </condition>
142         <sequence> ... </sequence>
143      </if>
144
145      <if name="HasHotel">
146         <condition>
147        not($ItineraryIn.itinerary/ota:ItineraryInfo/ota:ReservationItems/ota:Item/ota:Hotel)
148         </condition>
149         <sequence> ... </sequence>
150      </if>
151   </flow>
152   <reply name="ReturnItinerary" partnerLink="Travel"
153      portType="tres:TravelReservationPortType"
154      operation="buildItinerary" variable="ItineraryOut">
155   </reply>
156 </sequence>
157 </process>
```

Código 6.2: Definición en WS-BPEL del Proceso de Negocios de reserva de itinerario

Elemento `process`. Este elemento es la raíz de una definición WS-BPEL del proceso de negocios. Tiene un atributo **`name`** cuyo valor indica el nombre del proceso de negocios; el elemento también es usado para establecer los espacios de nombres referidos de la definición. En el Cód. 6.2 (líneas 3 a 11), se muestra el elemento `process` del proceso de reserva de itinerario de *Acme Travel*, al que se denomina `TravelReservationService`. Como toda definición, también incluye referencia a espacios de nombres que encierran descripciones a usarse en el resto del documento WS-BPEL. Los principales subelementos de `process` se explican a continuación.

Elemento `import`. Una descripción de proceso WS-BPEL depende XML-Schema y WSDL para la definición de los tipos de datos y de las interfaces de servicios. Sin embargo, las definiciones WS-BPEL pueden depender de otras construcciones como tipos, propiedades de variables, y alias definidos por un tercero, hechos utilizando las extensiones de WSDL o XML-Schema.

El elemento **`import`** es usado en una definición WS-BPEL como subelemento de `process` para declarar la dependencia a un documento XML Schema o WSDL externo a la definición en si. Cada elemento import contiene los siguientes atributos:

- **`namespace`**: indica un URI que identifica el documento que contiene las definiciones a importar.
- **`location`**: indica el documento que contiene la definiciones relevantes.
- **`importType`**: indica el tipo de documento a importar, proveyendo la URI que identifica el lenguaje que se usó para codificar el documento que se importa. En la mayoría de los casos, se refieren a documentos WSDL (`http://schemas.xmlsoap.org/wsdl/`) o a XML Schemas (`http://www.w3.org/2001/XMLSchema/`).

En el Cód. 6.2 (líneas 17 a 23), se muestra algunas de los elementos `import`. Como se puede observar, existe una importación de una definición XML Schema y otra WSDL.

Elementos `partnerLinks` y `partnerLink`. Un elemento **`partnerLink`** establece el tipo de puerto del servicio web asociado que será participante durante la ejecución del proceso de negocios. Los ***servicios asociados*** *(partner services)* pueden actuar como un cliente del proceso, responsables de invocar el Servicio de Proceso; de igual forma, un servicio asociado puede ser invocado por el Servicio de Proceso en si.

El contenido de un elemento `partnerLink` representa el intercambio de comunicación entre dos participantes, el Servicio de Proceso y otro servicio. El rol del Servicio de Proceso puede variar dependiendo de la naturaleza de la comunicación; por ende, el elemento `partnerLink` tiene los atributos **`myRole`** y **`partnerRole`** que establecen el rol del Servicio del Proceso y del servicio asociado respectivamente.

En resumen, el atributo `myRole` es usado cuando el Servicio de Proceso es invocado por un servicio asociado cliente, debido a que en esta situación el Servicio de Proceso actúa como proveedor de servicio. Por otro lado, el atributo `partnerRole` identifica el servicio asociado que el Servicio de Proceso invocará, siendo el servicio asociado el proveedor de servicio. Hay que tener en cuenta que ambos atributos, `myRole` y `partnerRole`, pueden ser usados por el mismo elemento `partnerLink` cuando se desea que el Servicio de Proceso actúe como consumidor y proveedor de servicio con el mismo servicio asociado. Esta situación es muy común cuando se establece comunicación asincrónica entre el Servicio de Proceso y sus servicios asociados.

En el Cód. 6.2 se muestran cuatro definiciones de `partnerLink`. En el primero (líneas 26 a 32) de nombre `Travel`, se indica que el Servicio de Proceso (de *Acme Travel*) puede ser invocado por un cliente externo. A tal efecto el Servicio de Proceso cumple el rol de `TravelReservationServiceRole` definido como rol propio o `myRole`. En el segundo `partnerLink` (línea 33 a 37) de nombre `Airline` representa al servicio web de la empresa de aerolíneas AirlineX asosciada a AcmeTravel, el cual será invocado en le proceso de negocios por el Servicio de

Proceso. La explicación de la definición del tercer y cuarto elemento `partnerLink` (líneas 38 a 47) es similar a la anterior, salvo que se referencia a los servicios de la empresa de alquiler de autos VehicleY y al hotel HolelZ.

En el Cód. 6.2 se puede ver que cada elemento `partnerLink` contiene un atributo `partnerLinkType`. Este elemento **`partnerLinkType`** identifica el elemento `portType` de WSDL referenciado por los elementos `partnerLink` dentro de la definición del proceso. Por consiguiente, estas definiciones están presentes en el documento WSDL de cada servicio participante, incluido el Servicios de Proceso. El Cód. 6.3 es la definición WSDL (abreviada) del servicio de reserva `AirlineReservationSoapService` de la aerolína asociada a *Acme Travel*.

En la definición WSDL del servicio, el constructor `partnerLinkType` contiene un elemento `role` por cada rol que el servicio pueda jugar, como los definidos en los atributos `myRole` y `partnerRole` en el elemento `partnerLink` de la definición WS-BPEL. Como resultado, un elemento `partnerLinkType` puede contener uno o dos subelementos `role`. Por ejemplo, el Cód. 6.3 de la definición del servicio asociado, contiene el constructor `partnerLinkType` del servicio web de la aerolinea asociada con *Acme Travel*. En el mismo, se ve el elemento `partnerLinkType` (líneas 50 a 53) que se corresponde con los mismos elementos en el Cód. 6.2 (líneas 33 a 37)

Cabe aclarar que varios elementos `partnerLink` pueden referenciar al mismo `partnerLinkType`. Esto es usual cuando un Servicio de Proceso tiene la misma relación con múltiples servicios asociados; por ende, todos esos servicios tiene el mismo valor para el elemento `portType`.

Elementos `variables` y `variable`. El Servicio de Proceso de WS-BPEL usa el constructor **`variables`** para definir las variables que almacenarán información del estado del flujo de trabajo. Mensajes y conjuntos de datos, descriptos en XML Schema, pueden se ubicados dentro de una variable para su posterior consulta y gestión durante la ejecución del proceso de negocios. Un tipo de dato, para que pueda ser asignado a un elemento `variable`, necesita ser predefinido usando alguno de los tres atributos: `messageType`, `element` o `type`.

El atributo **`messageType`** le permite a una variable contener un mensaje completo definido en un documento WSDL. Por su parte, el atributo **`element`** simplemente se refiere a un constructor básico XML Schema. Por último, el atributo **`type`** puede ser usado para representar un elemento XML Schema `simpleType`, como ser un string o entero. En el Cód. 6.2 se puede observar el constructor `variables` (líneas 50 a 59), el cual define variables que van a ser utilizadas en la definición posterior del proceso. Típicamente, una variable con un atributo `messageType` es definida de a pares, para el mensaje de petición y el de respuesta utilizados por la definición del proceso; en el ejemplo, las variables `ItineraryIn` e `ItineraryOut` (líneas 53 y 54) representan los mensajes de petición y respuesta del proceso de reserva de itinerario global.

Funciones `getVariableProperty` y `getVariableData`. WS-BPEL provee funciones incorporadas que permiten guardar o asociar información con variables a ser procesadas durante la ejecución del proceso de negocios.

- **`getVariableProperty(variable name, property name)`**: Esta función permite que valores de propiedades globales puedan ser recuperados desde las variables. Simplemente acepta el nombre de la variable y el de la propiedad como entrada y retorna el valor solicitado.

- **`getVariableData(variable name, part name, location path)`**: Debido a que las variables generalmente son usadas para gestionar información de estado del proceso de negocios, esta función es necesaria para proveer acceso de esta información a otras partes de la lógica del proceso de negocios. La función `getVariableData` tiene como parámetro obligatorio el nombre de la variable, y dos parámetros opcionales que pueden ser utilizados para especificar una parte de la variable.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright (c) 2007, Sun Microsystems, Inc. All rights reserved. -->
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:tns="http://www.airlineX.com/AirlineReservationService"
          xmlns:ota="http://www.opentravel.org/OTA/2003/05"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
          targetNamespace="http://www.airlineX.com/AirlineReservationService">
    ...
   <message name="ReserveAirlineIn">
      <part name="itinerary" element="ota:TravelItinerary"/>
   </message>
   <message name="CancelAirlineIn">
      <part name="itinerary" element="ota:ItineraryRef"/>
   </message>
   <message name="CancelAirlineOut">
      <part name="succeeded" element="ota:CancellationStatus"/>
   </message>
   ....
   <portType name="AirlineReservationPortType">
      <operation name="reserveAirline">
         <input message="tns:ReserveAirlineIn"/>
      </operation>
      <operation name="cancelAirline">
         <input message="tns:CancelAirlineIn"/>
         <output message="tns:CancelAirlineOut"/>
      </operation>
   </portType>
    ...
   <binding name="AirlineReservationSoapBinding" type="tns:AirlineReservationPortType">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="reserveAirline">
         <soap:operation soapAction="http://www.airlineX.com/AirlineReservationService/
             reserveAirline" style="document"/>
         <input> <soap:body use="literal"/> </input>
      </operation>
      <operation name="cancelAirline">
         <soap:operation soapAction="http://www.airlineX.com/AirlineReservationService/
             cancelAirline" style="document"/>
         <input> <soap:body use="literal"/> </input>
         <output> <soap:body use="literal"/> </output>
      </operation>
   </binding>
   ...
   <service name="AirlineReservationSoapService">
      <port name="AirlineReservationSoapHttpPort" binding="tns:AirlineReservationSoapBinding">
         <soap:address location="http://www.airlineX.com/AirlineReservationService"/>
      </port>
   </service>

   <plnk:partnerLinkType name="AirlineReservationPartnerLinkType">
      <plnk:role name="AirlineReservationServiceRole"
               portType="tns:AirlineReservationPortType">
      </plnk:role>
      ...
   </plnk:partnerLinkType>
</definitions>
```

Código 6.3: Una definición WSDL que contiene un constructor `partnerLinkType`

```
<faultHandlers>
    <catch faultName="SomethingBadHappened"
           faultVariable="TimesheetFault">
        ...
    </catch>
    <catchAll>
        ...
    </catchAll>
</faultHandlers>
```

Código 6.4: Ejemplo del uso de los elementos `faultHandlers`, `faultHandlers` y `catchAll`.

Elementos `faultHandlers`, `catch` y `catchAll`. El elemento **`faultHandlers`** sirve para construir bloques de gestión de manejo de excepciones que manipulan situaciones de error o fallas. Un elemento `faultHandlers` puede contener varios elementos **`catch`**, cada uno de los cuales provee las actividades que deben ejecutarse cuando se genera una falla disparando una excepción. Las fallas pueden ser generadas por el receptor de un mensaje de falla definido en WSDL, o bien, pueden ser explícitamente disparadas con el uso del elemento **`throw`**. El constructor `faultHandlers` puede culminar con un elemento `catchAll` para tener un tratamiento genérico por defecto de todos los errores manejados por las actividades. El Cód 6.4 muestra la construcción de un elemento `faultHandlers`, con subelementos `catch` y `catchAll`.

Elemento `Documentation`. Este elemento puede se agregado a cualquier constructor WS-BPEL para documentar la definición de un proceso de negocios, por medio de anotaciones orientadas a la compresión humana. En el 6.2 existe una anotación documental relacionada al proceso en si (líneas 13 a 15), y otra a un constructor `partnerLink`, (líneas 29 a 31).

Para definir la lógica del proceso de negocio en si (es decir, la parte ejecutable de la definición), se detalla el proceso a partir de ***constructores de actividad*** o simplemente ***actividades***. Cada proceso de negocio tiene una única actividad principal y las demás actividades, conforman módulos de ejecución los cuales pueden estar anidados. Según la especificación WS-BPEL 2.0 las actividades pueden ser: `sequence`, `assign`, `flow`, `pick`, `receive`, `reply`, `invoke`, `throw`, `exit`, `wait`, `empty`, `if`, `while`, `repeatUntil`, `forEach`, `scope`, `compensate`, `compensateScope`, `rethrow`, `validate`, `extensionActivity`. A continuación, solamente se detallan las más significativas.

Elemento `sequence`. El constructor de actividad **`sequence`** permite organizar una serie de actividades; de forma tal que éstas son ejecutadas en el orden secuencial definido por dicho constructor. Hay que tener en cuenta que el elemento `sequence` puede ser anidado, permitiendo asi la definición de secuencias de actividades dentro de otras secuencias. En el Cód. 6.2 tiene como actividad principal a una `sequence` (líneas 65 a 156), la cual tiene 4 pasos o actividades hijas:

1. una actividad `receive` (líneas 66 a 72), la cual tiene como finalidad recibir la solicitud de itinerario

2. a una actividad `assign` (líneas 74 a 84), la cual asigna a diferentes variables de entorno partes de los mensajes recibidos o a emitir; ya sea los mensajes de solicitud de itinerario, o de solicitud de reserva a las empresas asociadas.

3. una actividad `flow` (líneas 86 a 151), la cual encierra la vinculación concurrente con cada uno de los servicios web de las tres empresas asociadas para realizar las correspondientes reservas o cancelarlas.

4. una actividad `reply` (líneas 152 a 155), la cual devuelve el resultado del proceso global, contestando a la solicitud de itinerario original.

Elemento `receive`. El constructor de actividad **`receive`** permite establecer la información que un Servicio de Proceso espera al recibir una petición de un servicio asociado cliente externo. En este caso, el Servicio de Proceso es visto como un proveedor de servicio a ser invocado. Hay que tener en cuenta que el elemento `receive` puede ser empleado para recibir un mensaje de respuesta en un patrón asincrónico de comunicación.

El elemento `receive` contiene un conjunto de atributos, cada uno de los cuales tiene asignado un valor relacionado a la comunicación e información entrante de la petición:

- **`partnerLink`** este elemento indica el identificador del servicio asociado en el correspondiente constructor partnerLink
- **`portType`** es el elemento `portType` del Servicio de Proceso en el cual se espera recibir el mensaje de petición del servicio asociado, definido en el documento WSDL del Servicio de Proceso.
- **`operation`** es la operación del Proceso de Servicio que recibirá la petición.
- **`variable`** es la variable, definida por el constructor `variable`, en la cual se almacenará el mensaje entrante de petición.
- **`createInstance`** cuando este atributo esta con el valor "`yes`", el receptor de esta petición en particular, puede ser responsable de crear una nueva instancia del proceso de negocios.

En el Cód. 6.2 (líneas 66 a 72) el elemento `receive` es usado para aceptar la petición de intinerario del cliente externo, la cual se realiza invocando a la operación `buildItinerary` a través del portType `tres:TravelReservationPortType` del servicio web de *Acme Travel*. El mensaje de dicha petición, es contenido en la variable `ItineraryIn`, la cual fue definida como mensaje de tipo `tres:ItineraryIn`. Todas las definiciones de la operación y servicio deben estar estipuladas en el propio documento WSDL del proceso de negocios de *Acme Travel* para la construcción de itinerarios, expuesto por medio de un servicio web. La definición WS-BPEL de proceso se vincula con dicho servicio web través de la definición del vinculo `partnerLink` (Cód 6.2, líneas 26 a 32).

Elementos `assign`, `copy`, `from` y `to`. Este conjunto de elementos brinda la posibilidad de copiar valores entre variables de procesos, las cuales sirven para pasar información del proceso de una actividad a otra, y modificar su contenido a medida que el proceso se ejecuta. En si, el constructor **`assign`** encierra un constructor de actividad que puede incluir varias actualizaciones a diferentes variables, indicadas a través de los subelementos `from` y `to`. En el Cód. 6.2 (líneas 74 a 84), se utiliza el elemento `assign` para copiar la variable `ItineraryIn` a diferentes variables de entorno que serán utilizadas en las iteracciones con los servicios web asociados.

Es importante destacar que el constructor `copy` puede procesar una variedad de funciones de transferencia de datos; como por ejemplo, puede solamente copiarse una parte extraída de un mensaje a otra variable. Por su parte, los elementos `from` y `to` también pueden emplearse con los atributos opcionales `part` y `query` para permitir especificar partes o valores de variables a ser referenciados. En el Cód. 6.2 (líneas 107 a 116), se copia la parte `itinerary` de la variable `AirlineReservedIn`, a la parte homónima de la variable `ItineraryOut`.

Elementos `if`, `elseif` y `else`. Estos tres elementos permiten agregar lógica condicional a la definición del proceso, similar a los constructores homólogos en los lenguajes de programación. El elemento **`if`** establece el alcance de una lógica condicional. En su interior puede contener

varios constructores **elseif** para controlar varias condiciones especificadas a través del atributo **condition**. Cuando una condición se resuelve en `true`, las actividades definidas dentro de correspondiente elemento `if` (o `elseif`) son ejecutadas.

El elemento **else** puede ser agregado como una opción por defecto, si las demás condiciones no fueron cumplidas; como tal va como último subelemento dentro de un `if`. Hay que tener en cuenta que en especificaciones previas a WS-BPEL 2.0, los elementos que cumplian la misma misión eran los que conformaban el constructor **switch/case/otherwise**. En el Cód. 6.2 se muestra la utilización del constructor (líneas 87 a 90 y 136).

Elemento flow. Un constructor de actividad `flow` permite definir una serie de actividades que pueden ejecutarse, u ocurrir, concurrentemente y deben completarse todas para dar por terminado el bloque `flow`. Las dependencias entre las actividades dentro de una contrucción `flow` pueden ser definidas utilizando el subelemento opcional **link**.

En el Cód. 6.2 (líneas 86 a 151) el elemento `flow` es usado para encerrar la ejecución concurrente de las tres iteraciones con cada uno de los servicios asociados. En el código de ejemplo, sólo se encuentra detallada la iteración con el servicio web de la aerolínea; sin embargo, el tratamiento con el servicio web de la empresa de alquiler de autos y el del hotel presentan estructuras similares.

Elemento invoke. El constructor de actividad **invoke** identifica una operación de un servicio asociado, que se invoca por el Servicio de Proceso que ejecuta la definición WS-BPEL. El elemento `invoke` es configurado con cinco atributos principales, los cuales surgen para caracterizar a la invocación. Los atributos son:

- **partnerLink**: este atributo nombra al servicio asociado mediante su correspondiente elemento `partnerLink`.
- **portType**: es el atributo cuyo valor identifica el elemento `portType` de la definición WSDL del servicio asociado.
- **operation**: es el atributo que indica la operación del servicio asociado a la cual el Proceso de Servicio debe mandar su petición.
- **inputVariable**: encierra el valor del mensaje de entrada que será utilizado para comunicarse con la operación del servicio asociado. Debe notarse que es tratada como variable porque WS-BPEL hace referencia a un elemento `variable` con un atributo `messageType`.
- **outputVariable**: este atributo es usado cuando la comunicación es del patrón petición/-respuesta. El mensaje retornado es almacenado en una variable separada definida a través del elemento `variable`.

En el Cód. 6.2 (líneas 92 a 98) se muestra una actividad `invoke` que invoca a una operación del servicio web de la aerolínea AirlineX, para solicitar la reserva de los vuelos segun el itinerario. Se nombra a esta actividad `invoke` con el nombre `ReserveAirline`. La operación a invocar se denomina `reserveAirline` (en minúsculas); dicha operación se corresponde con la definida en el documento WSDL servicio web de la aerolínea (véase Cód. 6.3, líneas 33 a 36). Es posible hacer esta vinculación entre el Servicio de Proceso y el servicio web asociado gracias a la definición `partnerLink` ya descripta anteriormente (Cód. 6.2 líneas 33 a 36). Hay que notar también, que esta invocación tiene un parámetro de entrada, definido con el atributo `inputVariable`, caracterizado con la variable `ReserveAirlineIn`, la cual representa el mensaje SOAP que debe enviarse al servicio web asociado para realizar la reserva. En el Cód. 6.2 (línea 56) se ve la definición de esta variable la cual fue definida con el atributo `messageType`, indicanto que contiene el valor de un mensaje. Dicho mensaje es de tipo `ares:ReserveAirlineIn`, cuya definición se puede ver en el documento WSDL del servicio web de la aerolínea (Cód. 6.3, líneas 11 a 13).

También existe en el Cód. 6.2 (líneas 127 a 131) una actividad `invoke`, que encierra la invocación al servicio web de la compañía aérea para cancelar la reserva. En este caso, se hace referencia a la operación `cancelAirline`, descripta de igual forma en el documento WSDL del servicio web asociado (Cód. 6.3, líneas 37 a 41).

Elemento `pick`. El constructor de actividad **`pick`** sirve para realizar un procesamiento selectivo en base a eventos. Una actividad `pick` espera que ocurra un único evento de un conjunto de eventos posibles, para ejecutar las actividades seleccionadas al respecto. Una vez que un evento ha sido seleccionado, los otros eventos no son aceptados por la actividad `pick`; una vez que las actividades asociadas al evento seleccionado son cumplidas, se termina la actividad `pick`.

Una actividad `pick` contiene varias *"ramas"* de ejecución, cada una de las cuales encierra un par conformado por un *evento* y un *grupo de actividades* asociadas para ejecutarse en respuesta de ese evento. Los eventos descriptos en actividades `pick` pueden ser conformados de dos maneras:

- A través del subelemento **`onMessage`**, el cual tiene una semántica similar a la actividad `receive`. En este caso, el evento es la llegada de un mensaje entrante específico.
- Por medio del subelemento **`onAlarm`**, que hace referencia una alarma disparada por un tiempo de espera agotado. El tiempo de espera es especificado a través de un elemento `for` para indicar un lapso de tiempo, o de un elemento `until` en el cual se indica una fecha/hora específica.

Un elemento `pick` debe tener al menos un subelemento `onMessage`; y puede contener cualquier número de subelementos `onAlarm`. En el caso del Cód. 6.2, se establece una actividad `pick` (líneas 99 a 134) para esperar la respuesta o no de la invocación previa, hecha con la actividad `invoke` (líneas 92 a 98). Esta actividad define dos ramas de eventos diferentes. Por un lado, se establece como evento la recepción del mensaje de respuesta de la invocación anterior con el uso del elemento `onMenssage` (línea 100 a 102); en este caso se espera recibir al mensaje emitido por la operación `airlineReserved`, el cual se guardará en la variable `AirlineReservedIn`. Por otro lado, existe un evento asociado a un tiempo de espera estipulado por el elemento `onAlarm` con una duración de 20 segundos establecido con elemento `for` (líneas 118 y 119); si se cumple el tiempo (y no se activó el evento de la otra rama) se procede con dos actividades: una `assign`, en la que se prepara el mensaje de cancelación del pedido hecho con el `invoke` previo, y una actividad `invoke` en la cual se invoca la operación de cancelación de reserva en el servicio web de la aerolínea.

Elemento `reply`. A la vez que exista un elemento `receive`, debe existir un elemento `reply` cuando un intercambio sincrónico es mapeado en el proceso. El elemento **`reply`** es responsable de establecer los detalles de un mensaje de respuesta a una petición hecha por un servicio cliente asociado. Debido a que este elemento está asociado con el mismo elemento `partnerLink` que su correspondiente elemento `receive`, `reply` repite los la misma cantidad de atributos:

- **`partnerLink`**: indica el mismo `partnerLink` especificado en el elemento `receive`.
- **`portType`**: indica el mismo `portType` especificado en el elemento `receive` complementario.
- **`operation`**: indica la misma operación definida en el elemento `receive` complementario.
- **`variable`**: indica es la variable, definida por el constructor `variable`, en la cual se almacenará el mensaje de respuesta que se envía al servicio asociado.
- **messageExchange**: permite que el elemento `reply` sea explícitamente asociado con un objeto (actividad) capaz de recibir un mensaje (como puede ser un elemento `receive`).

En el Cód. 6.2 se muestra una actividad `reply` cuyo objetivo es retornar el resultado del itinerario generado en el proceso de negocios (líneas 152 a 155). Puntualmente se retorna el resultado, contenido en la variable `ItineraryOut`, producto de la invocación hecha de la operación `builtItinerary` realizada con la actividad `receive` inicial (líneas 66 a 72).

Correlaciones

Durante el ciclo de vida de una instancia de un proceso de negocio, generalmente se mantiene una o más conversaciones con servicios web asociados a la vez. Las conversaciones pueden estar basadas en una sofisticada infraestructura que correlacione los mensajes involucrados con la instancia correcta de un proceso, mediante del uso de algún tipo de identificador de conversación. De esta forma, los mensajes se encaminan automáticamente a la instancia respectiva sin necesidad de especificar cualquier otra información de correlación dentro del proceso de negocios. Sin embargo, en muchos casos las conversaciones involucran más de dos participantes, o usan protocolos de transporte livianos, que exigen que las partes o *tokens* de correlación sean incluidos directamente en los datos de aplicación que se intercambian. En tales casos, en necesario proveer mecanismos adicionales en el nivel de aplicación para corresponder conversaciones y mensajes con las instancias de procesos a las cuales están dirigidos.

La especificación WS-BPEL trata los escenarios de correlación mediante la declaración de ***conjuntos de correlaciones*** de operaciones dentro de una instancia de proceso. Un conjunto de tokens de correlación se define como un conjunto de ***propiedades*** compartidas por todos los mensajes dentro de un grupo de correlación.

Para definir conjuntos de correlaciones WS-BPEL utiliza el elemento `correlationSets`. Un elemento **`correlationSets`** puede ser declarado dentro de un elemento `process` o `scope` de forma análoga a la declaración de variables. Dentro de un elemento contenedor `correlationSets`, pueden existir varios subelementos **`correlationSet`**, que sirven para especificar cada conjunto de correlaciones en particular.

Un elemento `correlationSet` posee un atributo **`name`** para nombrar al conjunto de correlaciones, siendo dicho nombre único e irrepetible. También se provee un atributo **`properties`**, que sirve para estipular las propiedades o tokens que conforman el conjunto de correlaciones. Las *propiedades* usadas en una construcción `correlationSet` debe ser definidas usando tipos simples XML Schemas. Cada elemento `correlationSet` es un grupo determinado de propiedades que, tomadas en su conjunto, identifican una conversación. Un determinado mensaje puede transportar información que se ajusta o que inicializa a uno o más conjuntos de correlaciones.

Un elemento `correlationSet` es similar a una constante con ligadura tardía antes que a una variable. La ligadura de los valores a un conjunto de correlaciones es provocada por una operación de recepción o envío de mensajes, especialmente indicada. Un conjunto de correlaciones puede ser inicializado solamente una vez dentro del ambiente al cual pertenece. Una vez inicializado el conjunto de correlaciones debe mantener su valor, a pesar de cualquier actualización de variables. De esta forma, un conjunto de correlaciones puede ser inicializada a lo sumo una vez durante la vida de la instancia del proceso.

En conversaciones de múltiples servicios web, cada proceso participante en un intercambio correlacionado de mensajes puede ser *generador* o *seguidor* del intercambio. El proceso generador envía el primer mensaje (como parte de la operación de invocación) que comienza la conversación; y por consiguiente, establece los valores de las propiedades establecida en el elemento `correlationSet` que rotula la conversación. Todos los demás participantes son seguidores en la conversación; los cuales se ligan su propio conjunto de correlaciones al recibir un mensaje entrante que provee los valores de propiedades para el conjunto de correlaciones. Ambos tipos de participantes, generador y seguidor, marcan la primera actividad en sus respectivos grupos de actividades como aquella que inicia el conjunto de correlaciones.

Una especificación de conjunto de correlaciones puede ser usada con los constructores de actividades: `invoke`, `receive` y `reply`; también puede estar presente en las ramas de actividades

`pick` definidas con el elemento `onMessage`; y por último en las variantes de `onEvent` de los elementos `eventHandlers`.

Estas especificaciones o elementos **`correlation`** identifican los conjuntos de correlaciones por nombre y son usados para indicar cuáles conjuntos de correlaciones (y, por ende, sus correspondientes conjunto de propiedades) se producen en un mensaje enviado o recibido. El elemento `correlation` tiene un atributo `initiate` que establece si la correlación debe ser inicializada.

Luego que un conjunto de correlación es inicializado, los valores para sus propiedades deben ser idénticos para todos los mensajes en todas las operaciones que lleva el conjunto de correlaciones y se preserva dentro de su ámbito hasta que se complete. Esta *restriccion de consistencia de correlación* debe ser revisada en todos los casos, según los valores asignados al atributo `initiate`. Los valores legales del atributo son:

- **`"yes"`**: en este caso la actividad, en donde se encuentra el elemento `correlation`, debe inicializar el conjunto de correlaciones. Si el conjunto esta inicializado, se generará un error.
- **`"join"`** en este caso la actividad relacionada debe inicializar el conjunto de correlaciones sino está inicializado ya. Puede darse que se violen las restricciones de consistencia, en ese caso se generará un error.
- **`"no"`** es el valor por defecto; en este caso la actividad relacionada no debe intentar inicializar el conjunto de correlaciones. Si el conjunto no esta inicializado, entonces se generará un error.

En el Cód. 6.2 se establece un único conjunto de correlaciones (líneas 61 a 63) denominado `ItineraryCorrelator`, el cual tiene una sola propiedad `tres:ItineraryRefId`. La idea subyacente es individualizar cada construcción de itinerario (es decir, cada instancia del proceso de negocio) con un identificador. La correlación se inicializa luego de la actividad `receive`, encargada de capturar el mensaje de solicitud con construcción de itinerario (líneas 69 a 71); obsérvese que el atributo `initiate` está en `"yes"` indicando que los valores de propiedades deben ser inicializados. Luego, si se toma solamente la iteración con el servicio web de la aerolínea (primera rama de la actividad `flow` que comienza en la línea 87 y termina en la 136), se utiliza la correlación dos veces. La primera al invocar la operación `reserveAirline` (líneas 95 a 97), para lo cual debe usarse los valores de propiedades ya inicializados, por ende no se establece el atributo `initiate`, lo que significa que no se inicializa los valores respectivos. Por último, se hace uso de la correlación el la rama `onMessage` de la actividad `Pick` (líneas 103 a 106); al activar el evento de la entrada del mensaje generado por la operación `airlinereserved`, se utilizan las propiedades de la correlación ya instanciada, por eso el atributo `initiate` está en `"no"`.

6.4.2. Orquestación y SOA

Como se ha definido anteriormente, una *actividad* es un término genérico, aplicado a cualquier unidad lógica de trabajo que puede ser realizada por una solución orientada a servicios. El alcance de una orquestación puede, por consiguiente, ser clasificado como una actividad compleja y, generalmente, de largo tiempo de ejecución.

La orquestación, en la forma que se presenta en WS-BPEL, puede ser completamente usada en un contexto de trabajo gestionado por WS-Coordination mediante la incorporación del tipo de coordinación definido en WS-BussinesActivity; por ser esta especificación la que define los protocolos de coordinación diseñados para soportar actividades complejas y de largo tiempo de ejecución.

La lógica de procesos de negocios es la raíz de la soluciones automatizadas. La orquestación provee un modelo de automatización en el cual la lógica de proceso está centralizado y es ampliable y composicional a la vez. A través del uso de orquestaciones, los ambientes orientados

a servicios han llegado a ser inherentemente extensibles y adaptables. Las orquestaciones por si mismas establecen típicamente un punto común de integración para otras aplicaciones, lo cual hace que sean posibilitadoras claves para la integración y el reuso.

Estas cualidades conducen a incrementar la agilidad organizacional debido a que:

- La lógica de flujo de trabajo encapsulada por la orquestación puede ser modificada o extendida en una única ubicación central.
- El ubicar una orquestación centralizadamente puede significar facilitar la fusión y acople de diferentes procesos de negocios mediante la definición de las ligaduras que corresponden a la solución automatizada que proveerán en conjunto.
- Mediante el establecimiento de arquitecturas orientadas a servicios de gran escala, la orquestación puede soportar, a un nivel fundamental, la evolución de una diversidad de federaciones empresariales.

La orquestación es el principal ingrediente para lograr un estado de federación dentro de una organización que contiene varias aplicaciones basadas en plataformas de cómputo disímiles. Los avances en las tecnologías de middleware permiten a los motores de orquestación llegar a ser completamente integrados a los ambientes orientados a servicios. El concepto de orquestación orientada a servicios es fundacional en la Segunda Generación de Servicios Web; y tecnológicamente se ha convertido en el corazón de las arquitecturas SOA.

6.5. Coreografía en Servicios Web

En un mundo perfecto, todas las organizaciones podrían estar de acuerdo en cómo los procesos internos deberían estar estructurados, de forma tal que cuando se requiera que interoperen, se podría establecer soluciones automáticas en perfecta concordancia. A pesar que esta visión no es probable que se convierta en realidad, los requerimientos para la interoperación empresarial vía servicios web se ha incrementado, aumentando también su complejidad.

Esto es especialmente real cuando los requerimientos de interoperabilidad se extienden al dominio de colaboración, donde múltiples servicios de diferentes organizaciones necesitan trabajar juntos para lograr un objetivo común. El ***Lenguaje de Descripción para Coreografía de Servicios Web WS-CDL*** *(Web Services Choreography Description Language)* [85, 417] es una de las principales especificaciones que intentan organizar el intercambio entre varias organizaciones, o entre varias aplicaciones dentro de las organizaciones, con un énfasis en la colaboración pública. Esta es la especificación usada generalmente para representar el concepto de ***coreografía*** de servicios web.

Una importante característica de las coreografías, es que están orientadas al intercambio público de mensajes. El objetivo es establecer un tipo de colaboración organizada entre diferentes servicios los cuales representan diferentes entidades, con la diferencia que ninguna entidad necesariamente deba controlar la lógica de colaboración. La coreografía por ende, provee el potencial para establecer patrones universales de interoperatividad para tareas comunes de negocios interorganizacionales. Mientras que el énfasis de la coreografía es la integración B2B, también puede ser aplicada para permitir la colaboración entre aplicaciones de una misma empresa; sin embargo, el uso de orquestación es más común y adecuado para este requerimiento [150].

6.5.1. Conceptos

Dentro de una coreografía dada, un servicio web asume uno o más ***roles*** *(roles)* predefinidos. Esto establece qué hace el servicio web y qué puede hacer dentro del contexto de una tarea de negocios particular. Los roles pueden ser ligados a definiciones WSDL, y aquellos relacionados son agrupados categorizándose como ***participantes*** *(participants)*. Un *rol* identifica un conjunto de

comportamientos relacionados; por ejemplo el rol de "comprador" está asociado a la adquisición de bienes o servicios, y el rol "proveedor" está asociado a la provisión de bienes o servicios buscando una compensación económica. En tanto que un *participante* identifica un conjunto relacionado de roles; como por ejemplo una "empresa comercial" puede abarcar ambos roles (proveedor y comprador).

Cada acción que es planeada dentro de una coreografía puede ser entendida como una serie de intercambios de mensajes entre dos servicios. Cada intercambio potencial entre dos roles en una coreografía es, por ende, definido individualmente como una ***relación*** *(relationship)*. Consecuentemente, cada relación consiste en exactamente dos roles.

Una vez que se definen los interlocutores a través de una relación, se requiere el establecimiento de la naturaleza de la conversación. Los ***canales*** *(channels)* son los encargados de definir las característica del intercambio de mensajes entre dos roles. Para facilitar un intercambio complejo que involucran a múltiples participantes, la información relacionada a un canal puede ser pasada en un mensaje. Esto permite propagar los datos del canal entre participantes que requieran comunicarse entre si. Esta es una característica importante de la especificación WS-CDL; ya que favorece el descubrimiento dinámico de servicios e incrementa el número de potenciales participantes dentro de tareas colaborativas a gran escala.

Por último, la verdadera lógica detrás de los mensajes está encapsulada dentro de una *interacción*. Las ***interacciones*** *(interactions)* son los bloques de construcción fundamentales dentro de una coreografía. En relación con las interacciones están las *unidades de trabajo*. Las ***unidades de trabajo*** *(work units)* imponen reglas y restricciones que deben ser *adheridas* o cumplidas por una interacción para que ésta sea completada exitosamente.

Cada coreografía puede ser diseñada en una forma que promueva su reusabilidad, permitiendo ser aplicada a diferentes tareas de negocios compuestas por el mismo conjunto fundamental de acciones. Más aún, una coreografía, mediante el uso de la facilidad de importación, puede ser ensamblada a partir de módulos independientes. Cada uno de estos módulos pueden representar distintas sub-tareas y puede ser reusada por numerosas coreografías.

Finalmente, aunque una coreografía es, en efecto, una composición de un conjunto no específico de servicios cuyo objetivo es el logro de una tarea de negocios, las coreografías en si pueden ser usadas como ladrillos de construcción en una composición más grande.

6.5.2. Coreagrafía y Orquestación

Mientras ambos conceptos representan complejos patrones de intercambio de mensajes, existe una distinción fundamental que separa los términos de "orquestación" y "coreografía". Un orquestación expresa la organización específica de un flujo de trabajo; esto significa que existe una organización que es dueña y controla la lógica detrás de una orquestación (por ejemplo *Acme Travel*), aún si en el flujo de trabajo se involucran a empresas asociadas externas. Por otro lado, una coreografía no es necesariamente propiedad de una sola empresa; actúa como un patrón de intercambio comunitario usado para propósitos colaborativos por servicios web provistos por diferentes entidades.

Uno puede ver una orquestación como una aplicación específica de una coreografía. Este punto de vista es de alguna forma adecuado, con la salvedad de que alguna funcionalidad provista por las especificaciones correspondientes (WS-CDL y WS-BPEL) se sobreponen. Esto es debido a que ambas especificaciones fueron desarrolladas en forma aisladas y administradas por dos organizaciones de estandarización diferentes (W3C y OASIS, respectivamente).

Una orquestación es basada en un modelo en el cual la lógica de composición es ejecutada y controlada en forma centralizada. Una coreografía asume que no existe un propietario ni gestor de la lógica de colaboración. Sin embargo, un área de coincidencia y sobreposición en actuales extensiones de orquestación y coreografías es el hecho que las orquestaciones pueden ser diseñadas para incluir múltiples participantes de diferentes organizaciones; es decir, una orquestación puede efectivamente establecer actividades interempresariales de igual modo que las coreografías.

6.5.3. Coreagrafía en SOA

El concepto fundamental de exponer la lógica de negocios a través de servicios autónomos puede ser aplicado a cualquier ámbito de implementación. Dos servicios web dentro de la misma organización, cada uno exponiendo una función, pueden interactuar mediante un patrón básico de intercambio de mensajes para lograr completar una tarea. De igual forma, dos servicios web pertenecientes a diferentes organizaciones, cada uno exponiendo funcionalidades de negocios empresariales, pueden interactuar vía coreografía para compeltar una tarea más compleja. En ambos escenarios se involucran dos servicios, y ambos escenarios soportan una implementación de arquitecturas SOA.

La coreografía por consiguiente puede asistir al logro de un ambiente SOA a través de los límites de una organización y de sus empresas asociadas. Mientras que naturalmente soporta la composicionalidad, la reusabilidad y la extensibilidad, la tecnología de coreografía de servicios web también puede incrementar la agilidad empresarial. Las organizaciones son capaces de unirse en múltiples colaboraciones en línea. Las cuales pueden dinámicamente extenderse o incluso alterar sus procesos de negocios para integrarse con coreografías. Los servicios web participantes, al poder intercambiar la información referida a los canales coreográficos, permiten que organizaciones terceras se interesen en negocios comunes vía arquitecturas SOA.

6.6. Otras especificaciones de Segunda Generación

6.6.1. WS-Addressing

La especificación ***WS-Addressing*** [201] provee un mecanismo neutral de transporte para direccionar servicios web y mensajes. Puntualmente, la especificación define elementos XML para referenciar endpoints de servicios web y para la identificación punto-a-punto segura en mensajes. Esta especificación permite a los sistemas gestión de mensajes soportar la transmisión a través de redes que incluyen nodos intermediarios, como cortafuegos y gateways, en una forma transparente al protocolo de transporte utilizado [106].

WS-Addressing define dos constructores interoperables que están orientados a llevar información que es típicamente provista por protocolos de transporte y sistemas de mensajes. Estos constructores normalizan la información subyacente en un formato uniforme que puede ser procesado en forma independiente del protocolo de trasporte o de aplicación. Los dos constructores son las *referencias a endpoint* y los encabezados de información de mensajes o *encabezados MI* [121]

Referencias a endpoints

Ya se ha explicado que la naturaleza de bajo acoplamiento presente en la arquitecturas orientadas a servicios es gracias al uso de descripciones de servicios. En otras palabras, todo lo que se requiere para que un servicio consumidor contacte a un servicio proveedor es la definición WSDL de éste último. El documento WSDL, entre otras cosas, le suministra al servicio consumidor la dirección para contactar al servicio proveedor. Sin embargo, esta dirección explícita en el documento WSDL, no será suficiente si el consumidor necesita enviar un mensaje a una instancia específica del servicio proveedor.

En las aplicaciones web tradicionales existen diferentes formas de gestionar y propagar identificadores de sesión. La solución más común es agregar el *identificador* como un parámetro al final de un URL. Aunque sea sencilla su implementación, esta técnica resulta ser carente en seguridad y tampoco está estandarizada.

El concepto de ***Direccionamiento*** *(addressing)* introduce la idea de ***referencia a endpoint*** *(endpoint reference)*, que es una extensión usada como primera medida para identificar a una instancia particular de un servicio. Es de esperarse, que las referencias a endpoints de ser-

```
<wsa:EndpointReference>
  <wsa:Address>
     http://www.xmltc.com/railco/...
  </wsa:Address>
  <wsa:ReferenceProperties>
     <app:id>unn:AFJK323llws</app:id>
  </wsa:ReferenceProperties>
  <wsa:ReferenceParameters>
     <app:sesno>22322447</app:sesno>
  </wsa:ReferenceParameters>
</wsa:EndpointReference>
```

Código 6.5: Ejemplo del uso del `EndpointReference`.

vicio sean generadas en forma dinámica; también pueden contener un conjunto de propiedades suplementarias [150].

WS-Addressing utiliza el constructor **`EndpointReference`** para establecer referencias a endpoints. El constructor puede estar compuesto de una serie de elementos que proveen información de la interfaz del servicio web, incluyendo metadatos suplementarios, y de la identificación de la instancia particular del servicio.

Una referencia a un endpoint de servicio web, expresada por el elemento `EndpointReference`, tiene las siguientes partes o subelementos:

- **`Address`**: es el único subelemento obligatorio. Representa la *dirección* del servicio web, expresada por un URL.
- **`ReferenceProperties`**: son las *propiedades de referencia*. Un conjunto de valores de propiedades asociados con la instancia del servicio web.
- **`ReferenceParameters`**: los *parámetros de referencia* son un conjunto de valores de parámetros usados para el intercambio de mensajes.
- **`portType`**: indica el *portType* del servicio web.
- **`ServiceName`** y **`PortName`**: sirven para indicar la información específica de la interfaz del servicio web, brindando al receptor del mensaje la localización exacta de los detalles de la descripción del servicio (documento WSDL) requeridos para una contestación.
- **`policy`**: indica una *política* compatible con la especificación WS-Policy. Esta política provee de reglas y la información de comportamiento relevante a la interacción actual con el servicio.

En el Cód. 6.5 se muestra una definición de una referencia a un endpoint. Nótese que se utiliza el prefijo `wsa` para indicar el espacio de nombre definido por el XML Schema de WS-Addressing: **`http://www.w3.org/2005/08/addressing`**.

Encabezados de Información de Mensajes

Como se explicó en la Sec. 5.6.1, los mensajes SOAP tiene encabezados que sirven para el enrutado de los mensajes. Las extensiones provistas por WS-Addressing fueron ampliadas para incluir nuevos encabezados SOAP que establecen las características relacionadas al intercambio de mensajes en el mensaje en sí; comúnmente se los denominan *encabezados de información de enrutado*, *de información de mensaje* o, simplemente ***encabezados MI*** *(message information headers)*. En el Cód. 6.6 se muestra un encabezado MI.

Un encabezado MI provisto por WS-Addressing incluye los siguientes elementos:

- **`MessageID`**: es un *identificador de mensaje*; es decir, un valor que identifica unívocamente el mensaje o la retransmisión de mensaje. Sirve generalmente para propósitos de correlación de mensajes; esta información es requerida al utilizarse el endpoint de respuesta, al usarse los elementos `ReplyTo` o `FaultTo`.

- **`RelatesTo`**: define una *relación*, usada generalmente en escenarios petición/respuesta. Este encabezado es un elemento de correlación usado para asociar explícitamente el mensaje actual con otro mensaje.

- **`From`**: indica el *endpoint origen* o *remitente*. Es la referencia al endpoint (elemento `EndpointReference`) del servicio web que genera el mensaje.

- **`To`**: indica el *destino* o la dirección a la cual se destina el mensaje.

- **`ReplyTo`**: es un *endpoint de respuesta*. Es una referencia al endpoint del servicio web al cual debe enviarse la respuesta al mensaje; este elemento requiere el uso del elemento `MessageID`.

- **`FaultTo`**: indica un *endpoint de falla*; es decir, una referencia al endpoint del servicio web al cual debe enviarse cualquier notificación de falla. También requiere el uso del elemento `MessageID`.

- **`Action`**: contiene un URI que indica el propósito integral del mensajes (es el equivalente al valor estándar usado como SOAP HTTP action).

El hecho de incorporar esta información en los encabezados de los mensajes SOAP, incrementa su independencia como unidad de comunicación. Por consiguiente, los mensajes SOAP pueden ahora contener información detallada que define el comportamiento de interacción mensajes del servicio web al recibir un mensaje específico. El resultado es la obtención de un soporte estandarizado para el uso de intercambios de mensajes, que presenta alta flexibilidad y provee condiciones de ejecución adaptables y configurables.

Importancia de WS-Addressing

Históricamente, muchos de los detalles que tienen que ver en cómo una unidad de comunicación arriba al punto B luego de que fue transmitido del punto A fueron dejados bajo la responsabilidad de los protocolos individuales que manejan la capa de transporte de red. Aunque este nivel de abstracción provisto por los protocolos de transporte es conveniente para los desarrolladores, también conduce a restricciones en la forma en cómo la comunicación entre dos unidades de procesamiento logran su lógica de integración.

Los encabezados estandarizados MI, introducidos por WS-Addressing para SOAP, liberan de esta dependencia a nivel de protocolo de transporte. Estos encabezados ubican la responsabilidad para seleccionar su destino en el propio mensaje SOAP; incrementando de esta forma, la habilidad de actuar como una unidad autónoma de comunicación.

Es por esto que el direccionamiento logrado a través de especificaciones como WS-Addressing logra una importante estandarización a nivel de transporte de mensajes dentro de las arquitecturas SOA. Lo más importante es que la promoción de estándares abiertos establece un nivel de independencia de las tecnologías de transportes subyacentes a los sistemas distribuidos basados en SOA. El uso de las referencias a endpoints y los encabezados MI profundizan la inteligencia internalizada dentro de los mensajes SOAP, aumentando su autonomía.

Potenciar a los mensajes con la habilidad de autodirigir su carga útil, de la misma forma que contar con la habilidad para determinar cómo deberían comportarse los servicios web que reciben el mensaje, aumenta significativamente el potencial para que los servicios web sean intrínsecamente interoperables. Se ubica la lógica específica de la actividad dentro del mensaje y

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
                xmlns:app="http://www.xmltc.com/railco/...">
  <Header>
     <wsa:Action>http://www.xmltc.com/tls/vp/submit</wsa:Action>
     <wsa:To>http://www.xmltc.com/tls/vp/...</wsa:To>
     <wsa:From>
        <wsa:Address>http://www.xmltc.com/railco/ap1/...</wsa:Address>
        <wsa:ReferenceProperties>
           <app:id>unn:AFJK323llws</app:id>
        </wsa:ReferenceProperties>
        <wsa:ReferenceParameters>
           <app:sesno>22322447</app:sesno>
        </wsa:ReferenceParameters>
     </wsa:From>
     <wsa:MessageID>uuid:243234234-43gf433</wsa:MessageID>
     <wsa:ReplyTo>
        <wsa:Address>http://www.xmltc.com/railco/ap2/</wsa:Address>
        <wsa:ReferenceProperties>
           <app:id>unn:AFJK323llws</app:id>
        </wsa:ReferenceProperties>
        <wsa:ReferenceParameters>
           <app:sesno>22322447</app:sesno>
        </wsa:ReferenceParameters>
     </wsa:ReplyTo>
     <wsa:FaultTo>
        <wsa:Address>http://www.xmltc.com/railco/ap-err/</wsa:Address>
        <wsa:ReferenceProperties>
           <app:id>unn:AFJK323llws</app:id>
        </wsa:ReferenceProperties>
        <wsa:ReferenceParameters>
           <app:sesno>22322447</app:sesno>
        </wsa:ReferenceParameters>
     </wsa:FaultTo>
  </Header>
  <Body>
     ...
  </Body>
</Envelope>
```

Código 6.6: Ejemplo de un encabezado MI.

se promueve un diseño estándar altamente reutilizable y genérico de servicios que también facilitan el descubrimiento de metadatos de servicio. Más aún, el uso de encabezados MI aumenta el rango de lógica de interacción dentro de actividades complejas y permite que dicha lógica sea dinámicamente determinada.

Por último, las arquitecturas SOA se ven escalables en una forma estándar gracias el soporte a referencias a instancias de servicios, sin la necesidad de exigir diseños propietarios. Como efecto colateral de esta ventaja, WS-Addressing, al proveer la funcionalidad para soportar comunicación entre instancias de servicios, soporta también la creación de servicios web con información de contexto; siendo un marco para propagar y gestionar información relacionada a una interacción entre servicios web. Aquí se puede observar, la importancia de la tecnología propuesta por WS-Addressing en los contexto coordinación (WS-Coordination), de transacciones (WS-AT y WS-BA) y, en forma más general, en procesos de negocios (WS-BPEL).

6.6.2. WS-ReliableMessaging

Los beneficios de un marco de trabajo que promueva el bajo acoplamiento de mensajes exige como costo la pérdida de control sobre el proceso de comunicación. Luego que un servicio web ha transmitido un mensaje, no existe una forma inmediata de saber si el mensaje llegó exitosamente al destino previsto, si el mensaje falló al llegar y requiere una retransmisión, o si un grupo de mensajes ha llegado en la secuencia deseada [150].

La ***Confiabilidad de Mensajes*** *(Reliable messaging)* es el principio que soluciona estos requerimientos proveyendo calidad de servicio a los marcos de trabajos que gestionan actividades. La Confiabilidad de Mensajes provee garantía en la notificación de la entrega exitosa o fallida de mensajes.

La especificación ***WS-ReliableMessaging*** [337] es la que define un modelo y mecanismo para el Direccionamiento Confiable. WS-ReliableMessaging provee marco capaz de garantizar:

- que el servicio proveedor (generador del mensaje) será notificado del éxito o fracaso de la transmisión,
- que el mensaje es enviado respetando reglas específicas de secuenciado o se genera una falla en su defecto.

A pesar de que las extensiones introducidas por WS-ReliableMessage gobiernan el aspecto de las actividades o conversaciones entre servicios web, la especificación se diferencia de otras especificaciones por no requerir de un Servicio Coordinador para mantener el estado de una interacción. En su lugar, todas las *reglas de confianza* son implementadas como encabezados SOAP dentro de los mensajes en si.

WS-ReliableMessaging hace la distinción entre las partes que son responsables de iniciar la transmisión de un mensaje y aquellas que realmente realizan la transmisión. Es importante distinguir entre los términos ***"emisión"*** *(send)*, ***"transmisión"*** *(transmit)*, ***"recepción"*** *(receive)* y ***"entrega"*** *(deliver)*, para indicar cada una de las diferentes partes del proceso de envío de mensaje.

Una ***aplicación origen*** *(application source)* es el servicio web o lógica de aplicación que *envía* el mensaje al ***RM origen*** *(RM source)*, que es el procesador físico o nodo que realiza la verdadera *transmisión*, a través de la red, del mensaje. En el otro extremo de la red, está el ***RM destino*** *(RM destination)* que representa el procesador o nodo destinatario que *recibe* del mensaje; el cual, subsecuentemente, *entrega* a su ***aplicación destino*** *(application destination)*.

Secuencias

Una ***secuencia*** *(sequence)* establece el orden en el cual un grupo de mensajes deben ser entregados. Cada mensaje que es parte de una secuencia es etiquetado con un ***número de***

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
                xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <Header>
     <wsrm:Sequence>
       <wsu:Identifier>http://www.xmltc.com/railco/seq22231</wsu:Identifier>
       <wsrm:MessageNumber>12</wsrm:MessageNumber>
       <wsrm:LastMessage/>
     </wsrm:Sequence>
  </Header>
  <Body>
     ...
  </Body>
</Envelope>
```

Código 6.7: Ejemplo de un elemento sequence de WS-ReliableMessaging.

mensaje *(message number)* que identifica la posición de cada mensaje dentro de la secuencia. El mensaje final de una secuencia es rotulado con un identificador de *mensaje final (last message)*.

WS-ReliableMessaging establece un constructor **sequence** que se inserta como encabezado SOAP para indicar la ubicación del mensaje actual en relación con la secuencia global. En el Cód. 6.7 se muestra un encabezado SOAP con un elemento `sequence`, el cual tiene los subelementos: **`Identifier`** para identificar a la secuencia en si, **`MessageNumber`** para indicar el orden del mensaje en la secuencia, y **`LastMessage`** que indica que este mensaje en particular es el último de la secuencia.

Acuses de Recibo

Una parte central de la entrega confiable de mensajes es el sistema de notificación usado para comunicar condiciones entre el RM destino y el RM origen. En el común de los casos, al recibir el mensaje conteniendo el identificador de último mensaje, el RM destino emite un ***acuse de recibo de la secuencia*** *(sequence acknowledgement)*. Depende del RM origen determinar si los mensajes recibidos son iguales a los mensajes originales transmitidos. El RM origen puede retransmitir cualquiera de los mensaje perdidos, dependiendo de la *garantía de entrega* utilizada, como se explica más adelante.

La especificación WS-ReliableMessaging establece un constructor de encabezado SOAP **`SequenceAcknowledgement`** para que el RM destino notifique el éxito de la recepción de los mensajes. Este constructor, además de identificar la secuencia a través del subelemeno `Identifier`, debe indica cuáles mensajes fueron recibido y cuáles no. Para lograr esto, utiliza uno o varios elementos **`AcknowledgementRange`** que sirven para indicar, junto con los atributos **`Upper`** y **`Lower`**, el rango de mensajes que fueron recibidos. Este rango está basado en los valores de los elementos `MessageNumber` informados en cada mensaje de secuencia. Cada elemento `AcknowledgementRange` indica un rango continuo de valores de los mensajes recibidos. En el ejemplo del Cód 6.8, se puede apreciar que el acuse de recibo de secuencia indica que se recibieron correctamente los mensajes del 1 al 4, del 6 a 8, del 11 a 12 y de 14 a 15; consecuentemente no se recibieron los mensajes 5, 9, 10 y 13.

Otra alternativa es hacer que los RM destinos emitan ***acuses de recibos negativos*** *(negative acknowledgements)* que indiquen al RM origen que una condición de falla ha ocurrido. Para esto, WS-ReliableMessaging el elemento **`Nack`** como alternativa a `AcknowledgementRange`; este elemento sirve para indicar cada uno de los mensajes que no se recibieron o fallaron, en vez de indicar un rango de mensajes. En el Cód. 6.9 se muestra el encabezado de acuse de recibo negativo, alternativo al expresado en el Cód. anterior 6.8.

En determinados casos, un RM origen no necesita esperar hasta que el acuse de recibo de secuencia se genere en respuesta al último mensaje de secuencia. Puede exigirle al RM destino que

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
                xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <Header>
     <wsrm:SequenceAcknowledgement>
       <wsu:Identifier>http://www.xmltc.com/tls/seq22231</wsu:Identifier>
       <wsrm:AcknowledgementRange Upper="4" Lower="1"/>
       <wsrm:AcknowledgementRange Upper="8" Lower="6"/>
       <wsrm:AcknowledgementRange Upper="12" Lower="11"/>
       <wsrm:AcknowledgementRange Upper="15" Lower="14"/>
     </wsrm:SequenceAcknowledgement>
  </Header>
  <Body>
     ...
  </Body>
</Envelope>
```

Código 6.8: Ejemplo de un acuse de recibo de secuencia.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
                xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <Header>
     <wsrm:SequenceAcknowledgement>
       <wsu:Identifier>http://www.xmltc.com/tls/seq22231</wsu:Identifier>
       <wsrm:Nack>5</wsrm:Nack>
       <wsrm:Nack>9</wsrm:Nack>
       <wsrm:Nack>10</wsrm:Nack>
       <wsrm:Nack>13</wsrm:Nack>
     </wsrm:SequenceAcknowledgement>
  </Header>
  <Body>
     ...
  </Body>
</Envelope>
```

Código 6.9: Ejemplo de un acuse de recibo de secuencia, usando el elemento Nack.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
                xmlns:wsrm="http://schemas.xmlsoap.org/ws/2004/03/rm">
  <Header>
     <wsrm:AckRequested>
       <wsu:Identifier>http://www.xmltc.com/tls/seq22232</wsu:Identifier>
     </wsrm:AckRequested>
  </Header>
  <Body>
     ...
  </Body>
</Envelope>
```

Código 6.10: Ejemplo de un acuse de recibo de secuencia.

genere acuses por cada mensaje individual que se reciba. Para eso, WS-ReliableMessaging estipula el uso del constructor de encabezado SOAP **`AckRequested`**, el cual tiene un único subelemento `Identifier` que sirve para indicarle al RM destino la secuencia de la que se desea que se acuse el recibo de cada uno de sus mensajes. Opcionalmente, también puede agregarse un elemento `MessageNumber` para solicitar la confirmación de un mensaje en particular de la secuencia dada (véase Cod. 6.10).

Garantías de entrega

La naturaleza de una secuencia es determinada por un conjunto de reglas de confiabilidad conocidas como *garantías de entrega*. Las ***garantías de entrega*** *(delivery assurances)* son patrones de entrega de mensajes predefinidos que establecen políticas de fiabilidad. Estos patrones son de semántica similar a los expresados en la Sec. 3.2.3 para RPCs.

Las siguientes garantías de entrega son soportadas por WS-ReliableMessaging:

- ***"a los sumo una vez"*** *(AtMostOnce)*: garantiza la entrega de un solo mensaje o ninguno. Si más de una vez se entregó el mismo mensaje, se produce un error de condición.
- ***"por lo menos una vez"*** *(AtLeastOnce)*: garantiza la entrega una o varias veces del mismo mensaje. Si ninguna vez se entregó el mensaje, se produce un error de condición.
- ***"exactamente una vez"*** *(ExactlyOnce)*: garantiza la entrega una y solo una vez del mensaje. Si ninguna vez se entregó el mensaje, o si se entregó varias veces se produce un error de condición.
- ***"en Orden"*** *(InOrder)*: garantiza que la entrega de un grupo de mensajes siga estrictamente el orden estipulado por una secuencia. Si la entrega de los mensaje no respeta la secuencia se genera un error. Esta garantía de entrega puede ser combinada con alguna de las anteriores.

Importancia de la Confiabilidad de Mensajes

La especificación WS-Addressing está íntimamente relacionada con el marco de trabajo propuesto por WS-ReliableMessaging. De hecho, algunas reglas de uso establecidas por WS-Addressing para los encabezados MI fueron cambiadas para ser coherentes con la especificación WS-ReliableMessaging. En las primeras especificaciones de WS-Addressing se obligaba a que el identificador de un mensaje sea siempre distinto; sin embargo, para soportar las garantías de entregas, fue necesario contar con la posibilidad de retransmitir un mensaje con el mismo identificador de mensaje, usado en el mensaje original. Por tal motivo, fue necesario cambiar en tal sentido la especificación WS-Addressing [165].

La Confiabilidad de Mensajes le brinda a las soluciones basadas en SOA una tangible calidad de servicio. Introduce un sistema flexible que garantiza la entrega efectiva de secuencias, soportando por un sistema de reporte de fallas integral. Esto incrementa la robustez de las implementaciones de mensajes SOAP. Al mejorar la calidad en la entrega de los mensajes SOAP, colateralmente se incrementa la calidad de los canales de comunicación aplicación-a-aplicación (A2A), promoviendo el potencial para establecer integraciones interempresariales.

6.6.3. WS-Policy

Cada tarea automatizada relacionada a los negocios está sujeta a reglas y restricciones. Esta características modifican el comportamiento de los servicios subyacentes que automatizan dicha tarea. Los motivos de tales restricciones pueden ser: requerimiento reales a nivel organizativo o de negocios, la naturaleza intrínseca de los datos a intercambiar, medidas de seguridad a nivel organizacional, etc. Yendo más lejos, cada servicio o mensaje tiene una característica única que puede ser de interés para otro servicio a lo largo de su ruta. Por ejemplo: características de comportamiento, preferencias, limitaciones técnicas, características de calidad de servicios, etc. [150]. Los servicios web pueden equiparse con metadatos accesibles y públicos que describen propiedades estas. Esta información es contenida en lo que se denomina ***política*** *(policy)*.

El uso de políticas permite a los servicios web expresar varias características y preferencias, y preservarlas de cualquier implementación en particular que requiera el diseño de reglas o restricciones en forma personalizada. De esta forma, se agrega una importante capa de abstracción que permite que las propiedades de los servicios sean gestionadas independientemente [515].

El marco de trabajo ***WS-Policy*** establece extensiones que dictaminan la estructura de documentos que describen políticas; además, establece la asociación de tales políticas a recursos de la Web. El marco de trabajo está compuesto por tres especificaciones: WS-Policy [368], WS-PolicyAttachments [545] y WS-PolicyAssertions [75].

Las políticas pueden ser programáticamente accedidas para proveer a los consumidores de servicios un entendimiento de los requerimientos y restricciones de los proveedores de servicios en tiempo de ejecución. Alternativamente, las políticas pueden ser estudiadas por personas en tiempo de diseño para desarrollar consumidores de servicios capaces de interactuar con los proveedores de servicios que ayudan a describir. Recientes revisiones del marco de trabajo WS-Policy, han extendido su estructura a la descripción de políticas y su terminología asociada.

Las aserciones de políticas pueden ser categorizadas a través de *tipos de aserciones*. Un ***tipo de aserción de política*** asocia cada aserción con un documento específico XML Schema. De la misma forma que un vocabulario XML es definido en un esquema XML, los vocabularios de políticas representan una colección de tipos dentro de una política dada.

Elemento `policy`

El elemento **`Policy`** es el constructor raíz utilizado para contener varias aserciones que conforman una política. La especificación WS-PolicyAssertions provee un conjunto predefinido de elementos para describir una aserción:

- **`TextEncoding`**: especifica el uso de un formato específico de codificación de caracteres.
- **`Language`**: expresa el idioma preferido o requerido.
- **`SpecVersion`**: indica la necesidad de una versión particular de la especificación.
- **`MessagePredicate`**: indica las reglas de procesamiento de mensajes expresadas en sentencias XPath.

Estos elementos representan aserciones que pueden ser utilizadas para estructurar políticas básicas acerca de requerimientos comunes. Las aserciones de políticas pueden ser personalizadas, y otras especificaciones WS-* pueden proveer otras aserciones de política.

```
<wsp:Policy
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy">
    <wsp:ExactlyOne>
       <wsp:SpecVersion wsp:Usage="wsp:Required"
          wsp:Preference="10"
          wsp:URI="http://schemas.xmlsoap.org/ws/2004/03/rm"/>
       <wsp:SpecVersion wsp:Usage="wsp:Required"
          wsp:Preference="1"
          wsp:URI="http://schemas.xmlsoap.org/ws/2003/02/rm"/>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Código 6.11: Ejemplo de una definición de política basada la aserción `ExactlyOne`.

Cada aserción puede indicar si su uso es requerido o no, de acuerdo al valor asignado a su atributo `Usage`. Un valor "`Required`" indica que las condiciones son requeridas y deben ser cumplidas. De la misma forma, el uso del atributo `Preference` permite que una aserción establezca su importancia con respecto a aserciones del mismo tipo.

A continuación, se explican algunos ejemplos de aserciones de políticas, a modo de ejemplo de los conceptos más importantes del marco de trabajo de WS-Policy.

Elemento `ExactlyOne`. Este constructor encierra múltiples aserciones e indica que una y sólo una puede ser ser optada. En el Cód. 6.11, se muestra la definición de una política (elemento raíz `Policy`) que contiene a la aserción `ExactlyOne`; está a su vez tiene dos políticas alternativas. En este caso, las políticas alternativas tiene relación con las versiones de la especificación `WS-ReliableMessaging` que deben usarse para remitir mensajes. Esto se define con las alternativas expresadas con `SpecVersion`; en las cuales, se expresa la preferencia entre las opciones con valor del atributo `Preference`, indicando claramente la preferencia de la versión `http://schemas.xmlsoap.org/ws/2004/03/rm`.

Elemento `All`. Este elemento, introduce una regla que establece que todo lo referente a la aserción dentro del constructor debe ser satisfecho. El elemento `All`, puede ser combinado con el elemento `ExactlyOne`, por lo cual distintas colecciones de aserciones pueden ser agrupadas dentro de constructores `All`, los cuales será agrupados a su vez en constructores `ExactlyOne`. Esto indica que una política es ofrecida como un conjunto de aserciones agrupadas, pero sólo un grupo de aserciones dentro de una alternativa `All` debe ser cumplida.

En el Cód. 6.12 se muestra una extensión de los expresado en el Cód anterior 6.11. En este caso, se agrega lo relacionado al formato de codificación de texto. Agrega a las alternativas originales sendos elementos `TextEncoding`; y agrupa cada alternativa con el contructor `All`. Es posible, utilizar un constructor **`OneOrMore`** basándolo, de la misma forma, en un grupo de aserciones, al igual que el constructor `All`; con la salvedad que una (o varias) aserciones de políticas deben ser cumplidas.

Atributo `Usage`. Como se vio en los ejemplos de códigos anteriores, una cantidad de definiciones de aserciones WS-Policy pueden contener el atributo **`Usage`** para indicar si una determinada aserción es requerida o no. El atributo es una parte fundamental del marco de trabajo de WS-Policy y se aplica a la evaluación de parte o de la totalidad de las reglas de la política. Los principales valores de que puede tomar el atributo `Usage` son:

- **`Required`**: la aserción debe ser cumplida, en caso contrario se genera un error.
- **`Optional`**: la aserción puede ser cumplida o no. Ningún error se genera.
- **`Rejected`**: la aserción no es soportada.

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy">
  <wsp:ExactlyOne ID="Invoice1">
     <wsp:All>
       <wsp:SpecVersion wsp:Usage="wsp:Required"
          wsp:Preference="10"
          wsp:URI="http://schemas.xmlsoap.org/ws/2004/03/rm"/>
       <wsp:TextEncoding wsp:Usage="wsp:Required"
          Encoding="iso-8859-5"/>
     </wsp:All>
     <wsp:All ID="Invoice2">
       <wsp:SpecVersion wsp:Usage="wsp:Required"
          wsp:Preference="1"
          wsp:URI="http://schemas.xmlsoap.org/ws/2003/02/rm"/>
       <wsp:TextEncoding wsp:Usage="wsp:Required"
          Encoding="iso-8859-5"/>
     </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Código 6.12: Ejemplo de una definición de política usando el elemento `All`.

```
<Employee ...>
  <wsp:PolicyReference
     URI="http://www.xmltc.com/tls/policy1.xml#Employee1"/>
  <wsp:PolicyReference
     URI="http://www.xmltc.com/tls/policy2.xml#Employee2"/>
</Employee>
```

Código 6.13: Ejemplo de una vinculación de dos políticas con el sujeto <Employee>.

- **Observed**: la aserción se aplica a todos los sujetos de la política.
- **Ignored**: la aserción será intencionalmente ignorada.

Atributo `Preference`. Las aserciones de políticas pueden ser categorizadas según el criterio de orden de este atributo. Esto tiene especial relevancia si un servicio web es lo suficientemente flexible para proveer varias alternativas de políticas para diversos servicios consumidores potenciales. Al atributo `Preference` se le asigna un número entero. El valor más alto es el más preferido en la aserción. El valor del atributo por defecto, cuando no se usa, es "0".

Atributo `PolicyReference`. Ya se ha explicado cómo construir un documento que describe políticas. También es importante describir la forma en que una política puede ser asociada a los *sujetos* que la aplican. Una política puede ser asociado a un servicio web, a un mensaje o a otro recurso; en todo caso, esta asociación define el ***sujeto*** *(subject)* de la política. Debido a que una política puede tener más de un sujeto, la colección de sujetos asociados a la política se denomina ***alcance*** o ***ambiente*** *(scope)* de la política.

El elemento **`PolicyReference`** contiene un atributo **`URI`** que apunta a un documento de política o a una aserción dentro del documento. El atributo **`ID`** del elemento `Policy` (constructor raíz del documento de política) es referenciado mediante el valor agregado luego del símbolo "#" en el URI (véase Cód. 6.13).

Si se usan múltiples elementos `PolicyReference` dentro de un mismo elemento sujeto, los documentos de políticas son fusionados en tiempo de ejecución. Cabe aclarar, que los elementos `PolicyReference` pueden estar presentes en los constructores `Policy`, permitiendo así la creación de módulos reutilizables de políticas.

```
<wsp:PolicyAttachment>
  <wsp:AppliesTo>
     <wsa:EndpointReference
      xmlns:emp="http://www.xmltc.com/tls/employee">
      <wsa:Address> http://www.xmltc.com/tls/ep1</wsa:Address>
      <wsa:PortType>
         emp:EmployeeInterface
      </wsa:PortType>
      <wsa:ServiceName>
         emp:Employee
      </wsa:ServiceName>
     </wsa:EndpointReference>
  </wsp:AppliesTo>
  <wsp:PolicyReference
     URI="http://www.xmltc.com/EmployeePolicy.xml"/>
</wsp:PolicyAttachment>
```

Código 6.14: Ejemplo del uso de `PolicyAttachment`.

Elemento `PolicyAttachment`. Otra forma de asociar una política con un sujeto es a través del uso de constructores **`PolicyAttachment`**. Un constructor `PolicyAttachment` tiene dos importantes subelementos. Por un lado, un elemento **`AppliesTo`** que encierra varios subelementos sujetos de la política. Por otro lado, un elemento `PolicyReference` (ya explicado previamente) que hace referencia a la política que adoptan los sujetos especificados en `AppliesTo`. En el Cód. 6.14 se muestra un ejemplo del uso de `PolicyAttachment`: en dicho ejemplo la aserción especificada por `PolicyReference` está asociada con una referencia a un endpoint de servicio.

Importancia de WS-Policy

El marco propuesto por WS-Policy puede ser utilizado en otros marcos de trabajos de otras especificaciones. En si, WS-Policy es transversal a varias especificaciones de Segunda Generación.

Cuando un Servicio Coordinador que implementa WS-Coordination genera la información de contexto para servicios participantes de una conversación, puede hacer que la distribución de la información de contexto sea validada con credenciales u otras formas de políticas de información. Para asegurar el cumplimiento de estos requerimientos, WS-Coordination puede incorporar reglas establecidas en políticas descriptas en WS-Policy.

Las políticas pueden ser aplicadas a casi cualquier sujeto que sea parte de una orquestación o coreografía. Por ejemplo, una política puede establecer varios requerimientos para los servicios asociados de una orquestación o para los participantes de una coreografía.

La especificación WS-ReliableMessaging depende del marco de trabajo propuesto por WS-Policy para implementar las garantías de entregas. Esto se logra adjuntando políticas de garantías a los mensajes que toman parte en un intercambio.

Los principios sustentados por WS-Policy tienen un alto impacto en la filosofía de orientación a servicios. Si una arquitectura SOA es una ciudad, las políticas son sus leyes, regulaciones y normas que coordina el orden entre sus habitantes. Las políticas son un requerimiento para construir ambientes orientados a servicios a nivel interempresarial. Las políticas proveen los medios para establecer restricciones, reglas y normas para la comunicación relacionada a cualquier faceta de interacción de servicios. Como resultado, las políticas mejoran la calidad que se requiere en colecciones de servicios bajamente acoplados.

Las políticas permiten a los servicios expresar mucho más de ellos mismos, yendo más allá del formato de datos y de los mensajes establecido en definiciones WSDL. Las políticas también posibilitan que los servicios abarquen un rango de metadatos, preservando aún su respectiva independencia.

El uso de políticas incrementa la calidad de servicios de las arquitecturas SOA al nivel de

restringir la transmisión de mensajes válidos a aquellos que conforman un politica de reglas y requerimientos. Un beneficio adicional de insertar restricciones a nivel de endpoint, es que la lógica de aplicación de los servicios subyacentes no requieren que se realice gestión propietaria de manejo de excepciones para tratar el envío de mensajes inválidos.

Las políticas naturalmente incrementan la habilidad de los servicios de alcanzar mejores niveles de interoperación debido a que mucho de la información de los endpoints de los servicios web puede ser expresada y publicada. Por último, al permitir contratos entre servicios mucho más ricos y expresivos, las políticas son una puerta abierta al descubrimiento y ligaduras dinámicas entre servicios web.

6.6.4. WS-Security

Los requerimientos de seguridad, al igual que en cualquier área relacionadas con las tecnologías de la información, deben ser cubiertos en las aplicaciones orientadas a servicios; principalmente aquellos relacionados con la protección de la información y la autenticación de acceso. Sin embargo, el marco de trabajo propuesto por la comunicación en base a mensajes SOAP, enfatiza aspectos particulares de seguridad que necesitan ser tenidos en cuenta por un marco de seguridad especialmente diseñado para Servicios Web.

Una familia de extensiones sobre seguridad, basadas en la especificación ***WS-Security***, componen dicho marco. En el presente trabajo, solamente se hará referencia a tres de ese grupo de especificaciones, las que conforman el núcleo principal del marco: WS-Security [339], XML-Signature [412] y XML-Encryption [141].

Cabe aclarar que WS-Security no sólo presenta un marco de trabajo, sino que es también una especificación que define elementos de lenguajes. En este sentido, la información atinente a seguridad es expresada en bloques de encabezados SOAP, y su principal constructor es el elemento `Security`.

Elemento `Security`. Este constructor representa el bloque de encabezado fundamental provisto por WS-Security. El elementos **`Security`** puede tener una variedad de subelementos, que abarcan desde constructores de XML-Encryption y de XML-Signature hasta los elementos o *tokens* provistos por la especificación WS-Security en si. Los elementos `Security` pueden ser equipados con atributos `role` que se corresponde con los roles que deben cumplir los nodos en la ruta de un mensaje SOAP, como se explicó en la Sec. 5.6.2. Esto permite agregar múltiples bloques `Security` a un mensaje SOAP, cada uno destinado a diferentes receptores en la ruta de mensaje.

Para entender las diferentes aristas que presenta de seguridad en Servicios Web, se explicarán a continuación las problemáticas de identificación, autenticación, autorización, confidencialidad e integridad en Servicios Web.

Identificación, Autenticación y Autorización.

Para que un servicio consumidor acceda a un proveedor seguro de servicios, debe suministrar información que exprese su origen o propiedad. A este hecho comúnmente se lo denomina *proclama* de ***identificación***. Una ***proclama*** *(claim)* es representada por información relacionada con la identidad del servicio consumidor almacenada en un encabezado de un mensaje SOAP. La especificación WS-Security establece un bloque de encabezado estándar para almacenar la información de una proclama de identidad.

Por su parte, la ***autenticación*** requiere que un mensaje entregado a un determinado receptor pueda probar que, de hecho, proviene de emisor que proclama venir. En otras palabras, el servicio debe dar pruebas que la proclamada de identidad sea genuina.

Una vez que el mensaje es autenticado, el receptor del mensaje puede necesitar determinar si el emisor del mismo tenga permisos o privilegios para emitirlo. Es decir, que el servicio emisor

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Header>
    <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
      <wsse:UsernameToken>
        <wsse:Username>rco-3342</wsse:Username>
        <wsse:Password Type="wsse:PasswordDigest">
          KE6QugOpkPyT3EoOSEgT30W4Keg=
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

Código 6.15: Ejemplo del uso del constructor `Security`.

del mensaje pueda acceder al servicio web proveedor y tenga privilegios para invocar a sus operaciones. Este proceso se llama ***autorización***.

Para agregar información referente a autenticación y autorización en un bloque de encabezado `Security`, se utilizan los elementos `UsernameToken`, `Username` y `Password`, definidos en el vocabulario de WS-Security. El elemento **`UsernameToken`** provee un constructor que puede ser usado para albergar información para propósitos de autenticación y autorización; generalmente tiene dos subelementos, **`Username`** (para indicar el nombre de usuario) y **`Password`** (para indicar la clave de seguridad); sin embargo pueden agregarse otros elementos de autenticación. Por otro lado, el elemento **`BinarySecurityToken`** sirve para contener tokens almacenados en datos binarios, como pueden ser certificados RSA. También se puede hacer referencia a un token de seguridad que está fuera del documento SOAP del mensaje; para esto, se utiliza el elemento **`SecurityTokenReference`**.

A los efectos de ejemplificar el uso de los elementos básicos de WS-Security para autenticación y autorización, se presenta el Cod. 6.15. En este Cód., se muestra un encabezado `Security`, el cual tiene un único subelemento `UsernameToken`. Este tiene a si vez, sólo los dos únicos subelementos exigidos `Username` y `Password`. En el caso del subelemento `Password`, tiene un atributo `Type` con valor `wsse:PasswordDigest`; esto significa que la cadena de carácteres encerrada en el elemento es el valor hash de la clave calculado por el algoritmo SHA1.

Un gran desafío es permitir que la autenticación y autorización, dentro de arquitecturas SOA, puedan propagar información de un servicio consumidor a través de múltiples servicios proveedores detrás del servicio web inicial que recibió la primera petición segura. Los servicios web al ser autónomos e independientes entre si por naturaleza, requieren de la existencia de un mecanismo que logre persistencia del contexto de seguridad establecido después que un servicio web consumidor haya sido autenticado. De otra manera, el servicio consumidor deberá re-autenticarse en cualquier petición subsecuente; aún si esta forma parte de una conversación o actividad entre servicios web.

El concepto de ***firma única*** *(single sign-on)* se enfoca en esta cuestión. El uso de la tecnología de *firma única* permite a un servicio consumidor ser autenticado una única vez y mantener su información de contexto de seguridad para compartir con otros servicios, a los cuales el servicio consumidor puede acceder sin necesitar otra autenticación. Existen tres principales extensiones que soportan la implementación del concepto de firma única: SAML (Security Assertion Markup Language) [338], .NET Passport [306] y XACML (XML Access Control Markup Language) [330].

Tomando a SAML como ejemplo, se debe decir que soporta un sistema centralizado de autenticación y autorización. SAML implementa un sistema de firma única, en cuya vinculación con el servicio consumidor puede actuar como una *autoridad emisora* de confiabilidad. Esto permite

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
 <Header>
    <wsse:Security xmlns:wsse= "http://schemas.xmlsoap.org/ws/2002/12/secext">
     <saml:Assertion xmlns:saml="..."...>
     <saml:Conditions ...>
     <saml:AuthorizationDecisionStatement
        Decision="Permit"
        Resource="http://www.xmltc.com/tls/...">
        <saml:Actions>
          ...
         <saml:Action>Execute</saml:Action>
        </saml:Actions>
        ...
     </saml:AuthorizationDecisionStatement>
    </wsse:Security>
 </Header>
 <Body>
    ...
 </Body>
</Envelope>
```

Código 6.16: Ejemplo del uso de elementos SAML dentro de un constructor `Security`.

a la lógica subyacente, no solamente autenticar y autorizar el servicio consumidor, sino también asegurarles a otros servicios que el servicio consumidor en cuestión, ha alcanzado el nivel de certeza y veracidad.

Por ende, los demás servicios (proveedores) no necesitan realizar los pasos de autenticación y autorización con el servicio que se verificó en el sistema de firma única basado en SAML. Solamente basta que estos servicios se contacten con el servicio de autoridad de única firma para solicitar la información de seguridad de autorización y autenticación que obtuvo el servicio consumidor. El servicio de autoridad de firma única provee esta información en la forma de aserciones de políticas que establecen los detalles de seguridad. Existen dos tipos de aserciones de políticas de seguridad: las aserciones de autenticación y las de autorización.

Como se explicó, un elemento `Security` es un contenedor estandarizado de bloques de encabezados referidos a seguridad, en el cual puede tener bloques expresados con elementos de otras especificaciones de seguridad. En el Cód. 6.16 se muestra un bloque SAML ubicado dentro de un constructor `Security`.

Confidencialidad e Integridad

La ***confidencialidad*** es relacionada con la protección del contenido de un mensaje. Un mensaje se considera que mantiene su confidencialidad si ningún servicio o agente no autorizado ve su contenido en algún punto de la ruta del mensaje.

La ***integridad***, por otra parte, asegura que un mensaje no haya sido alterado desde su despacho desde el emisor original. Esto garantiza que el estado de un mensaje permanece intacto en el viaje entre los extremos de una interacción.

El tipo de tecnología utilizado para proteger un mensaje, determina el tramo por el cual un mensaje permanece protegido mientras atraviesa su ruta de mensaje. El mecanismo propuesto por ***SSL*** *(Secure Sockets Layer)* [528, 172, 218] por ejemplo, es una tecnología muy popular para asegurar canales HTTP; sin embargo, en escenarios de comunicación de Servicios Web puede proteger los mensajes solamente entre endpoints de servicios. Por ende, es que se afirma que SSL solamente afronta la seguridad a nivel de transporte.

Si, por ejemplo, un servicio web intermediario toma posesión del mensaje, se está en potencia de alterar su contenido, aún preservando la seguridad a *nivel de transporte*. Para asegurar que los mensajes estén completamente protegidos a lo largo de toda la ruta de mensaje, se requere

```
<InvoiceType>
  <Number>2322</Number>
  <Total>$32,322.73</Total>
  <Date>07.16.05</Date>
</InvoiceType>
```

Código 6.17: Ejemplo del documento XML a encriptar.

```
<InvoiceType>
  <Number>2322</Number>
  <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
     Type="http://www.w3.org/2001/04/xmlenc#Element">
     <CipherData>
       <CipherValue>R5J7UUI78</CipherValue>
     </CipherData>
  </EncryptedData>
  <Date>07.16.05</Date>
</InvoiceType>
```

Código 6.18: Ejemplo del documento XML con una parte encriptada.

algún método que provea seguridad a *nivel de mensajes*. En estos casos, es necesario que las medidas de seguridad se apliquen al mensaje en si, no solamente a su medio de transporte; de esta forma las medidas de seguridad van con el mismo mensaje, independientemente de por dónde el mensaje pueda viajar. La confidencialidad a nivel del mensaje para un formato de mensaje basado en XML, como ser SOAP, puede ser realizada a través del uso de especificaciones que sean compatibles con el marco propuesto por WS-Security. Existen dos importantes extensiones de WS-Security para proveer la confidencialidad e integridad de un mensajes: *XML-Encryption* y *XML-Signature*.

XML-Encryption [412] es una tecnología de encriptación diseñada para usarse con XML, siendo una piedra angular para el marco de trabajo de WS-Security. Provee características que posibilitan que un mensaje sea encriptado en su totalidad, o solamente en determinadas partes.

El elemento **`EncryptedData`** es un elemento definido en XML-Encryption y es el constructor principal que contiene toda porción encriptada de un documento XML. Si está colocado en la raíz de un documento XML, los contenidos del documento completo serán encriptados. El elemento `EncryptedData` tiene un atributo **`Type`** que indicá qué se incluye en el contenido encriptado. Si el valor del atributo es `http://www.w3.org/2001/04/xmlenc#Element` que indica que el elemento y su contenido es encriptado, o si es `http://www.w3.org/2001/04/xmlenc#Content` establece que la encriptación sólo se aplica al contenido encerrado entre etiquetas específicas.

Dentro de un elemento `EncryptedData` se exige la presencia del subelemento **`CipherData`**. Este elemento el cual puede contener un subelemento **`CipherValue`**, el cual contiene los caracteres del texto encriptado; o bien, el subelemento **`CipherReference`** que provee una referencia a un valor encriptado.

En el Cód. 6.17 se muestra como ejemplo una porción de un documento XML, relacionado a una factura comercial. En el Cód. 6.18 se muestra una porción codificada de dicho mensajes, en este caso se codificó el elemento `Total`.

Para asegurar la integridad del mensaje, se exige un mecanismo que sea capaz de verificar que un mensaje recibido por un servicio es auténtico y no ha sido alterado de ninguna manera desde su emisión. Para esto, se propuso ***XML-Signature*** [141]; siendo una tecnología que permite que un documento XML sea acompañado de una pieza de información que representa una ***firma digital*** *(digital signature)*. Esta firma es ligada al contenido del documento, de forma tal que la verificación de la firma por el servicio destinatario, sólo tiene éxito si el contenido permaneció inalterado desde su generación con dicha firma.

Elemento	*Descripción*
CanonicalizationMethod	identifica el tipo de algoritmo de "canonicalización" (canonicalization algorithm) usado para detectar o representar sutiles diferencias en el contenido de los documentos, como por ejemplo la ubicación de los espacios en blanco.
DigestMethod	identifica el algoritmo a utilizar en la firma digital.
DigestValue	contiene el valor que representa el documento a ser firmado. Este valor es generador por la aplicación del algoritmo definido en `DigestMethod` al documento XML.
KeyInfo	es un constructor opcional que contiene la clave pública del emisor del mensaje.
Signature	es el elemento raíz de la firma digital, contenedor de todos los demás subelementos que la describen.
SignatureMethod	indica el algoritmo usado para producir la firma digital. Los algoritmos definidos en `CanonicalizationMethod` y `DigestMethod` son tenidos en cuenta a la hora de crear la firma digital.
SignatureValue	es el verdadero valor del la firma digital.
SignedInfo	es un constructor que contiene elementos con información relevante para el elemento `SignatureValue`, pero que reside fuera de ese constructor.
Reference	cada documento que es firmado por la misma firma digital es representado por un elemento `Reference` que contiene detalles relacionados con el compendio y la transformación del documento firmado.

Cuadro 6.1: Elementos utilizados en la definición de una firma digital basada en XML-Signature

Hay que notar, que las firmas digitales también soportan el principio de *no-repulsión*; el cual indica que se puede probar que un mensaje conteniendo un documento fue enviado por un servicio consumidor específico y entregado a un servicios proveedor específico. Ambas tecnologías, encriptación y firma digital, se fundamentan en el uso de *claves*. Las ***claves*** son valores especiales para desbloquear un algoritmo sobre el cual están basadas la encriptación y la firma digital. Las ***claves compartidas*** son típicamente usadas para las tecnologías de encripación y requieren que ambos servicios, consumidor y proveedor, conozcan la misma clave. Por su parte, el ***par de claves pública/privada*** es usado como medio para asegurar firma digital. El mecanismo consiste en codificar y firmar el documento en su generación con una clave (la privada); y usar la otra clave (la pública) para la recepción y decodificación del mismo.

Una firma digital es una compleja pieza de información compuesta por partes específicas, cada una de las cuales representa un aspecto del documento a ser firmado. En el Cuad. 6.1 se muestran varios elementos, definidos en XML-Signature, que pueden ser utilizados cuando se define un elemento que define una firma digital. En el Cód. 6.19 se muestra un bloque de encabezado SOAP que define la firma digital de un documento.

WS-Security y SOA

La seguridad a nivel de mensaje ha llegado a ser un componente básico de las soluciones orientadas a servicios. La medidas de seguridad pueden ser ubicadas sobre cualquier capa de transporte para proteger el contenido de los mensajes o para proteger al receptor. El marco de trabajo definido por WS-Security y sus especificaciones conexas colman los requerimientos fundamentales para asegurar la calidad de servicios; esto le permite a las organizaciones utilizar soluciones basadas en SOA para procesar información sensible y datos privados, y restringir el

```
<Envelope ...>
  <Header>
    <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
      <Signature Id="RailCo333" xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
          <CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
          <SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
          <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
            <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <DigestValue>LLSFKO32093548=</DigestValue>
          </Reference>
        </SignedInfo>
        <SignatureValue>9879DFSS3=</SignatureValue>
        <KeyInfo>...</KeyInfo>
      </Signature>
    </wsse:Security>
  </Header>
  <Body>
    ...invoice document with total exceeding $30,000...
  </Body>
</Envelope>
```

Código 6.19: Ejemplo la definición de firma digital basada en XML-Signature.

acceso a los servicios web según se requiera.

WS-Security establece un bloque de encabezado SOAP estandarizado que es usado para definir información sobre los tokens de seguridad o contener otros bloques de encabezados de seguridad. De esta forma, los principios y medidas de seguridad son adjuntadas a los mensajes SOAP, reforzando su autonomía. WS-Security es un marco de trabajo que puede usarse en conjunto con casi cualquier especificacion WS-*.

6.7. Resumen

Las tecnologías de Segunda Generación de Servicios Web ha brindado el impulso tecnológico necesario para que las arquitecturas SOA sean una opción viable y confiable para el desarrollo de sistemas distribuidos, altamente escalables, componibles, autónomos, integrables y reutilizables.

El principal logro de las tecnologías de Primera Generación, o tecnologías fundamentales de Servicios Web, es llegar a alcanzar grados importantes en la interoperatividad entre componentes o aplicaciones, posibilitando de esta forma la integración de las mismas; incluso en entornos en donde intervienen múltiples empresas, cada una con su propia infraestructura tecnológica y políticas de desarrollo.

Los escenarios de integraciones empresariales son requerimientos cada vez más comunes en los desafíos informáticos actuales; por ende, fue necesario extender las posibilidades y funcionalidades provistas por las especificaciones fundamentales de Servicios Web. Si bien la barrera de la heterogeneidad entre aplicaciones a integrar se ha solucionado usando un formato común de intercambio (XML), una marco de descripción estandarizado (WSDL) y un mecanismo de interacción basado en mensajes logrando bajo acoplamiento (SOAP); estaban pendiente de resolución cuestiones esenciales que hacen posible la realización de aplicaciones integradoras que surgen de la composición entre varios servicios web.

Un primer paso en la aplicación tecnológica de Servicios Web, fue posibilitar la interoperación entre sistemas informáticos de distintas organizaciones o dominios. De hecho, al poder interactuar las aplicaciones usando a la Web como canal, utilizando un intercambio de datos y una descripción de servicios estandarizados, los Servicios Web se consolidaron como tecnología que

derribó las barreras impuestas por la diversidad de plataformas, la falta de confianza entre dominios empresariales y las restricciones de seguridad impuestas al trafico de redes (cortafuegos). Fue así, que los Servicios Web y su núcleo de tecnologías fundamentales, hicieron mas común y más realizable los ambientes de integración de aplicaciones empresariales (EAI).

Para interrelacionar sistemas existentes de empresas diferentes y lograr interoperabilidad entre ellos, esta primera etapa de adopción de Servicios Web se centró en "enmascarar" las tales aplicaciones legadas exponiéndolas como servicios web. Es decir, realizar interacciones B2B con el propósito de fusionar flujos de trabajos. Sin embargo, cada vinculación era realizada en forma ad hoc, prefijando el formato de interacciones, los puntos de intercambio de datos y la política de seguridad con mecanismos específicos y modelos propietarios. A medida que la necesidad de estas interacciones B2B exigieron más flexibilidad, y los sistemas informáticos empresariales adoptaron las arquitecturas SOA para su diseño y desarrollo, fue necesario estipular y fijar especificaciones y tecnologías centradas en la composición de servicios web como mecanismo de integración, y en la autonomía de los servicios web para lograr esos niveles deseados de flexibilidad y bajo acoplamiento.

Fue así que surgen las especificaciones de Segunda Generación de Servicios Web, para hacer frente a estos desafíos. En este capítulo se describieron las tecnologías WS-* centrales y se detalló en cada caso la importancia de su impacto en las arquitecturas SOA y tecnologías de Servicios Web.

Como primera medida se describió a la especificación WS-Coordination, por ser la que describe el marco común para "conversaciones" entre múltiples servicios web. Propone una estrategia estandarizada para preservar el estado de una conversación o actividad, mediante el uso de un contexto de coordinación. De esta forma, ya no se requieren mecanismos propietarios para preservar la información de estado, que podrían ser incompatibles al interactuar servicios de diferentes empresas. También establece un método estándar para interactuar, el ciclo (Activación-Propagación-Registración) el cual posibilita a los servicios web participantes iniciar o incorporarse en las conversaciones de su interés, respetando un rol. Para gestionar los contexto de coordinaciones y el registro de sus participantes, se exige la existencia del Servicio de Coordinación. WS-Coordination, posee extensiones para posibilitar la interacción entre varios Servicios de Coordinación, logrando que la administración sea distribuida. Este hecho, permite lograr alta escalabilidad en el uso del marco de trabajo propuesto por WS-Coordination.

También se estudiaron las especificaciones WS-* relacionadas con transacciones. Por ser un requerimiento recurrente en cualquier sistema informático distribuido y por definir protocolos específicos que se emplean sobre WS-Coordination. En particular WS-AtomicTransaction establece los mensajes y estados por lo que debe pasar una conversación entre servicios web, si la interación que estos producen deben preservar las propiedades ACID. Por otro lado, WS-BussinessActivity está orientado a lograr transacciones con larga latencia y tiempo de ejecución, en las cuales es posible visualizar los efecto parciales del proceso transaccional.

Se introdujeron en este capítulo dos conceptos fundamentales que caracterizan y fundamentan esta nueva etapa de evolución en los Servicios Web materializada a través de las especificaciones WS-*. Estos conceptos son el de orquestación y coreografía de servicios web. Ambos conceptos plantean como principal objetivo la composición de servicios web, describiendo en forma autónoma la lógica de composición. Por un lado la orquestación, propone la definición de flujos de trabajo cuyo control de ejecución y preservación de información de estado son centralizados. Por otro lado, la coreografía propone la cooperación descentralizada de servicios web, especialmente aquellos que pertenecen a diferentes dominios empresariales.

A modo de implementación de un esquema de orquestación de servicios web, se presentó a la especificación WS-BPEL, que estipula la descripción de los flujos de trabajo en la forma de procesos de negocios, estableciendo las actividades que conforman el proceso, los servicios participantes que deben ejecutarlas, y las variables y medios para mantener la información de estado de ejecución del proceso. Por otro lado, y como ejemplo de marco de coreografía de

servicios web, se presentó la especificación WS-CDL, enunciando sus fundamentos principales y su ambiente de aplicación centrado en integraciones B2B.

El capítulo se cierra con una descripción de especificaciones WS-* que son importantes por aportar a cada uno de los marcos de trabajos explicados anteriormente importantes mejoras en confiabilidad, calidad de servicio y seguridad. Esta especificaciones ayudan a los servicios web, y a los mensajes que estos generan, que sean unidades de software altamente autónomas.

Todas estas especificaciones de Segunda Generación posibilitan que los servicios web sean unidades de software altamente reutilizables, con facilidad de integración. Las especificaciones WS-* son la culminación del proceso de instauración de los principios de SOA como método y medio para afrontar y solucionar las problemáticas descriptas en los paradigmas de B2B y EAI.

Capítulo 7

Servicios Web y Reusabilidad

Una de las principales problemáticas establecidas en la industria informática es la aplicación del principio de Reusabilidad de software. Lograr reutilizar software fue una meta establecida en los orígenes de la misma computación y, hasta el día de hoy, su estudio y adopción sigue siendo fuente de innumerables investigaciones y desarrollos técnológicos.

Para inicia este capítulo, se describirá qué se entiende por reusabilidad, reutilización y reuso de software y se plantearán las características y problemáticas referentes a los fundamentos y prácticas de la reutilización. A continuación, como el presente trabajo se basa en las tecnologías de Servicios Web, se presenta al paradigma SOA como marco teórico para la realización de servicios, siendo la Reusabilidad uno de sus principios fundamentales. Se estudia el principio de Reusabilidad en arquitecturas SOA y su influencia en el diseño de servicios y en otras cualidades de orientación a servicios. Se valoriza la adecuación de los Servicios Web como medio tecnológico para la realización de un ambiente SOA empresarial, estableciendo a la reusabilidad como práctica sistemática al adoptar dicho paradigma.

Posteriormente se tratarán las diferentes facetas que vinculan a la reusabildiad con los servicios web. En primer lugar se estudia a la utilización de Servicios Web como medio para modernizar aplicaciones legadas, haciendo especial énfasis en la técnica de envolturas de servicios web. Se describirán algunos criterios metodológicos para la aplicación de envolturas con el propósito de reutilizar la lógica encapsulada en componentes legados.

A continuación se hace un análisis del reuso de información ontológica mediante tecnologías de la Web Semántica. Se plantea el marco teórico de las ontologías y su materialización con la Web Semántica. Se presentan diferentes lenguajes ontológicos haciendo énfasis en su nivel de expresividad. También se enuncian a diferentes metodologías para la construcción de ontologías que muestren a la reutilización y reuso como actividades normadas.

Para concluir el capítulo, se presenta el concepto de Servicio Web Semántico, siendo un concepto que fusiona a los Servicios Web con la Web Semántica para automatizar las tareas de utilización de servicios web mediante descripciones semánticas de los mismos; estas actividades conllevan a la automatización del reuso. Se presentan marcos de trabajo para la definición de Servicios Web Semánticos, realizándose una valoración de los mismos.

7.1. Reusabilidad de software

7.1.1. Concepto y problemática

La reusabilidad de software ha sido, y sigue siendo, uno de los importantes temas de investigación en la Ingeniería del Software, citándose a menudo como una de las principales técnicas para incrementar la productividad de los desarrolladores de software. Algunos expertos de la temática [307] afirman que la investigación en las décadas pasadas presentan a la reutilización como única aproximación realista para llegar a los índices de productividad y calidad que la in-

dustria del software necesita. Sin embargo, se reconoce que no se han conseguido grandes avances en la adopción sistemática de la reutilización, o *reuso sistémico*, en el proceso de construcción del software.

La idea de *reuso formal* fue introducida por primera vez en un artículo de 1968 presentado por M. Douglas McIlroy [290] La concepción de reusabilidad presentada en dicho trabajo implica el desarrollo de una industria de componentes de códigos fuentes reutilizables y la industrialización de la producción del software de aplicación a partir de componentes generales *(off-the-shelf components)*. La reutilización se concebía como la combinación de componentes de código almacenados en una biblioteca y llevada a cabo de una forma carente de cualquier método.

Sin embargo en la actualidad, los fundamentos y alcances del reuso han cambiado; hubo una evolución en el concepto hacia la idea de que todo el conocimiento y productos derivados de la producción de software son susceptibles de ser reutilizados en la construcción de nuevos sistemas [180], surgiendo de esta forma el concepto de *bien*, *artículo* o *componente* de software reutilizable [248]. Un ***bien*** *(asset)* obtenido en en el ciclo de vida de un software, puede ser reutilizable para el ciclo de vida de un software distinto, con independencia de su nivel de abstracción [132, 398].

Según lo expresado anteriormente, la reutilización se puede definir como cualquier actividad que produce o ayuda a producir un sistema nuevo mediante el (re)uso de algún elemento o artefacto procedente de un esfuerzo de desarrollo anterior [181]. Una definición más específica consiste en definir a la reutilización como la utilización de conceptos o artefactos de software existentes durante la construcción de un nuevo sistema software, en forma directa o adaptada. Para ello, estos conceptos y/o artefactos deberán encontrarse codificados en un nivel de abstracción establecido y deberán permitir ser recuperados [261].

La idea del ***reuso*** es bastante simple en teoría: simplemente es hacer que un programa de software sea usado con más de un propósito. Las razones para hacer esto son bastante obvias. Mientras que algo sea utilizable para un único propósito tiene un valor inherente, si algo es utilizado repetidamente proveerá un incremento repetido de valor y por consiguiente será una inversión más atractiva. La fundamentación es bastante lógica, pero hay que entender que si bien el reuso es un concepto simple, lograr efectivamente reuso en la Industria del Software no ha sido fácil a lo largo de su historia [151].

La *reutilización de software* ha sido un tema de debate y discusión por más de 30 años en la industria informática. Muchos desarrolladores han logrado algunos éxitos aplicando ***reuso ad-hoc***, o *"reuso oportunista"*: el cual consiste en utilizar, según el criterio y voluntad del programador, fragmentos de código ya desarrollado en programas existentes para la confección de nuevos programas. Si bien el reuso ad-hoc funciona adecuadamente en pequeños equipos de desarrollo y en un bajo número de proyectos, no sirve para lograr que la reusabilidad sea un principio de diseño aplicado efectivamente y una actividad escalable a través de diferentes unidades de negocios o empresas para lograr sistemáticamente reuso. Solamente el ***reuso sistemático*** es el medio promisorio para lograr reducciones de costos y tiempo en el ciclo de vida del software, mejorar la calidad de software y justificar y promover los esfuerzos existentes para construir y usar artefactos reutilizables multiuso como ser arquitecturas, patrones, componentes y marcos de trabajo [422].

Si se piensa en reuso, no puede pensarse en software con un único propósito. El hecho de construir un software para un único propósito posibilita enfocarse en un único conjunto de requerimientos muy específicos de un proyecto; todo el sistema puede ser optimizado y personalizado para esos requerimientos puntuales. Además el software es adaptado para aquellos escenario de usos que fueron relevados y analizados para este proyecto puntual. Este alcance acotado afecta a todo el ciclo de vida del software: resultan más fácil el diseño, el desarrollo y el testeo debido a que los ambientes de uso son limitados y predecibles. De la misma forma, la puesta en marcha y la subsecuente administración del software es bastante simple por necesitar solamente asegurarse que cumpla su único objetivo.

Sin embargo, se puede optar por el diseño de programas que pueden ser utilizados con más

de un propósito. Para lograr esto, se necesitan tener varias consideraciones importantes. Por ejemplo, cuando se diseña un programa multipropósito es necesario definir su forma de uso en múltiples escenarios. Esto tiende a cambiar y extender su lógica de programación o *lógica de solución*, que requiere ser más genérica y, eventualmente, debe proveer más funcionalidades. El diseño se vuelve más complejo, lo que exige un incremento en los esfuerzo de desarrollo para poder acomodar todos los escenarios asociados con un rango planificado de capacidades. El testeo es otra fase que se ve profundamente afectada. Programas más grandes pueden requerir lógica adicional que pueda manejar un grupo de excepciones adicionales; de igual forma se necesitan mayor cantidad de casos de prueba, según existan más escenarios de aplicación. En lo referente a la implementación y puesta en marcha, las aplicaciones reutilizables pueden necesitar un ambiente de ejecución que sea capaz de satisfacer los crecientes requerimientos de disponibilidad y escalabilidad. Una alternativa, sería su implementación redundante, lo que exige demandas adicionales de infraestructura. Por último, una vez que el programa ha sido implementado y está en uso, es esencial decidir en qué forma debe evolucionar. Indistintamente si el programa será usado por personas u otras aplicaciones, una vez liberado, se pierde la libertad de realizar cambios arbitrarios. Si una aplicación o cliente estableció dependencias con el software, y programáticamente se vincula a través de sus interfaces, esas dependencias, expuestas en la primera versión, fueron especificadas en la etapa de diseño del software reusable y deben ser respetadas en las subsiguientes versiones.

Cualquier programa que fuese construido para la venta al público en general fue diseñado con mentalidad en el reuso. Ese fue uno de los motivadores del artículo original de McIlroy [290] para hacer frente a la *Crisis del Software*. Lograr concebir una industria de componentes de códigos fuentes o artefactos reutilizables y la industrialización de la producción del software en sí a partir de estos componentes. Es decir, lograr software o componentes de software de uso público y potencial en diferentes ambientes desconocidos a la hora de su diseño *(off-the-shelf software)*.

No importa si es un sistema operativo, un procesador de texto, o una plataforma de middleware completa, en los estados iniciales de diseño de estos programas, se tomaron en cuenta las consideraciones antes mencionadas. Esto hace que la noción de reuso sea tan antigua como la propia industria del software. También brinda un claro entendimiento de lo que se requiere para tratar con la reusabilidad en el ciclo de vida del producto software. La reusabilidad incrementa la complejidad, costo, esfuerzo y tiempo consumido para construir software. Más aún, puede ser complicado y exigente construir soluciones a partir de la incorporación y ensamble de programas ya desarrollados por otros equipos. La reusabilidad no siempre ha sido un característica de diseño; sin embargo, las organizaciones han tenido que elegirla para continuar con el desarrollo de sus soluciones internas [151].

Una de las críticas más importantes que tiene la práctica del reuso es cuestionar la necesidad de analizar y desarrollar software para futuros escenarios potenciales de uso, que pueden existir o no; siendo que ya se cuenta con un real conjunto específico de requerimientos que necesitan ser cubiertos ahora. En otras palabras, por qué atender a funcionalidades no urgentes cuyas necesidades pueden nunca darse; si existen requerimientos definidos y urgentes.

Es real que la construcción de programas para un único propósito tiene una cierta y definida compensación económica, dado por su objetivo bien especificado. Es una medida de inversión y retorno de inversión; sin embargo, este tipo de razonamiento es el que generalmente conduce a los ambientes de aplicaciones aisladas, que no tienen como objetivo la interoperabilidad con otras aplicaciones, y mucho menos el reuso.

El advenimiento de la orientación a objetos tiene el mérito de generar conciencia acerca de los beneficios que se obtiene cuando se construyen aplicaciones a partir de *componentes* (objetos) capaces de servir a más de un propósito inmediato. El reuso a partir de aplicaciones diseñadas orientadas a objetos fue intentado y alcanzó diferentes niveles de éxito [510]. Como ejemplo de esto pueden citarse las propuestas de patrones de diseño orientado a objetos [173], promoviendo la

concreción de sistemas de patrones [88, 183] los cuales pueden conformar repositorios distribuidos [423].

Muchos de los equipos involucrados en iniciativas poco exitosas de reuso llegaron a desilusionarse con la visión de establecer un inventario de objetos compatibles que cosecharía grandes retornos en los años por venir. Varios problemas comunes surgieron en estos proyectos, algunos son:

- el reuso potencial de un componente estaba limitado a un ambiente de ejecución propietario (por ejemplo: a una máquina virtual o *runtime* determinado),
- la funcionalidad de un componente reutilizable era consumida por aplicaciones clientes propietarias,
- los componentes reutilizables sufren una sobrecarga de recursos resultando en dependencias fuertes con otros componentes (como por ejemplo las estructuras de herencia y sus diseños altamente acoplados),
- los componentes que son legítimamente reutilizables no fueron usado lo suficientes,
- los componentes reutilizables fueron equipados con demasiada funcionalidad que no fue requerida a fin de cuentas.

A pesar que las iniciativas previas de reuso no fueron del todo exitosas, las empresas de software y las organizaciones de estándares continuaron trabajando hacia una visión que permita a las organizaciones establecer recursos de software empresariales efectivos y reutilizables. La subsecuente aparición de las tecnología de Servicios Web ha sido un significante avance respecto al reuso, debido a que el marco de trabajo propuesto, neutral a las plataformas y vendedores de software, afrenta las limitaciones asociadas los ambientes de ejecución propietarios (primer y segundo items en la lista anterior). Los Servicios Web lograron exitosamente incrementar el reuso potencial. Una porción de lógica de aplicación expuesta como un servicio web puede ser accesible a cualquier parte de la empresa que soporte la tecnología de mensaje estipulada por el marco de trabajo de Servicios Web (Véase Sec. 5.6). Por consiguiente, en la medida que las restricciones impuestas por el marco de Servicios Web no sea un factor restrictivo, el rango de potenciales aplicaciones clientes consumidoras de servicios se incrementa naturalmente.

Sin embargo, ha resultado evidente que la adopción de la tecnología innovadora no es suficiente por si misma, para solucionar gran parte de los obstáculos que bloquearon los esfuerzos previos de reuso (Vér más adelante Sec. 7.1.2). El hecho es que ninguna innovación tecnológica, inclusive las tecnologías SOA, pueden solucionar problemas endémicos organizacionales de índole no-técnica. Estos deben ser resueltos en la organización en sí [397].

Muchos de los principios de diseño orientado a servicios fueron inspirados por los desafíos pasados. Estos establecen un paradigma en el cual la reusabilidad es una consideración medular y central. Para organizaciones interesadas en lograr reuso a gran escala, un compromiso con la orientación de servicios es un claro camino. Depende de las organizaciones determinar el alcance de este compromiso.

En este punto, es importante destacar las diferencias entre el término *reusabilidad*, *reutilización* y *reuso*. La primera, ***"reusabilidad"***, es una característica de diseño que se espera adoptar con este principio; ***reutilización*** es la actividad o procedimiento de utilizar bienes o artefactos preexistente para concretar o bien o artefacto nuevo, mientras que el último término: ***"reuso"***, es el resultado final que se intenta lograr aplicando el principio de reusabilidad, en un proceso de reutilización o para reutilización. Extender la reusabilidad en la construcción de un software, permite determinar las posibilidades de reutilización y valorar el potencial de reuso final.

7.1.2. Inhibidores del reuso

Sin embargo, al igual que otras tecnologías prometedoras en la historia del software, la práctica de la reutilización de software no ha logrado significativas mejoras universales en la calidad y productividad. Existen, desde ya éxitos al respecto, como por ejemplo la existencia de sofisticados frameworks de componentes reutilizables en lenguajes orientados a objetos que corren en varias plataformas. Pero en general estos marcos de trabajo se han centrado en un número pequeño de dominios de aplicación, como son las interfaces gráficas de usuario y las bibliotecas de códigos de C++ como la STL [216]. Es decir, el reuso de componentes estaba limitado, en la mayoría de los casos prácticos, a las bibliotecas y herramientas de terceras partes. Y lo más grave era el hecho que no se considera a la reusabilidad como una parte integral de los procesos de desarrollo de software.

En teoría, las organizaciones reconocen el valor del reuso sistemático y premian las iniciativas que lo sustentan. Pero en la práctica, muchos factores conspiran haciendo que la practica del reuso sistemático sea difícil de llevar a cabo. En particular, en aquellas organizaciones que cuentan con una base instalada de aplicaciones legadas y desarrolladores no propensos al cambio de paradigmas.

Como impedimentos no-técnicos del reuso sistemático se pueden citar [422]:

- *Impedimentos organizacionales*: el desarrollo, puesta en marcha y soporte del reuso sistemático requiere un profundo entendimiento de las necesidades de los desarrolladores de aplicaciones y de los requerimientos de la actividad. A medida que aumenta el número de desarrolladores y proyectos utilizando artefactos reutilizables, comienza a ser crítico una estructura organizativa para proveer la correspondiente retroalimentación entre estas entidades.

- *Impedimentos económicos*: financiar un programa de reuso de alcance organizacional requiere inversión, en particular si los grupos de reuso operan en forma centralizada. Muchas organizaciones no pueden realizar una adecuada valuación presupuestaria y cálculo de tasa de retorno de estas actividades; por consiguiente es difícil financiar al reuso sistémico.

- *Impedimentos Administrativos*: es un trabajo arduo catalogar, archivar y recuperar artefactos reutilizables en una organización que cuenta con varios departamentos. A pesar que es común reutilizar pequeñas clases y funciones oportunamente de programas existentes, es difícil para los desarrolladores buscar y localizar oportunidades de reuso más allá de su grupo de trabajo inmediato.

- *Impedimentos políticos*: Rivalidades internas entre los diferentes sectores de una organización pueden atentar contra la evolución del plan de reuso. Otros grupos de producción de software pueden ver como un amenaza a su seguridad laboral la influencia de los grupos relacionados con la tarea de reutilización. Por ejemplo, los grupos de trabajos orientados al desarrollo y mantenimiento de plataformas de middlewares son a menudo vistos por otros desarrolladores como invasores a su forma de trabajo, ya que se sienten excluidos de decisiones ingenieriles respecto a la arquitectura de sistemas.

- *Impedimentos psicológicos*: muchos desarrolladores de aplicaciones ven el esfuerzo gerencial hacia el reuso como un indicio de falta de confianza a sus habilidades profesionales. Además, el síndrome *"no hecho acá"* está latente en muchas organizaciones, en particular aquellas que poseen programadores talentosos.

Como si estos impedimentos no-técnicos contra la práctica de la reusabilidad fueran pocos, existen aquellos que hacen fracasar los esfuerzos de reusabilidad y que tienen su origen en la falta de habilidades técnicas en los desarrolladores y falta de competencias necesarias por parte de la organización para crear y/o integrar los artefactos reutilizables en forma sistemática.

También se observan que los desarrolladores depositan mucha fe en las características de los lenguajes de programación que ellos eligen usar. Por ejemplo herencia, polimorfismo, plantillas, manejo de excepciones son concebidas como fundamentales para la existencia del reuso. Sin embargo, los lenguajes de programación por si solos no capturan adecuadamente lo común y particular de cada abstracción y componente requerido para construir sistemáticamente software reutilizable en dominios complejos.

Estos son algunos ejemplos de factores que tradicionalmente fueron inhibidores del éxito de reuso en las tecnologías de la información. La orientación a servicios trata de frente con estas cuestiones promoviendo los principios que preparan a un servicio para que sea reutilizable desde su gestación.

7.2. Reusabilidad y SOA

En el mundo de la informática de hoy, las arquitecturas abiertas y distribuidas tiene su mejor ejemplo en las ***Arquitecturas Orientadas a Servicios SOA*** *(Service Oriented Architectures)*. Sin lugar a dudas es posible establecer niveles de reusabilidad y lograr oportunidades de reuso en utilizar Servicios Web. Sin embargo, el hecho de establecer a los servicios como artefactos o bienes reutilizables de una empresa, conformando una arquitectura SOA, permite lograr mayor flexibilidad en la utilización de tales servicios, no sólo en los proyectos existentes y por concretar, sino también en futuros escenarios no del todo previstos. Establecer a los servicios como bloques de construcción tiene el efecto de lograr aplicaciones en forma más eficiente y eficaz, lo cual redunda en promover una empresa más ágil y adaptable al cambio [151].

7.2.1. El paradigma SOA

Las arquitecturas SOA son en sí un paradigma de diseño; un paradigma de diseño es un enfoque para diseñar una *lógica de solución*. Una ***lógica de solución*** *(solution logic)* es un diseño orientado a solucionar cierto requerimiento o problema computacionalmente resoluble. Cuando se construye una lógica de solución distribuida, las diferentes propuestas giran al rededor de la teoría de ingeniería en software conocida como *separación de conceptos* o *separación de asuntos (separation of concerns)* [136, 378, 125]. En una forma resumida, la ***separación de conceptos*** establece que un problema grande es resuelto más efectivamente cuando se lo descompone en un conjunto de problemas o asuntos más pequeños. Esto brinda la posibilidad de partir la solución en pequeñas porciones de lógica, denominadas ***capacidades*** *(capabilities)*, cada una de las cuales está diseñada para solucionar un problema o asunto individual. Las capacidades relacionadas pueden ser agrupadas en unidades de lógica de solución. El beneficio fundamental de resolver problemas en esta forma es que una cantidad de unidades de lógica de solución pueden ser diseñadas para resolver los problemas inmediatos (particionados) y permanecer genéricas al problema más grande. Esto permite la constante oportunidad de reutilizar dichas capacidades en aquellas unidades destinadas a resolver otros problemas.

Existen diferentes paradigmas de diseño para lógicas de soluciones distribuidas. Lo que distingue a la orientación a servicios de otros paradigmas de diseño es la forma en la cual se lleva a cabo la separación de conceptos y cómo formar las unidades individuales de la lógica de solución. Desde la perspectiva del paradigma de diseño SOA, estas unidades individuales se denominan *servicios*. La organización OASIS define a ***servicio*** como "un mecanismo para acceder a una o más capacidades (funcionalidades), en dónde el acceso a tales capacidades se provee mediante el uso de interfaces predefinidas (y públicas), el cual es ejecutado consistentemente con las restricciones y políticas conforme a lo especificado en la descripción del servicio" [332]. Tal como se explicó en la Sec. 5.3.1, y ampliando tales definiciones, las arquitecturas SOA separan las funciones en servicios, los cuales pueden estar distribuidos en una red y pueden ser combinados y reutilizados para crear procesos de negocios con diferentes propósitos [258].

Lo importante y destacado de SOA es que la reusabilidad es uno de sus principios fundamentales, el cual establece que *que todo servicio es reutilizable*. Es decir, los servicios contienen y expresan una *lógica agnóstica* [1] y pueden ser considerados como un recursos empresarial reutilizable [151].

7.2.2. Principios de Diseño SOA

La *orientación de servicios* es un tema multidimensional. Es a través de la aplicación de sus principios de diseño que sus beneficios son realizados y se pueden construir lógicas de solución que pueden ser clasificadas como realmente "orientadas a servicios". Esto resulta en la automatización de un ambiente con dinámica y características únicas, cada una de las cuales deben ser bien comprendidas y planificadas. Como todo paradigma de diseño de software, se deben aplicar estos principios para lograr obtener una estructura de software orientada a servicios. Dicho de otra forma, los servicios deben cumplir con ciertas caracteristicas de diseño que estipulan determinados principios. Se detallan a continuación los principios de diseño SOA, segun el trabajo más completo y acabado del tema propuesto por Thomas Erl [151].

Contrato Estandarizado de Servicio

Los servicios expresan su propósito y capacidades mediante un *contrato de servicio*. En varios de los conceptos y definiciones, se han utilizado los términos *proveedor de servicio* (indicando el componente software que implementa el servicio accesible a través de un endpoint) y *consumidor de servicio* (indicando la aplicación cliente que invoca capacidades del servicio). Esta analogía con la economía se completa con la definición de ***contrato*** *(contract)*, denominándose así a los protocolos que deben cumplir cliente y proveedor de servicio para poder interactuar. El ***diseño por contrato*** *(Design by Contract)* tiene sus orígenes en las primeras aplicaciones del principio de separación de conceptos [136, 220, 204] y en el diseño y programación orientada a objetos [294, 295, 312].

El principio ***Contrato Estandarizado de Servicio*** *(Standardized Service Contract)* es quizás la parte más fundamental de la orientación a servicios; esencialmente requiere que se tomen en cuenta consideraciones específicas al diseñar la interfaz pública y se realice una evaluación de la cantidad de contenido que será publicado como parte del ***contrato oficial del servicio***. Un gran énfasis se coloca en los aspectos específicos del diseño del contrato, incluyendo la manera en la cual lo servicios expresan su funcionalidad, cómo los tipos y modelos de datos son definidos, y cómo se declaran y vinculan las políticas asociadas. Existen restricciones que se centran en asegurar que los contratos del servicios sean optimizados, adecuadamente granulados y estandarizados para asegurar que los endpoints establecidos por el servicio sean consistentes, fiables y gobernables.

Bajo Acoplamiento de Servicio

El *acoplamiento* se refiere a la conexión o relación entre dos cosas; una medida de acoplamiento es el nivel de dependencia entre estas cosas [370]. El ***bajo acoplamiento*** es un enfoque para el diseño de aplicaciones distribuidas con énfasis en la agilidad para adaptarse a los cambios. El bajo acoplamiento intencionalmente sacrifica optimización de interfaz para lograr flexibilidad en interoperabilidad entre sistemas que son de diferente tecnología, ubicación, disponibilidad y performance. Un aplicación bajamente acoplada es aislada de los cambios internos de otras aplicaciones mediante el uso de abstracción, indirección y ligadura tardía en las interfaces entre aplicaciones. En comparación con las aplicaciones "tradicionales" estrechamente acopladas, las

[1] El término "agnóstico/a" significa "sin creencia" o "sin conocimiento". Por consiguiente, una lógica que es lo suficientemente genérica o abstracta, de forma tal que no es específica (o no tiene conocimiento de) una tarea o acción específica, es clasificada como *lógica agnóstica*. Debido a que el conocimiento específico de una tarea con un único propósito es intencionalmente omitido, las lógicas agnósticas son consideradas multipropósito. Por el contrario, las lógicas que encierran conocimiento específico son denominadas *lógicas no-agnósticas*.

aplicaciones con bajo acoplamiento están orientadas a ser más reusables y adaptables a escenarios y requerimientos inesperados [249].

El principio de ***Bajo Acoplamiento de Servicio*** *(Service Loose Coupling)* aboga por la creanción de un tipo específico de relación dentro y fuera de los límites del servicio, con un constante énfasis en la reducción de las dependencias entre el contrato de servicio, su implementación y las aplicaciones consumidoras del servicio. Además promueve el diseño y evolución independiente entre la lógica del servicio y su implementación mientras aún garantiza interoperabilidad con aplicaciones consumidoras que deben confiar en la capacidades del servicio. Existen numerosos tipos de acoplamientos involucrados en el diseño de servicio, cada uno de los cuales pueden tener influencia en el contenido y la granularidad de su contracto. Lograr establecer el nivel apropiado de acoplamiento requiere que exista un balance entre las consideraciones prácticas y varias preferencias de diseño de servicio.

Abstracción de servicio

La ***abstracción*** concuerda con muchos aspectos de la orientación a servicios. En un nivel fundamental, el principio de ***Abstracción de servicio*** *(Service Abstraction)* enfatiza la necesidad de ocultar los detalles subyacentes del servicio tanto como sea posible; esto influye directamente en permitir y preservar la calidad de bajo acoplamiento descripta anteriormente. El diseñar y desarrollar un servicio es diferente a desarrollar un objeto u otro componete de software. Un servicio es definido por los mensajes que intercambia con otros servicios, antes que con su signatura de método. Un servicio debe ser definido con un mayor nivel de abstracción que un objeto ya que es posible mapear la definición de un servicio con una rutina hecha en un lenguaje procedural como COBOL o PL/1, o con un sistema de mensajes como JSM o MSMQ, y también con sistemas orientados a objetos como los hechos con JavaEE o .Net [328].

La Abstracción de Servicios también juega un rol significativo en la postura y diseño de composiciones de servicios. Varias formas de metadatos entran en escena cuando se evalúan niveles apropiados de abstracción. El nivel de abstracción aplicado en un servicio puede afectar la granularidad de su contrato y también puede influir en el costo y administración de servicio.

Como ejemplo más importante de abstracción en SOA se presenta el concepto de *interfaz*. Una interfaz de servicio define, en forma precisa, un contrato y las obligaciones de las partes. Por ende, permite al consumidor del servicio usar su funcionalidad sin preocuparse de la implementación subyacente. Si tomamos la definición del W3C, un *servicio* es un recurso abstracto que representa una capacidad de realizar una tarea que conforma una funcionalidad coherente desde el punto de vista de las entidades proveedoras y consumidoras del servicio. Con el propósito de ser usado, un servicio debe ser realizado (implementado) por un agente proveedor concreto [96].

Reusabilidad de Servicio

El ***reuso*** es fuertemente promovido en la orientación a servicio; de tal forma que llega a ser una de las partes principales de procesos típicos de análisis y diseño de servicio, y también forma la base de modelos de servicios claves. Con el advenimiento de la tecnología madura y no-propietaria de servicios se ha logrado la oportunidad de maximizar el potencial de reuso de lógicas multipropósitos a un nivel nunca pensado.

El principio de ***Reusabilidad de Servicio*** *(Service Reusability)* enfatiza la instauración de los servicios como recursos y bienes empresariales con contextos funcionales agnósticos. Deben ser tomadas varias consideraciones de diseño para asegurarse que las capacidades individuales de los servicios son definidas apropiadamente en relación con un contexto de servicio agnóstico, y para garantizar que éstas pueden facilitar los requerimientos de reuso. En el resto de la sección se explicará más detenidamente el impacto y caracteristicas de la Reusabilidad de Servicios, su valoración y vinculación con los otros principios.

Autonomía de Servicio

Para que los servicios lleven a cabo sus capacidades en una forma consistente y fiable, su lógica de solución subyacente necesita tener un significativo grado de control sobre su ambiente y recursos. El principio de ***Autonomía de Servicio*** *(Service Autonomy)* favorece a aquellos otros principios de diseño que pueden ser logrados en ambientes reales de producción mediante la promoción de la fiabilidad y el comportamiento predecible del servicio.

La ***autonomía*** es la característica que permite a los servicios ser implementados, modificados y mantenidos en forma independiente de otros y de la lógica de solución que los usa. El ciclo de vida de un servicio autónomo es independiente de otros servicios. Al evaluar un servicio para determinar si cumple la características de autonomía, deben realizarse preguntas como si preserva bajo acoplamiento respecto a otros; es decir, si sus responsabilidades y capacidades están debidamente aisladas; si su ciclo de vida es independiente de otros servicios; y si su puesta en funcionamiento no afecta o condiciona a otros servicios [415].

Este principio conlleva varias cuestiones que se refieren al diseño de la lógica de servicio, como también al establecimiento del ambiente implementación. Las consideraciones sobre los niveles de aislamiento y normalización de servicios se toman en cuenta para alcanzar adecuados niveles de autonomía, especialmente en servicios reutilizables que son frecuentemente compartidos.

Servicio "Sin Estado"

Este principio hace referencia al *"estado de aplicación"* o *"estado de interacción"* en aplicaciones distribuidas. El ***"estado de interacción"*** es información que registra uno o más eventos en una determinada secuencia de interacciones con un usuario, con otra computadora o programa, con un dispositivo o con otro elemento externo. Una computadora o aplicacción puede mantener la información de los eventos constituyendo un estado de interacción; esta información se guarda en un medio de almacenamiento diseñado para tal propósito y se dice que la aplicación es ***"con estado"*** *(statefull)*. En el caso contrario, cuando no se registran ninguna de las interacciones previas y cada interacción requiere ser manejada enteramente con la información propia de ésta, se dice que la aplicación es ***"sin estado"*** *(stateless)* [535].

La gestión de excesiva información de estado puede comprometer la habilidad de un servicio y limitar su potencial de escalabilidad. Por ende, los servicios son diseñados para mantener información de estado sólo cuando se lo requiere. Aplicar el principio de ***Servicio "Sin Estado"*** *(Service Statelessness)* requieren que sean evaluadas las medidas de información de estado, basándose en la adecuación de la arquitectura tecnológica utilizada para la gestión y preservación de estados. Diseños capaces de diferir el procesamiento de datos de estado y gestión de estados permite que el servicio implementado maximice su disponibilidad; siendo esto una importante calidad de diseño, especialmente utilizada en ambientes de mucha concurrencia.

Descubrimiento de Servicio

La oportunidad para que los servicios sean utilizados en su completo potencial puede solamente ser realizada si su existencia, propósito y capacidades son conocidas, fácilmente localizables y comprensibles. Es por eso que uno de los aspecto más críticos para incentivar el consumo de servicios disponibles es la publicidad y publicación de los mismos mediante un mecanismo de descubrimiento como ser un Servicio de Registro. Un consumidor de servicios debe conocer la posibilidad de encontrar servicios que potencialmente pueden cubrir sus requerimientos. Para lograr esto, los servicios a utilizar deberían ser capaces de registrarse como productores, y el programa cliente debería poder negociar el contrato para consumirlos [278]. Es importante aclarar, que en Servicios Web, esta secuencia de tareas se realiza en forma automática si se cuenta con un marco de trabajo que implemente un Servicio de Registro, como se presentó en la Sec. 5.8.

Una de las claves desde una perspectiva de desarrollo es la capacidad de descubrir servicios reutilizables; por ende el ***descubrimiento*** es una capacidad técnica fundamental que debe lo-

grarse en cualquier iniciativa de SOA para posibilitar el consumo y reuso de servicios. Para que los servicios sean tenidos en cuenta como bienes informáticos con un respetable retorno de inversión, necesitan ser fácilmente identificados y comprendidos cuando las oportunidades de reuso se presentan. Por consiguiente, al diseñar un servicio se necesita tener en cuenta la "facilidad de comunicación" del servicio y sus capacidades individuales; además del mecanismo de descubrimiento (como por ejemplo un Servicio de Registro) que debe estar presente en el ambiente inmediato en el cual el servicio se implementa. La aplicación del principio de ***Descubrimiento de Servicio*** *(Service Discoverability)* tiene gran impacto en la capacidad de interoperatividad y posterior reuso del servicio.

Composición de Servicio

En una forma simplificada, la composición de servicios tiene como objetivo proveer medios eficientes y efectivos para crear, ejecutar, adaptar y mantener servicios que "confían" en (interactúan con) otros servicios de alguna forma. Con el propósito de lograr llevar a cabo estas promesas de la ***composición de servicios***, se requieren herramientas de desarrollo que incorporen abstracciones de alto nivel para facilitar, e incluso automatizar, la tarea asociada con la composición. Es decir, debe existir una infraestructura que permita el diseño y la ejecución de servicios compuestos [46].

A medida que se incrementan el número de soluciones informáticas orientadas a servicios, también lo hacen la complejidad de las configuraciones subyacentes de las composiciones de servicios. La habilidad de componer efectivamente los servicios es un requerimiento crítico para lograr algunos de los objetivos centrales de la computación basada en orientación a servicios. Como se explicó en Sec. 6.2, se espera que los servicios sean capaces de participar como miembros de una composición, sin importar si ellos son en sí mismos una composición.

Es posible crear una agregación de servicios forma tal que sólo se publique el que encapsula a múltiples otros servicios. Esto permite que varios servicios con interfaces de baja granularidad compongan a un servicio con una interfaz de más alta granularidad. De esta forma, se puede inferir que tiene sentido publicar lo servicios de alta granularidad, y dejar a los servicios de baja granularidad "privados" al servicio que los agrega [328].

Como se explicó en las Sec. 6.4 y 6.5, existen dos técnicas de diseño e implementación para realizar el principio de Composición de Servicios: la coreografía y orquestación de servicios. Una difiere de la otra al determinar dónde reside la lógica que controla la interacción o conversación entre los servicios participantes en la composición.

Una *coreografía* describe la colaboración entre servicios empresariales, generalmente de organizaciones diferentes, para el logro de un objetivo común. El control de la lógica está distribuido entre los servicios participantes, y la coreografía emerge a medida que los servicios interactúan entre sí. Para diseñar una coreografía, se deben describir en primer lugar las interacciones que los servicios deben cumplir para lograr el objetivo común, y las relaciones que existen entre estas interacciones. Una coreografía no describe las acciones que son llevadas a cabo internamente por los proveedores para realizar los servicios; simplemente establece reglas de colaboración.

Una *orquestación*, por otro lado, describe el comportamiento que implementa un proveedor para realizar un servicio. Por ende, a lógica de control está centralizada en el proveedor de servicio. Al diseñar una orquestación, se describen las interacciones que el proveedor del servicio debe tener con los otros servicios asociados, y las operaciones que el proveedor de servicio debe llevar a cabo internamente para realizar el servicio. Una orquestación requiere de un motor que la ejecute, ya que la orquestación en sí, es un proceso de negocios ejecutable.

7.2.3. Servicios reutilizables y servicios agnósticos

La entrega reiterada de servicios con alto grado de reusabilidad puede tener un tremendo impacto en la empresa. Debido a que tradicionalmente la entrega de una solución informática está

orientada a un único propósito o proyecto, los sistemas informáticos son típicamente asociados con la automatización de una tarea particular organizativa en un ambiente específico de aplicación. El perseguir la reusabilidad en SOA requiere posicionar la lógica de un servicio para que permanezca lo más neutral o *agnóstica* posible respecto al entorno de trabajo. Este principio establece el concepto de *servicio agnóstico* [151].

Entendiendo cada una de estas definiciones, se puede ver como un principio de diseño, la Reusabilidad, afecta a las cualidades del servicio. Existe diferencia entre el concepto de un servicio agnóstico y un servicio reutilizable. Si bien ambos comparten características de diseño, y por ende se relacionan, cada idea se distingue. Un ***servicio agnóstico*** **(agnostic service)** es independiente del proceso (o procesos) de negocio en el que se utiliza, de cualquier tecnología propietaria o plataforma de aplicación. Cuanto más agnóstico es un servicio, más genéricas son sus capacidades; su lógica es genérica y multipropósito. Por ende, cuanto más agnóstico sea un servicio, más grande es su potencial de reuso. Por ejemplo, un servicio tiene potencial de reuso si: provee capacidades que no son específicas a un proceso de negocios; y es usado en la automatización de más de un proceso de negocio. La primera cualidad caracteriza al servicio como agnóstico, y la segunda como reusable.

El poder liberar los servicios de vínculos altamente acoplados con procesos de negocios específicos o con detalles de implementaciones propietarias promueve la visión de construir un inventario de servicios que puede ser reutilizado a través de recomposición a medida que nuevos requerimientos surgen. Debido a su potencial de reuso, servicios agnósticos y bien diseñados proveen el valor más significativo a este inventario [151, 131].

Reusabilidad y modelos de servicios

Los ***modelos de servicios*** proveen un medio para planificar y construir servicios con reuso potencial; y de igual forma, brindan un criterio para distinguir contextos en dónde se pueden reutilizar o no reutilizar servicios. De hecho, la fundamentación para construir modelos de servicios es la identificación y definición de servicios agnósticos. Es por ésto, que los modelos de servicios juegan un rol fundamental en el fomento de la reusabilidad a partir de un inventario o catálogo de servicios. El enfoque agnóstico para la entidad y utilidad de los servicios, está claramente previsto para brindar un contexto funcional adecuado para el encapsulamiento de una porción de lógica reutilizable.

7.2.4. Plano del Inventario de Servicios Web

El fin último de un esfuerzo de transición hacia una arquitectura empresarial SOA es producir una colección de servicios estandarizados que conformen un *inventario de servicios*. El inventario puede ser estructurado en capas de acuerdo a los modelos utilizados y a su granularidad. Hay que reconocer que lo que posiciona a los servicios como bienes informáticos empresariales es la aplicación del paradigma de orientación a servicios a cada uno de ellos, en concordancia con los objetivos estratégicos asociados con el proyecto SOA. Sin embargo, antes que cualquier servicio sea realmente construido, es aconsejable establecer un plano conceptual de los servicios candidatos para un determinado inventario. Esta perspectiva es documentada en un ***plano de inventario de servicios*** *(service inventory blueprint)*.

El ***plano*** *(blueprint)* o ***modelo de Inventario de Servicio*** provee un mecanismo para organizar todos los servicios a través de una empresa. Muestra en forma integral el conjunto de servicios y las relaciones entre estos; puede decirse que presenta el siguiente nivel de detalle de las capacidades de un dominio de información, luego de la identificación de los procesos de negocios. Se puede pensar en un modelo de inventario como un mapa de responsabilidades de las capacidades y de interfaces de servicios. El modelo provee una forma bidimensional de organizar los servicios; una dimensión se relaciona con el tipo de servicio, especificando su utilidad,

fundamentación y alcance; la otra relacionada con las divisiones de dominios de negocios en los cuales interactúa [278].

Como fuente principal para la construcción del plano de servicio, está la información y el conocimiento contenido en diferentes modelos de negocios y de datos, como puede ser modelos de entidades de negocios, lógicos de datos y de mensajes, ontologías y cualquier otra información estructurada en modelos.

El objetivo principal del ***plano de inventario de servicios*** **(service inventory blueprint)** es definir una completa perspectiva de la lógica de solución establecida en la empresa (o dentro de un dominio predefinido) representada por un inventario de servicios candidatos. Definir esta perspectiva requiere un esfuerzo significante y un gran nivel de compromiso y acuerdo entre los analistas organizacionales, sistemistas y arquitectos informáticos.

Cada alcance de servicio es cuidadosamente modelado de manera tal que representa adecuadamente el contexto funcional del servicio sin sobreponerse los límites. Además, generalmente la forma en que los servicios se relacionan entre si es planificada, de manera que se puede realizar una mejor evaluación y definición del tipo y cantidad de lógica que cada uno debe encapsular.

La disponibilidad de un plano de inventario de servicios reduce dramáticamente el esfuerzo y el riesgo asociado al diseño y servicios reutilizables. Los servicios candidatos establecidos en este modelo, conforman la base de los contratos de servicios con alcances bien definidos. Este modelo es el que provee los contextos funcionales agnósticos y noagnósticos (menos agnósticos) para servicios, posibilitando la identificación y clasificación inicial de la lógica de los servicios con potencialidad de reuso.

7.2.5. Reuso Estandarizado de Servicios Web

Centralización de Lógica

La reusabilidad representa una característica clave que generalmente debe ser lograda a gran escala para que algunos objetivos estratégicos de la orientación a objetos se logren. Para alcanzar estos objetivos, el reuso en sí debe formar la base para soportar los diseños internos estandarizados. La misión más importante de estos estándares es establecer a los servicios clasificados como reutilizables en el único medio por el cual la lógica que ellos encapsula sea accedida. Esto conduce a lograr patrones de diseño basados en estándares conocidos como *Centralización de Lógica*.

La centralización impone de por sí, una limitación. Sin embargo, esta limitación ayuda a mantener normalizado un plano de inventario de servicios. Con un inventario de servicios normalizado, cada servicio representa un dominio de funcionalidades distintas, lo cual asegura que los límites y alcances de cada dominio no se sobrepongan. Respetando esta perspectiva, los servicios deben ser posicionados como los puntos de accesos oficiales (y únicos) para la lógica que estos encapsulan. El nivel de realización de la Centralización de Lógica determina la extensión del mismo como estándar empresarial.

La Centralización de Lógica introduce reglas que regulan la forma que una lógica de aplicación basada en servicios es construida en relación a un inventario de servicios, el cual actúa como un repositorio central de servicios agnósticos. A pesar que este patrón fundamental de diseño se aplica a todos los servicios, son los servicios reutilizables los que requieren especial atención; ya que si un servicio tiene una lógica específica, ésta no es tan propensa a ser duplicada por otro servicio.

Cuando la solución para un nuevo proceso ha sido creada, existe siempre el riesgo que el equipo de proyecto construya una lógica que ya existe en un servicio o servicios reutilizables. Las razones comunes son:

- El equipo de proyecto no es consciente de la existencia del servicio o sus capacidades debido a que el servicio no es fácil de descubrir o interpretar.

- El equipo de proyecto se niega a usar el servicio debido a que considera que es difícil o costoso hacerlo.

El primer impedimento puede ser evitado a través de la existencia de un registro central de servicios y mediante la aplicación de principios que obliguen o promuevan la búsqueda y descubrimiento de servicios a través de criterios de funcionalidad. El último impedimento es generalmente solucionado mediante el uso de un estándar empresarial de diseño que determine la obligación de usar servicios reutilizables, aún si estos no contienen toda las funcionalidades requeridas.

Centralización de Contrato

Los servicios expresan su propósito y capacidades bajo la figura conceptual de un *contrato de servicio*. El principio de diseño ***Contrato Estandarizado de Servicios*** *(Standardized Service Contract)* es uno de los más fundamentales en la orientación a servicios, ya que establece qué consideraciones deben tomarse en cuenta cuando se diseña la interfaz pública del un servicio y determina la cantidad y naturaleza del contenido que será publicado como el contrato oficial del servicio.

Por su parte, la Centralización de Contratos es considerado un patrón de diseño que se enfoca en el establecimiento del contrato oficial del servicio como único punto de entrada a la lógica del servicio. En este contexto, mientras la Centralización de Lógica exige a los desarrolladores construir aplicaciones clientes que solamente invoque a servicios diseñados e inventariados, en el momento que se requiera realizar un determinado tipo de procesamiento de información; la Centralización de Contratos obliga a los desarrolladores a que esas aplicaciones clientes solamente puedan acceder a los servicios a través de su contrato público/publicado. Combinar estos patrones forma una arquitectura que no sólo es estandarizada, sino que es naturalmente normalizada en la cual son intrínsecamente promovidas las relaciones con bajo acoplamiento entre consumidor y proveedor de servicios.

Cuando se implementan servicios centralizados utilizando tecnología de Servicios Web, se incrementa el énfasis en el contrato del servicio. El hecho que las definiciones WSDL, XML Schemas y WS-Policy deban representar endpoints oficiales (por soportar Centralización de Contratos) a segmentos de lógicas oficiales y estándares (por respetar la Centralización de Lógica), requiere que los detalles de contrato sean cuidadosamente diseñados para adecuarse al rol del servicio como recurso centralizado.

Estas consideraciones necesitan ser balanceadas teniendo en cuenta las posibilidades de centralización de los esquemas y políticas subyacentes de los servicios. Los esquemas XML y las políticas pueden establecer capas arquitecturales que pueden limitar la habilidad de adecuarse a los requerimientos específicos. Sin embargo, éstos pueden lograrse con la mezcla correcta de definiciones de esquemas y políticas generales y específicas. Llevar a cabo la Centralización de Lógica puede requerir un esfuerzo enorme para instaurarse en las bases de la empresa. En grandes organizaciones, lograr un estado en el cual todos los equipos de desarrolladores estén de acuerdo en reusar servicios en lugar de construir lógica redundante en ocasiones parece una idea irreal [422].

Muchos otros patrones de diseños arquitecturales existen para adecuarse es esta cuestión, el más destacado es el patrón de Inventario de Dominio, el cual permite que los requerimientos de estandarización sean contenido dentro de inventarios específicos de dominios de información que representen, en forma individual, partes de la organización. La realización de este patrón se vincula directamente con la adopción de la Centralización de Lógica; y además, logra que las iniciativas orientadas a SOA procedan en fases, con bases en dominios acotados.

Existe la percepción errónea que para lograr los beneficios estratégicos relacionados con los principios de SOA, éstos deben ser implementados a lo largo y ancho de la empresa. En ambientes organizaciones grandes, esta premisa puede llevar a una organización a tomar más allá de lo que

puede manejar, en términos de los alcances y magnitudes de los cambios que debe sufrir la empresa en la transición hacia SOA. En ocasiones, esta situación traumática, lleva a que la organización rechace todo principio de orientación a servicios. No hay dudas que definir y lograr una arquitectura orientada a servicios global e integral es ideal. Sin embargo, como se explicó previamente, no es la única opción. Para algunas organizaciones, un enfoque basado en dominios de la única forma viable para lograr adoptar SOA.

7.2.6. Reusabilidad y Diseño orientado a servicios

La realización del principio de reusabilidad puede requerir que se deban cambiar las prioridades de diseño y el método de construcción de software. La reusabilidad, junto con la autonomía y la capacidad de descubrimiento de servicio, son los principios de la orientación a objetos que deben ser tenidos en cuenta en la etapa de análisis. Cuando se conceptualiza los servicios como parte de una actividad de modelamiento, se debe definir y establecer los servicios candidatos teniendo en cuenta el principio de reusabilidad como un norma empresarial.

En relación con la reusabilidad de servicios y el diseño, es interesante explorar lo siguientes aspectos:

- El refinamiento de las capacidades candidatas existentes o definidas de servicios, con el propósito de lograr servicios mas genéricos y reusables.
- La definición de nuevas capacidades candidatas que vayan más allá de la funcionalidad requerida para la automatización de un proceso de negocios que forma la base para el proceso de modelamiento de servicios.

El último ítem es más propicio de lograr cuando se modelan servicios antes de construirlos, debido a que no se ha logrado ni un diseño físico ni una implementación. Por ende existe un mínimo riesgo de explorar formas en las cuales un servicio puede ser extendido para proveer un conjunto de capacidades reutilizables. Debido a que los servicios están siendo definidos durante una etapa preliminar de análisis, existe una verdadera oportunidad de obtener el conocimiento de los expertos de las funciones organizativas propias de la empresa, quienes pueden no volver a ser involucrados en las etapas subsiguientes.

Cualquier capacidad adicional definida en el modelamiento es simplemente candidata a implementarse o no. Sin embargo, documentar tal modelo permite proveer un mejor entendimiento de la dirección en la cual evoluciona el servicio. Además, poder establecer un amplio rango de capacidades candidatas en el modelado, permite mejorar la elección y determinación de prioridades respecto a las capacidades que deben entregarse de inmediato, a corto y a largo plazo.

Reusabilidad de Servicios y Granularidad

Darle el lugar a un servicio como recurso reusable de la empresa tiene directa influencia en las consideraciones de granularidad. A continuación se detalla algunas facetas de reusabilidad y su dinámica respecto a la aplicación de reusabilidad.

Granularidad de Servicios. En primer lugar, la ***granularidad de servicio*** generalmente se racionaliza con la aplicación de reusabilidad. Esta racionalidad surge por entender que un enfoque más acotado a la hora de diseñar un servicio, hace que sea más reutilizado por estar menos equipado que un servicio con alta granularidad. Reducir la granularidad de servicio también distribuye las demandas de procesamiento y balanceo de carga de los servicios agnósticos por tener más de una implementación a través de servicios. Aunque el principio de reusabilidad de servicios conduce a lograr niveles de baja granularidad, existen repercusiones en la performance relacionadas con la composición de numerosos servicios de bajo grosor que deben ser cuidadosamente consideradas.

Granularidad de Capacidades. Se pueden crear capacidades reutilizables de servicios para soportar una variedad de escenarios de uso y de valores de entrada y salida. Este enfoque conduce a las capacidades con alta granularidad sean más reusables debido a que son, en verdad, extensiones multipropósitos de los servicios. Sin embargo, esto puede resultar en un exceso en el intercambio de datos y procesamiento entre consumidores y productores de tales capacidades de servicios. Las capacidades, operaciones o funciones pueden ser tan altamente granuladas que se retorna una determinada cantidad de datos a los consumidores, los cuales necesitan usar sólo una pequeña parte de esta. Por estas situaciones, el principio de reusabilidad de servicios debe encontrar un balance en la granularidad de las capacidades que lo servicios exponen. No se descarta la posibilidad de que los servicios puedan ofrecer grupos de versiones para una misma capacidad; cada una de estas versiones con diferentes grados de granularidad.

Granularidad de Datos. El estilo de mensaje centrado en documentos de los servicios ha sido un reductor de la granularidad en datos; en especial, cuando se lo compara con el intercambio de datos basados en parámetros de baja granularidad visto en las soluciones basadas en RPCs (Véase Sec. 3.2). Sin embargo, los patrones de mensajes y el procesamiento adicional que viene con el uso de marcos de trabajos para la gestión de mensajes en soluciones basadas en Servicios Web, conllevan a la necesidad de aumentar la granularidad en el intercambio de mensajes. Por ejemplo, cuando se utiliza el marco de trabajo propuesto por WS-ReliableMessaging (Véase Sec. 6.6.2), se logra que los mensajes SOAP sean autónomos y autocontenidos, haciendo que el mensaje en si, como unidad de información, sea altamente granulado.

Granularidad de Restricciones. Para fomentar el reuso de las capacidades de servicios tanto como sea posible, se requiere que las mismas sean ampliamente consumibles. Esto conlleva al énfasis en la reducción de la lógica de validación, logrando asi que las restricciones de uso sean altamente granuladas.

7.2.7. Reusabilidad y otros principios SOA

Siendo una parte prominente de la orientación a servicios, el principio de reusabilidad de servicios influye a otros principios. Sin embargo en estos casos, el reuso no altera la forma en la cual estos principios son aplicados. La reusabilidad simplemente enfatiza la importancia de las características de diseño que estos principios promueven, para lograr cualidades requeridas en un servicio efectivamente reutilizable.

Reusabilidad y Contrato Estandarizado de servicios. Como la idea de estandarización de contratos se aplica a cualquier servicio, los detalles de la interfaces, como ser los tipos de datos de los parámetros y las precondiciones de invocación, son infuenciados teniendo en cuenta la reusabilidad en el sentido de mantenerlos lo más genéricos posible. Un servicio reutilizable necesita ser lo suficientemente flexible para soportar múltiples tipos de consumidores con diferencias razonables en los requerimientos de interacción. Esta exigencia puede inspirar al diseño de estándares que reducen las restricciones de validación de contrato, especialmente aquellas susceptibles a cambios.

Reusabilidad y Abstracción. Para lograr maximizar el potencial de reuso en los servicios, existe una inclinación a realizar los contratos lo más autodescriptivos posibles. Sin embargo esto debe ser balanceado con la consideración que los servicios reutilizables están diseñados para soportar la mayor cantidad de consumidores de servicios posibles, muchos de los cuales son desconocidos en el momento de implementación del servicio. Por ende, es importante que un contrato de servicio sea lo suficientemente consistente de manera que no se describa ni se restrinja en una forma que inhiba su futuro reuso. Estas consideraciones son tomadas cuando se

define y se diseña el tamaño y la calidad de la metainformación que debe ser abstracta, en el contrato de un servicio.

Reusabilidad y Bajo Acoplamiento. El principio de reusabilidad es un potenciador para el bajo acoplamiento entre servicios, por considerar que a menores requerimientos de dependencia, más fácilmente se genera el reuso. Por ende, cuando se persigue reusabilidad en la lógica de servicio, existe la tendencia general de reducir las restricciones e interdependencias en el contrato de los servicios. Por ejemplo, es más fácil mantener un servicio con su cualidad de agnóstico, cuando se pueden diferir la especificación de las reglas de validación por fuera del contrato y de la lógica subyacente del servicio. A pesar que este enfoque incremente el procesamiento de la lógica, se previene en el diseño original del servicio posibles incopatibilidades en los futuros consumidores del servicio. Incrementar la "longevidad" y el espacio de vida de un contrato de servicio permite extender la disponibilidad y el valor de un servicio como recurso reutilizable. La reducción y minimización del grado de acoplamiento es un medio efectivo para lograr ésto.

Reusabilidad y Autonomía. Debido a que las demandas potenciales de performace y uso concurrente de servicios reutilizables, es una consideración importante de diseño el garantizar el control que dichos servicios pueden ejercer sobre los potenciales ambientes subyacentes, en un nivel aceptable de comportamiento predecible. Por ejemplo, los requerimientos de centralización de lógica y de composición de servicios pueden demandar que los servicios tengan autonomía incremental para mantener dicha garantia a medida que la calidad de los programas consumidores crece y se vuelve más complejas las composiciones de servicios.

Reusabilidad y Descubrimiento. Para que se realice el reuso a escala empresarial y la centralización de lógica, los servicios reutilizables deben ser fácilmente descubiertos e interpretados. Un servicio reutilizable, por ser parte de un esfuerzo serio para construir un inventario o catálogo de servicios, debe ser equipado con toda la metainformación requerida para ser localizado, y su propósito y capacidades correctamente entendidas.

Reusabilidad y Composición. La composición de servicios puede verse como una forma de reuso. El objetivo del principio de reusabilidad en este contexto no es producir servicios para un tipo específico de composición; sino lograr que cada servicio sea un participante efectivo de varias composiciones con diferentes configuraciones. Mientras más reutilizable sea el servicio, más serán las oportunidades de participación en diferentes composiciones.

7.2.8. Aplicación de la Reusabilidad en Servicios Web

En resumen, para que un servicio sea reutilizable, debe encerrar una lógica de solución que sea agnóstica para fomentar su reuso en varios procesos o actividades de negocios. La capacidad implementada por dicha lógica debe ser descripta mediante un contrato de servicio, el cual deben cumplir las aplicaciones consumidoras que lo utilicen. Dicho contrato y la descripción de tales capacidades deben ser publicadas para que se puedan buscar y acceder si cumplen los requerimientos de un nuevo sistema. Debe existir un servicio que gestione tales descripciones y brinde los medios para localizacción y acceso a los servicios reutilizables. Los servicios, al constituirse en bloques de construcción, deben ser propensos a la composición, teniendo su adecuado nivel de granularidad, para fomentar su reutilización y oportunidades de reuso.

Los Servicios Web son el marco de trabajo para implementar verdaderos servicios reutilizables que adopten el paradigma SOA. Y cada una de las tecnologías que ofrecen cumplen con el propósito general de SOA y particular del principio de Reusabilidad de Servicio.

Por un lado, todo servicio web es un módulo de software que implementa una determina lógica de aplicación que expone una capacidad. La condición de que dicha lógica sea lo suficientemente

agnósticas o no es una decisión de diseño; sin embargo su capacidad es expuesta a través de la web, utilizando protocolos estándares basados en XML: para su descripción (WSDL), descubrimiento (UDDI) y acceso (SOAP, HTTP). Como se explicó en la Sec. 5.3, uno de los principales aportes de los Servicios Web en sistemas distribuidos consiste en lidiar efectivamente con la heterogeneidad de plataformas de desarrollo y de ejecución; de hecho, se presenta a dicha tecnología como la última evolución en el concepto de middleware. De esta forma, se solucionan las barreras de interoperabilidad e integración, quedando solamente la decisión de (re)uso limitada por la adecuación o no de dicha lógica a las necesidades del cliente.

Al elegir Servicios Web como tecnología de desarrollo, no se realiza ninguna suposición respecto a las potenciales aplicaciones clientes; en este sentido el desacople es completo. Esto favorece a la adopción de contextos agnósticos exigidos en una arquitectura SOA. Eventualmente se puede dotar a un servicio web de la adecuada portabilidad y autonomía a diferentes ambientes de ejecución, llegando a niveles de ubicuidad coherentes con los ambientes agnósticos requeridos en el diseño orientado a servicios.

La adopción del principio de Contrato Estandarizado de Servicios encuentra un excelente medio de realización en las descripciones WSDL contenidas en el mismo servicio. Adicionalmente, es posible establecer criterios más finos del contrato con el uso de políticas establecidas con normas como WS-Policy.

Los Servicios Web también brindan un medio tecnológicos para implementar el principio de composición de servicios, siendo uno de los escenarios más ricos para posibilidades de reuso. Se cuenta con marcos de trabajos para lograr conversaciones o interacciones de servicios (WS-Composition) y especificaciones estándares para implementar modelos de composición a nivel de proceso de negocios, orquestación (WS-BPEL) y coreografía (WS-CDL). Por ejemplo, un script ejecutable WS-BPEL implementa una proceso de negocios cuyas actividades son llevadas a cabo por servicios web. A medida que esos servicios web sean más agnósticos, puede reiterarse su uso en varios script WS-BPEL.

El uso de Registros de Servicios UDDI de los servicios web de una empresa, brinda soporte para el Inventario de Servicios, esencial a la hora de buscar oportunidades de reuso, sean éstas producidas en forma dinámica o en base de requerimientos de diseño. Un servicio centralizado de registro da efectiva implementación a la centralización de contratos y de lógica. Una vez que el inventario de servicios web es relativamente maduro, los servicios reutilizables participarán en un número incremental de composiciones.

En cualquier ambiente de desarrollo que soporte la construcción de servicios web y en base a servicios web, se debe llevar un sólido *sistema de control de versiones*, para permitir la evolución de los contracto de los servicios web reutilizables.

Además se debe contar con analistas y diseñadores de servicios web con un alto grado de experiencia para asegurar que los límites y contratos de los servicios web representan adecuadamente el contexto funcional y reusable del servicio. Deben poseer una pericia adecuada para estructurar la lógica subyacente de los servicios en componentes y rutinas genéricas, potencialmente componibles y agnósticas. Al ser Servicios Web una tecnología de rápida adopción y basada en estándares libres, lograr recursos humanos calificados es menos trabajoso que otras tecnologías distribuidas previas.

7.3. Reutilización de Aplicaciones Legadas mediante Servicios Web

En la actualidad las Tecnologías de la Información juegan un rol cada vez más grande para lograr el éxito en los negocios. Los sistemas informáticos creados, manejados y mantenidos cumplen el rol de motores de una empresa, llevando a cargo funciones críticas y suministrando información sencible para la toma de desiciones. A medida que se requieren cambios en los ambientes empresariales, los procesos de negocios y funciones organizativas atraviesan los límites

de las empresas y comienzan a ser más complejos [45].

Los departamentos de sistemas deben encontrar una forma eficiente y económica para la promoción y extención los sistemas existentes para soportar nuevas necesidades. Las organizaciones deben permitir que la comunicación e integración entre las entidades de un proceso de negocio sean flexibles y bajamente acopladas. Las aplicaciones altamente acopladas deberían ser transformadas en unidades de construcción reutilizables para los futuros sistemas. Esto significa que las aplicaciones y sistemas deberían interactuar a pesar de su lenguaje de desarrollo, plataforma o configuración de red. Las infraestructuras informáticas, que fueron originariamente construidas teniendo a la performance como primicia, deben reestructurarse enfocándose en el reuso.

Esto origina varios desafíos en términos de comunicación, reusabilidad, interoperatividad, performance y seguridad. Cualquier respuesta a estas problemáticas debe tener en consideración que los métodos de integración pueden darse entre aplicaciones del mismo departamento, traspasando los límites de la empresa, o una combinación de ambas. Estos desafíos requieren que las organizaciones reestructuren su infraestructura informática. Varios estándares han sido desarrollados para reducir el costo de desarrollo y mantenimiento de las infraestructuras informáticas como son los estándares de objetos o componentes. Como se explicó en la Sec. 5.3, los Servicios Web son una tecnología que se presenta como último escalón en la evolución de los middleware, siendo su principal objetivo aliviar las barreras de comunicación, interoperatividad e integración de aplicaciones, las cuales pueden recidir en ambientes heterogéneos. De la misma forma, en la Sec. 7.2.8, se presentó los Sevicios Web el principal medio tecnológico para lograr una infraestructura SOA en una empresa; siendo estos fundamentos esenciales para lograr la dinámica en las estructuras de software adecuada para los cambios y adaptaciones requeridos según nuevos procesos negocios se conformen o se modifiquen en una organización.

Como nuevo paradigma de desarrollo de software, SOA requiere que se modifiquen y adapten las infraestructura informática, y además las prácticas de diseño y desarrollo de aplicaciones de una organización. Generalmente, una empresa al decidir optar por tener arquitecturas SOA en su patrimonio informático, también asume un proceso que llevará tiempo y dedicación. No se puede suplantar las aplicaciones existentes con aplicaciones orientadas a servicios, desarrolladas desde cero, de un día para el otro. Existe un proceso gradual que se debe respetar para lograr una adecuada asimilación en la normalización y práctica de la orientación a servicio; y por ende también, en la cultura de reuso que viene con ésta. Es por eso, que una de las aplicaciones que surgieron primero de Servicios Web, fue servir de puente entre aquellas aplicaciones legadas, no desarrollada en forma orientada a servicios, y las arquitecturas SOA que surgen a raíz de los nuevos esfuerzos de adopción. Dicho de otra forma, las aplicaciones legadas encontraron en los Servicios Web, el medio tecnológico para exponer sus capacidades y funcionalidades hacia ambientes heterogenos, para los cuales no fueron diseñadas.

7.3.1. Concepto de Aplicaciones Legadas

El término ***"sistema legado"*** *(legacy system)* describe a un antiguo sistema informático en operación dentro de una organización. Generalmente, los sistemas legados han sido desarrollados de acuerdo a prácticas y tecnologías que hoy se consideran obsoletas; también, tiene una larga vida y han sido modificados en forma extensiva. Los análisis modernos tienden a clasificar a las aplicaciones como *"legadas" (legacy)* dependiendo de su tiempo de uso y/o por utilizar una plataforma, lenguaje o tecnología de diseño antigua. Sin embargo, hoy en día estas aplicaciones legadas son verdaderas inversiones que soportan y ejecutan procesos de negocios críticos para la empresa propietaria de las mismas [10]; además conllevan un alto costo de mantenimiento. El valor de estos sistemas legados se incrementa debido a que estos sistemas encapsulan sustanciales conocimiento y significativas funciones, los cuales son accedidos y empleados en un formato ya asimilado y comprendido por los usuario que lo utilizan desde hace tiempo [498]. Sin embargo, la carga de mantenimiento de estos sistemas es proporcional a su edad; es asi que eventualmente las desventajas de los sistemas legados pesarán más que el valor que le brinda a la organización

que los usa, exigiendo su modernización hacia nuevas tecnologías [64].

Esto establece un dilema: dado un sistema legado, se debe elegir entre preservarlo y mantenerlo operativo asumiento un costo altísimo de mantenimiento y limitando su adaptación a cambios futuros; o emprender una tarea de modernización que preserve su lógica de aplicación y mejore sus perspectiva de escalabilidad e interoperatividad. Por ende, las organizaciones hacen frente a decisiones difíciles respecto al futuro de tales sistemas. El castigo por tomar la decisión incorrecta podrá ser más onerosa para la empresa. Enfrentar a los futuros cambios usando técnicas de mantenimiento puede tornarse demasiado costoso y fallar en satisfacer los nuevos requerimientos. En este escenario, cualquier esfuerzo de inversión realizada en el sistema legado, será una pérdida. En paralelo, reemplazar el sistema puede ser muy costoso; además, el conocimiento de negocio encerrado en el sistema corre el riesgo de perderse, resultando en el desarrollo de un sistema que tampoco cumpla con los requerimientos [527].

Muchos sistema legados fueron desarrollados previos a la programación estructurada. Los modelo de procesamiento y los principios básicos de modularidad, acoplamiento, cohesión y buena programación emergieron tiempo después. Estos sistemas, en el peor de los casos, fueron desarrollados con procedimientos ad-hoc y con técnicas de programación que limitan seriamente su escalabilidad e integración.

Existen tres razones por la cuales muchas aplicaciones o sistemas legados no fueron diseñados para ser propensos al cambio:

- *Expectativas de corta vida.* En el momento de diseño de las aplicaciones, no fue provisto que se conviertan en sistemas legados décadas más tarde.

- *Falencias en los modelo de procesos e ingeniería de software.* Las culturas adoptadas al respecto no concebían a la evolución y escalabilidad como actividades fundamentales en la practica de diseño y desarrollo. Se pueden extraer los requerimientos de evolución de los objetivos empresariales, pero de acuerdo a las prácticas antiguas estos requerimientos eran ignorados durante la fase de especificación en el desarrollo.

- *Satisfacer restricciones existentes en el tiempo de desarrollo.* Por ejemplo: el hardware es actualmente considerablemente más barato y poderoso en comparación con los tiempo de gestación de aplicaciones legadas. En las décadas pasadas, el poder de procesamiento y la cantidad de memoria fueron limitantes enormes para elegir el diseño de software. Se emplearon técnicas para hacer un uso económico de tales recursos, a expensas de la manutención del sistema. Para sistemas de larga vida, la manutención es una medida fundamental de calidad.

Las aplicaciones legadas pueden ser divididas en tres categorías básicas, con respecto al grado de dependencia que tienen con su ambiente [309]:

- *aplicaciones que no son dependientes de su ambiente*: son aquellos programas escritos en lenguajes de programación convencionales como Fortran, COBOL y C/C++. Estos programas pueden ser rápidamente reutilizables en cualquier ambiente que provea un compilador del lenguaje en cuestión.

- *aplicaciones que son parcialmente dependiente de su ambiente*: encierra a los programas escritos en lenguajes de programación que usan intérpretes de tiempo de ejecución o vínculos a funciones o librerías de sistemas. A estas categoría pertenecen aplicaciones desarrolladas en PL/I, Smalltalk y Forté 4GL [476]. Estos programas pueden ser reutilizados en otros ambientes, con la única exigencia que sus rutinas de tiempo de ejecución sean sustituidas por módulos acordes a la nueva plataforma.

- *aplicaciones totalmente dependiente de su ambiente*: consisten en los programas desarrollados en lenguajes de 4ta generación, como ser ADS-Online, Natural [443], CSP y Oracle

Frames, los cuales requieren una ambiente específico para ejecutarse. Tales sistemas no pueden utilizarse en otros ambientes; son ambiente-dependientes. Por ende, la única forma de reutilizar estos programas es mantenerlos en su ambiente nativo y construir vínculos de ejecución a esos ambientes.

Lo que se puede concluir de esta observación es que, mientras más elemental sea el lenguaje de programación, son más fácil de reutilizar sus aplicaciones. Esta es una consideración importante que deben realizar los gestores y administradores de sistemas, al elegir una tecnología de desarrollo. Se debe optar por alta productividad en un corto rango, o por alta reusabilidad y protabilidad en un amplio rango [527].

La combinación de proceso de desarrollo técnicas y tecnologías obsoletos, en conjunción con el tiempo de vida extendido y los reiterados cambios, generan sistemas que sufren los siguientes padecimientos [527, 498, 64, 82]:

- *Deficiencias de Aplicación*: Falta de documentación, módulos altamente acoplados, arquitectura monolítica no-modular, falta de consistencia en la arquitectura subyacente (código, funciones y datos redundantes), dificultad para la integración y escasas interfaces con tecnologías distribuidas abiertas, limitaciones en la interfaz de usuario y asistencia de ayuda, y otras.

- *Efectos colaterales*: alto costo de licenciamiento, baja agilidad para el cambio, conocimiento encerrado en las aplicaciones cautivo en pocas personas, limitación en la integración de funcionalidades, limitado soporte de flujo de trabajo, no aprovechamiento de nuevas tecnologías, limitaciones y riesgos inherentes a la utilización de una única plataforma, y otros.

- *Deficiencias para el cambio y adaptación*: escasez de desarrolladores capacitados disponibles para las tecnologías antiguas, reducción de soporte del vendedor de la plataforma y herramientas de desarrollo, dificultad para entrenar nuevos desarrolladores y usuarios finales, y otros.

7.3.2. Modernización de Aplicaciones Legadas

La idea de exponer a componentes o aplicaciones legadas como servicios web, es una forma de extender su ámbito, vida útil y funcionalidad. En sí, puede verse como un proceso de *modernización* del sistema legado [138]. Una ***modernización*** de este tipo puede requerir más cambios que el simple mantenimiento del sistema legado, pero conserva grandes porciones del sistema existente, de ahí que se observa el potencial y posibilidad de reuso de este último.

La modernización de sistemas de software puede ser llevada a cabo en diferentes formas. Dentro de las diferentes ideas propuestas [64, 65, 138] se puede hablar de: *re-desarrollo*, *envolturas* y *migración*.

Modernización a través de Re-Desarrollo

El re-desarrollo es comúnmente referenciado como el enfoque *Big-Bang* o *Cold Turket* [82]. Consiste en re-desarrollar la lógica de negocios contenida en el sistema legado desde cero, utilizando nuevas plataformas de hardware, arquitecturas de software, herramientas y bases de datos [64]. Produce el reemplazo completo de un sistema legado que no cumple actualmente con las necesidades de la empresa y son inaplicables las otras formas de modernización. El re-desarrollo o ***reemplazo*** es normalmente usado con sistemas que estan indocumentados, no se dispone de su código fuente, o no son extensibles [138].

Como ejemplo de esta forma de modernización el proyecto *Renaissance* [152] propone un método sistemático para la evolución de un sistema y su reingeniería. El proyecto define un conjunto de actividades y tareas para soportar la totalidad de un proyecto de reingeniería y el flujo

de control entre las actividades identificadas, las cuales conducen a la cooperación entre tareas. *Renaissance* identifica actividades genéricas que pueden ser especializadas para la organización y el sistema legado que se implementa en particular [527].

Por otro lado, existe la propuesta de utilizar marcos de trabajos de alto nivel de abstracción para la reingeniería de sistemas legados desde varias *perspectivas* [501]. Cada *perspectiva* divide el problema de reingeniería en fases, cada una de las cuales tiene problemáticas a resolver. Esté método puede constituirse como una adecuada guía del proceso de reingeniería, pero sin embargo en ocasiones es demasiado general para ser efectivamente aplicado.

Otros autores [184] han propuesto criterios de acción generales para la transformación de un sistema legado establecido en un ambiente centralizado hacia un ambiente distribuido. Teniendo en cuenta esto, se seleccionan procesos de negocios a re-desarrollar y migrar a ambientes distribuidos y luego se los vincula con la aplicación legada, la cual provee los datos y la lógica de negocios para el nuevo ambiente. Es decir, que las nuevas aplicaciones son desarrolladas para encajar en estos procesos.

Un enfoque metodológico general para el re-desarrollo basado en SOA propuesto en los últimos años [548], establece el análisis de código legado para obtener un entendimiento acabado de su funcionalidad. Este análisis es utilizado para identificar procesos de negocios soportados por la aplicación legada. Estos procesos de negocios identificados son implementados (desde cero) como servicios; por ende, conlleva virtudes de un diseño SOA, como ser el fácil mantenimiento, su extensibilidad y reuso potencial.

El re-desarrollo de software tiene ventajas respecto a las otras formas de modernización. Al ser un software completamente nuevo desarrollado con tecnologías actuales, no se requieren capas adicionales para vincularlo con las nuevas infraestructuras de hardware y las nuevas arquitecturas de software [438]. Como principal desventaja esta el riesgo que se corre al tomar una decisión de re-diseño y reemplado de software. El re-desarrollar un sistema desde cero es significativamente costoso en tiempo y dinero. El sistema legado sólo se lo utiliza para entender las funcionalidades que deben ser implementados; dicho de otra forma, se extrae el conocimiento del sistema legado. Este es un proceso de reingeniería lento y tedioso, y puede llevar errores ya que parte de ese conocimiento puede no ser debidamente abstraído y por ende no implementado. Visto desde este punto de vista, no existen garantías que el nuevo sistema sean tan correcto, robusto o funcional como el sistema antiguo [434, 451].

Modernización a través de Envolturas

Debido al gran riesgo en el que se incurre utilizando el re-desarrollo y reemplazo de aplicaciones, muchas organizaciones se vieron forzadas a buscar formas alternativas para lidiar con la problemática de aplicaciones legadas. La mayoría de las soluciones prácticas en este sentido se centran en el diseño y uso de ***envolturas*** *(wrappers)*, que encierran o "envuelven" datos, programas individuales, aplicaciones e interfaces existentes con nuevas interfaces. En esencia, esto brinda a los componentes legados nuevas operaciones y un aspecto nuevo y renovado [532]. La *envoltura* es un componente que actúa como un servidor, ejecutando alguna funcionalidad requerida por un cliente externo que no requiere saber cómo el servicio está implementado [435]. El uso de envolturas le permite a las organizaciones reutilizar componentes adecuadamente testeados a lo largo del tiempo, que gozan de confianza y constituyen una inversión ya realizada en sistemas legados.

Una de las técnicas mas utilizadas para envolturas es el *"rascado de pantalla" (screen scraping)* [15], generalmente utilizado para integración y modernización de arquitecturas 1-capa, como se explicó en la Sec. 2.2.1. Esta técnica sirve para envolver el nivel de presentación de un sistema legado basado en carácter, con un cliente que provee una interfaz gráfica u orientada a web [48]. El implementar interfaces GUI es más barato y efectivamente promueve la utilización de los datos legados y le permite a los usuarios emplear herramientas gráficas comunes para el manejo de las entrada de datos y procesamiento de las salidas. A pesar que la técnica de rascado de pantalla

tuvo un relativo éxito comercial, sigue siendo una respuesta a muy corto plazo; se adquieren otras deficiencias como la sobrecarga de una capa adicional, limitación de la escalabilidad y altos costos de mantenimiento [64].

Sin embargo, la creación de envolturas en base a Servicios Web es una de las técnicas más utilizadas y difundidas para exponer funcionalidad de sistemas legados en un ambiente SOA. En especial, las técnicas recomendadas y exitosas son enriquecidas con algunos procesos de reingeniería. Los conceptos y características de tal aplicación de Servicios Web, por ser de especial relevancia para el presente trabajo, serán explicadas en detalle en la sección siguiente.

Modernización a través de Migración

La ***migración*** *(migration)* en un sentido amplio es el propósito de todas las técnicas de modernización; sin embargo, en un sentido más estricto, es una variación del método de envolturas. El proceso de migración es similar, pero incorpora diferentes cantidades de re-diseño de aplicación y envolturas de componentes. El método general consiste en identificar y localizar el código legado, desacoplarlo y extraerlo por métodos similares. El paso siguiente es realizar reingeniería de las interfaces para conformarlas con una estructura SOA. Es decir que, no sólo es envuelto el componente completo, sino que también son re-diseñadas sus interfaces; las envolturas en estos casos son necesarias para implementar peticiones web que el lenguaje del componente legado no soporta. Como ejemplo de la técnica, se puede citar la migración de sistemas COBOL a una arquitectura SOA basada en web [501]. En este caso, la interfaz de servidor del sistema legado fue la pieza fundamental a ser envuelta para lograr integración en un nuevo sistema orientado a servicios. Esta forma minimiza la cantidad de envolturas que son requeridas para promover el diseño de la arquitectura y la migración a largo plazo del sistema.

Como ha pasado en varias situaciones, un método es la fusión y evolución de sus precedentes. Esta técnica de modernización es una mezcla de las dos anteriores; sin embargo, el nivel de granularidad de re-desarrollo y uso de envolturas es balanceado para producir soluciones adecuadas en términos de eficiencia, diseño y costos. Su método utiliza las mejores prácticas de las otras técnicas para lograr migrar sistemas legados a arquitecturas SOA [451].

La modernización de sistemas legados también puede ser clasificada según el nivel de entendimiento del sistema existente requerido para afrontar el esfuerzo de modernización o transformación [532]. Las modernizaciones de sistemas que requieren un conocimiento acerca de la arquitectura interna del sistema legado se denominan ***modernización de caja-blanca*** *(white-box modernization)*; en tanto que si solamente se requieren conocimientos de la interfaz externa del sistema, se está en presencia de una ***modernización de caja-negra*** *(black-box modernization)*.

Modernización de Caja-blanca

La ***modernización de caja-blanca*** requiere un proceso inicial de ingeniería inversa para lograr entender la operatoria interna del sistema. Los componentes internos del sistemas y sus relaciones son identificados y se produce una representación del sistema a un nivel de abstracción superior [99].

El ***entendimiento de programa*** *(program understanding)* es la principal forma de ingeniería inversa usada en las transformaciones de caja-blanca; su método involucra el modelamiento de dominio, extracción de información desde el código, y la creación de abstracciones que ayudan a entender la estructura de la aplicación subyacente [501]. Este proceso de análisis y compresión de lo que un sistema legado hace, es una tarea difícil y conlleva un alto riesgo, ya que la mayoría de estos sistemas tiene una enorme complejidad por los reiterados cambios realizados en su etapa de mantenimiento [518, 425].

Luego que el código legado es analizado y entendido, se procede a la etapa de reestructuración. La reestructuración puede ser vista como la transformación de una forma de representación

(o implementación) a otra con el mismo nivel de abstracción, mientras se preserva el mismo comportamiento externo del sistema; es decir, su funcionalidad y semántica operacional [99]. El principal motivo para realizar esta transformación es lograr algún atributo de cualidad que mejore o *modernice* el sistema legado como ser interoperatividad, facilidad de mantenimiento, o rendimiento. La partición y modularización del código legado es una técnica popular en la transformación de sistemas [265].

Modernización de Caja-negra

La ***modernización de caja-negra*** involucra el análisis de las entradas y salidas de un sistema legado dentro de su contexto operativo para lograr entender el contrato que encierra las interfaces de sistemas. A pesar que es una tarea difícil, especialmente si no se tiene documentación al respecto, tiene menos nivel de exigencia que la modernización de caja-blanca.

La modernización de caja-negra es generalmente basada en la aplicación de diferentes patrones de diseño para exponer las funcionalidades de una aplicación legada. Dicha tarea no resulta ser tan invasiva como las requeridas para caja-blanca. Las técnicas más comunes para exponer funcionalidades de esta forma son: las ***"envolturas"*** *(wrappers)*, los ***adaptadores*** *(adapters)* y los ***proxies*** [45]. Lo común entre estos patrones es que todos ellos son usados como interfaces entre aplicaciones que utilizan tecnologías y protocolos de comunicación incompatibles u obsoletos.

La diferencia entre estos patrones radica en que los adaptadores y proxies conocen los clientes con los que interectuarán. En ese sentido están altamente acoplados a la aplicación que exponen y sus clientes. Por otro lado, las envolturas están enfocada en proveer una interfaz que exponga al componente legado, sin tener previo conocimiento acerca de sus potenciales clientes. Las envolturas solucionan las problemáticas de heterogeneidad de plataforma, lenguaje de programación y redes; encierran al sistema legado con una capa de software que oculta la complejidad no deseada y exporta una interfaz moderna. Las envolturas son utilizadas para remover las diferencias entre las interfaces expuestas por los artefactos a reutilizar y las requeridas para las prácticas actuales de integración [523, 426].

Las envolturas son la técnica preferida en el mundo de Servicios Web a la hora de realizar modernizaciones de caja-negra. En esta tarea de reingeniería, idealmente se ignoran los detalles internos del sistema legado; sin embargo, esto no siempre se logra en la práctica por depender de la estructura algorítmica del sistema, y se requiere alguna técnica de caja-blanca para entender la composición modular de la aplicación legada [386].

Diseño Bottom-Up para modernizar aplicaciones

En ambos tipos de modernizaciones, de caja-blanca y de caja-negra, pueden utilizarse Servicios Web para exponer las funcionalidades de un artefacto de software legado. Al aplicarse Servicios Web en la solución, cada tipo de modernización tiene una estrategia de desarrollo diferente. Sin embargo, un factor común a estos métodos es la aplicación del principio de diseño *bottom-up*. En la secciones 2.1.2 y 2.1.3 se indicaba los diferentes diseños que puede tener un sistema informático distribuido. De igual forma un ambiente de Servicios Web puede ser construido siguiendo el método de diseño *top-down* o *bottom-up*. Como se explicó, el primer método es utilizado cuando se requiere que los servicios web sean desarrollados desde cero, como cuando se aplicaca la técnica de re-desarrollo. Por otro lado, el método *bottom-up* es usado para extender las funcionalidades de aplicaciones legadas preexistentes [45]; aplicándose generalmente al uso de envolturas y la técnica de migración.

En el método del diseño ***top-down*** el servicio tiene que ser diseñado e implementado para servir en un ambiente de Servicios Web puro. En otras palabras, debe ser descubrible, autocontenido y reutilizable. El servicio actúa como un consumidor o proveedor. Si es un consumidor o invocador de otros servicios, el servicio incluye la lógica para construir mensajes SOAP y enviarlos al servicio del cual se invoca una operación. Si el servicio es un proveedor, entonces su principal

propósito es esperar y escuchar peticiones. Cuando una petición llega, el proveedor determina que tipo de petición (mensaje) es, ejecuta la acción apropiada a dicha petición y retorna en un mensaje SOAP los resultados de la operación invocada. El desarrollo de aplicaciones en Servicios Web requiere un enfoque diferente al desarrollo de aplicaciones tradicionales. El diseño de un servicio web es hecho a través de herramientas de desarrollo que proveen GUIs para modelar y producir documentos WSDL que describe todos los servicios y tipos de datos que conforman a la interfaz y su contrato de servicio. Cuando la implementación de un servicio está culminada y su interfaz definida, la descripción WSDL es publicada en un Registro UDDI para que pueda ser accesible por otros servicios web. Como todo componente, el servicio web publica sus funciones e interfaz y mantiene privados los detalles de implementación.

Por otro lado, el método ***bottom-up*** es usado para extender las funcionalidades de las aplicaciones legadas. Las organizaciones y departamentos que gestionan la infraestructura informática necesitan re-estructurar estas aplicaciones legadas debido a que éstas están altamente acopladas a plataformas y sistemas propietarios; por ende, resultan difíciles de integrar con otras aplicaciones heterogéneas. Pueden existir limitaciones respecto al número de clientes que las pueden acceder, ya sea por licenciamiento o por limitaciones funcionales en las plataformas de middleware que las soportan. Estas aplicaciones legadas deberían poder trascender a los cambios en las tecnologías informáticas sin interrumpir los procesos de negocios que estas soportan.

Teniendo en cuenta estos conceptos explicados relacionados a las técnicas y tipos de modernización de sistemas legados, se profundizará en las técnicas y métodos para modernizar aplicaciones legadas hacia arquitecturas SOA. En general, las soluciones propuestas más exitosas usan la técnica de "envolturas" por ser la que, por principios y propósitos, más se ajusta a el paradigma SOA y a Servicios Web. Esta adopción brinda mayor desacoplamiento respecto a las aplicaciones clientes y a sus ambientes por manejar comunicación basada en mensajes SOAP. Además, permite publicar las interfaces de la aplicación envuelta con estándares WSDL y eventualmente permite su descubrimiento si se utiliza un Servicio de Registro UDDI. Todo esto redunda en lograr exponer a la aplicación legada con mayor abstracción, interoperatibilidad, autonomía, y con un potencial extendido de reuso. Los métodos propuestos aplican, en menor o mayor grado, el principio de diseño botton-up para realizar la reingenieria de las capacidades contenidas en la aplicaciones legadas y extender su funcionalidad a los ambientes Web Services y conformarse como bienes activos de una arquitectura SOA.

7.3.3. Envolturas de Servicios Web para Aplicaciones Legadas

Una *envoltura de Servicio Web* es un programa usado para permitir que una pieza de código existente de software interactue en un ambiente SOA, diferente para el que fue creado. Comúnmente se establece una interrelación que tiene a la aplicación legada en un extremo y al ambiente de Servicios Web del otro; esta interrelación da origen a dos tipos de escenarios:

- El primer escenario es cuando el servicio web necesita acceder a la aplicación legada. En este caso, la envoltura acepta peticiones de servicios web en la forma de mensajes SOAP. Por cada petición, la envoltura invoca a la función correspondiente en la aplicación legada para recuperar los resultados necesarios para completar la petición, convierte estos datos en una respuesta para el servicio web y envía la respuesta en forma de mensaje SOAP.

- El segundo escenario es cuando la aplicación legada necesita acceder al servicio web. En este caso, la aplicación legada coloca su petición en una porción de memoria compartida con la envoltura. La envoltura lee esta porción de memoria compartida (que puede ser un archivo compartido) e invoca a una serie de operaciones de servicios web. Una vez que se obtienen los resultados de las invocaciones, la envoltura los escribe en otra porción de memoria compartida que es accesible por la aplicación legada.

Enfoques	***JavaEE***	***.Net***
Sesión	Java Server Pages (JSP))	Active Server Pages (ASP)
Transacción	JDeveloper	.Net Framework
Datos	Java Database Connectivity (JDBC)	Open Database Conectivity (ODBC)

Cuadro 7.1: Herramientas para implementar enfoques del diseño de envolturas

Enfoques en el diseño de envolturas

La implementación de envolturas depende de la forma en que la aplicación legada será accedida, y esto plantea diferentes enfoques de diseño. Los enfoques para establecer envolturas de servicios web en aplicaciones legadas son: *basado en sesiones*, *basado en transacciones* y *basado en datos*. Cada uno de estos enfoques tiene relación con el nivel conceptual o de abstracción (explicados en la Sec. 2.1.1) por el cual se accede del sistema informático distribuido legado a través de la envoltura.

El enfoque basado en sesiones. Este enfoque es usado cuando la aplicación legada es escrita en una forma tal que la lógica de negocios y el nivel de presentación no están claramente separados, similar a las aquitecturas 1-capa presentadas en la Sec. 2.2. Se requiere construir una interfaz gráfica (GUI) sobre el nivel más alto de la aplicación legada. En este enfoque solamente se modifica la capa de presentación del sistema existente, dejando a la lógica de negocios y la gestión de recursos sin cambios. Este enfoque ofrece un modo de un "solo sentido" ya que la aplicación legada puede ser expuesta como servicio web, pero no puede consumir otros servicios web.

El enfoque basado en transacciones. Este enfoque es usado en una aplicación legada que tiene debidamente separados los niveles de acceso de datos, lógica de negocios y presentación, como lo presentan aquellas aplicaciones de arquitectura 2-capas ó 3-capas. Las transacciones son llevadas a cabo en la aplicación legada sin perturbar el estado de la aplicación. A diferencia del enfoque anterior, el enfoque basado en transacciones ofrece la posibilidad de servicio web de "dos sentidos", debido a que permite al sistema legado exponer sus funcionalidades y consumir otros servicios. Depende si el manejo de transacciones está encapsulado en procedimientos almacenados en la gestión de datos, o si se conforman en parte de la lógica de aplicación separada, este enfoque puede vincularse con los dos niveles inferiores de una arquitectura distribuida.

El enfoque basado en datos. Este enfoque es usado cuando se desea exponer datos legados. Se requiere una envoltura que estandarice la conectividad con las bases de datos y otras fuentes de datos. Para esto, el uso de XML en este enfoque permite la integración de varios sistemas de gestión de datos, ya que se provee un poderoso medio de intercambio de datos. Ofrece un servicio en "un sentido" también.

Herramientas en JavaEE y .Net

Existen dos plataformas de desarrollo bien conocidas para Servicios Web: JavaEE y .Net. Estas proveen las herramientas necesarias para construir servicios web en una manera *top-down*, proveyendo la descripción de servicios web a través de WSDL, el transporte de datos mediante SOAP y el registro y búsqueda de servicio por medio de UDDI. También proveen herramientas apropiadas que permiten la *envoltura (wrapping)* de aplicaciones legadas en servicios web; es decir, permite la aplicación del método *bottom-up* [45]. El Cuad. 7.1 ilustra las herramientas usadas por ambas tecnologías de desarrollo para implementar los diferentes enfoques de envolturas.

Java Server Pages (JSP) y *Active Server Pages (ASP)* son dos ejemplos de la aplicación del enfoque basado en sessiones para construir envolturas. Una página ASP o JSP puede ser

construida sobre el nivel de presentación de un sistema legado altamente acoplado para exponer su funcionalidad como servicio web sin cambiar su lógica de negocios. JDeveloper para JavaEE y .Net Framework son herramientas que posibilitan la construcción de envolturas basadas en transacciones. Por su parte, ambas plataformas poseen librerías de conectividad a bases de datos para realizar envolturas a nivel de datos.

Método para el diseño de envolturas

Se han propuesto varios métodos para realizar la aplicación de envolturas de servicios web a aplicaciones legadas. A modo de ejemplo, y por ser una de las propuestas que mejor describen las problemáticas y actividades involucradas en la modernización de aplicaciones legadas mediante envolturas, se detalla la metodología especificada por Harry M. Sneed [438], siendo uno de los varios trabajos que dicho autor escribió sobre la temática.

Al iniciarse cualquier proceso de integración con reutilización, y más precisamente al iniciarse cualquier proceso de orientación a servicios, surgen diferentes fuentes legadas que pueden o deben ser incluidas como activos empresariales en tales esfuerzos. Algunas aplicaciones de este tipo están altamente ligadas a los ambientes en los cuales fueron desarrolladas, en particular aquellas en dónde su nivel de presentación fue desarrollado por GUIs específicas y no fueron debidamente desacopladas. Otras aplicaciones legadas pueden están estrechamente asociadas con un sistema particular de base de datos, desarrolladas con software denominado de acceso a datos; como por ejemplo, los desarrollados con los monitores TP livianos (Véase Sec. 3.3.2). En la medida que la misma base de datos puede ser utilizada por otras implementaciones de presentación y lógica de negocios (como por ejemplo una aplicación web) las bases de datos pueden ser reutilizadas. Otra parte de las aplicaciones legadas encierra la lógica de aplicación o de negocios, la cual debe ser extraída del código existente [77].

Localizar y recuperar dicho software *orientado-a-negocios* es comúnmente referenciado como *minería de código* o *reciclado de código*. De forma análoga al hecho de reciclar partes de una construcción en ruinas para ser usada en un nuevo edificio. La tecnología para realizar esto está disponible desde mediado de los 90' y es adecuadamente explicada en la literatura de reingeniería de sistemas [68]. Lo nuevo en este contexto, es poder reutilizar estos viejos bloques de códigos en la forma de servicios web en una arquitectura orientada a servicios.

Las ventajas de reutilizar el código propio, en lugar de otras fuentes de servicios web, es obvia. El usar servicios web "enlatados", de terceras partes, es económico; sin embargo, es raro que tales servicios se adecuen a los requerimientos exactos de una organización. En el mejor de los casos, pueden utilizarse para suplantar servicios propios con alto grado de generecidad. Además, por no pertenecer a la empresa usuaria, ésta es dependiente del proveedor para el mantenimiento y evolución del servicio web. Desarrollar el servicio web desde cero, o re-desarrollarlo a partir del comportamiento de un sistema legado, es una alternativa tentadora, especialmente para desarrolladores deseosos de experimentar la nueva tecnología, pero se cae en el error de subestimar el esfuerzo requerido para testear los nuevos servicios y posicionarlos como verdaderos recursos reutilizables con todas las exigencias que esto conlleva, explicadas en la Sec. 7.2 [500].

El costo de desarrollar servicios web con alta calidad es, para algunas empresas usuarias, simplemente demasiado alto. Es por eso, que cuando se decide afrontar tal costo, la construcción de los servicios web propios desde cero es un objetivo a largo plazo que se alcanzará a lo largo de varios ejercicios financieros; no es algo que pueda realizarse a corto término. En esta situación, las empresas se topan con la opción de modernizar sus procesos de negocios para adecuarse a los servicios web estándares disponibles, o reutilizar su software existente o legado el cual fue hecho desde un comienzo para adecuarse a su particular proceso de negocios.

Aplicaciones legadas candidatas a envolver a través de servicios web. Existen varios escenarios adecuados para reutilizar aplicaciones legadas en la forma de servicios web. Hay estudios que indican que el núcleo de las funcionalidades presentes en las administraciones públicas

están basadas en sus aplicaciones existentes. Es inútil reproducirlas en otra forma que no sea "envolverlas" y exponerlas como servicios web públicos y universales. El gobierno electrónico *(e-goverment)* es uno de los primeros candidatos para las técnicas de reciclado de software mediante servicios web. Los mismo se aplica a las funciones organizativas de la administración privada. Existe un cúmulo de tareas únicas específicas de cada compañía. Ejemplos típicos son los modos de pago, la aceptación de créditos, el cálculo de intereses y el manejo de clientes importantes. Las funciones y procesos de negocios convencionales conforman el núcleo organizativo y cuentan con soporte informático específico cuyas aplicaciones han sido desarrolladas y evolucionadas a lo largo de los años. Son partes esenciales de la operación de la empresa. El problema es que esta lógica de negocios hecha a medida no es fácilmente accesible. Esta funcionalidad debe resguardarse y extraerse del código legado para ser reutilizada. Generalmente deben ser identificadas metódicamente a través de relevamientos y varias técnicas de minería [24].

En la reutilización de aplicaciones existentes, la primera tarea es identificar aquellas candidatas a exponerse como servicios web. Las empresas que desean encarar un plan de reuso de este tipo, basándose en arquitecturas orientadas a servicios, deben hacer un relevamiento y análisis de las aplicaciones existentes y detallar sus operaciones esenciales. En este proceso, es necesario desglosar las operaciones complejas en operaciones elementales que pueden ser contenidas en unidades lógicas.

Lograr obtener operaciones de baja granularidad aumenta el potencial de reutilización de tales funciones, como se explicó en la Sec. 7.2.6. Si tales operaciones están encapsuladas en uno o varios módulos existentes son candidatas para ser servicios web. Como resultado, éstas pueden ser adaptadas en cualquier proceso de negocios.

El segundo paso es estimar el valor de estos componentes de software candidatos. Algunas ideas [69] sugieren un esquema de clasificación, el cual involucra la categorización de items, calculando para cada uno el valor a partir del cual se calcula un coeficiente. Los componentes de software existentes pueden categorizarse por plataforma de desarrollo, propósito, tipo y criticidad. El cálculo del valor de un item está basado en el análisis de costo de desarrollo, del costo de mantenimiento, el costo estimado de reemplazo y el costo administrativo anual asociado con el item. Los items reutilizables son ordenados según el valor de dicho índice. Esta lista establece cuales operaciones o aplicaciones legadas tiene el potencial más alto para convertirse en servicios web.

En el contexto de reutilización de funciones legadas, la clave para definir servicios web adecuados es la granularidad de servicios. Estos deben tener el nivel de granularidad en el cual cada servicio realiza una transformación o computación simple bien definida, insumiendo un limitado conjunto de parámetros para proveer un resultado singular. Al ser un servicio web, debe cumplir el principio *"sin estado" (stateless)*; es decir, que el servicio no requiere que se mantenga ninguna información de estado, entre invocaciones de las operaciones del servicio. La preservación de objetos persistentes, es responsabilidad del proceso de negocio subyacente. Un ejemplo claro de este principio se mostró al tratarse WS-BPEL (Sec. 6.4.1), como mecanismo para definir una composición de servicios web sin estado, pero asumiendo la responsabilidad de llevar el estado de la coordinación entre ellos.

Es cierto que el proceso de negocios es sobrecargado a medida que crecen las invocaciones; sin embargo, de esta forma no será necesario cambiar constantemente el servicio web. En este punto, es una decisión de diseño definir si la lógica de aplicación del proceso de negocios está distribuida en cada uno de los servicios web; o está contenida dentro del mismo proceso de negocios. WS-BPEL va en esa dirección. Todos los posibles cambios son hechos a nivel de proceso de negocios, alterando el orden de invocación de los servicios web participantes, agregando nueva invocaciones, o cambiando algunos parámetros de invocación.

La esencia del éxito de las arquitecturas de orientación a servicios es la flexibilidad. La arquitectura debe ser adaptable a los cambios en el ambiente de negocios con el mínimo de esfuerzo en el tiempo. Esto se logra solamente manteniendo a los servicios web subyacentes con

un nivel bajo de complejidad. La complejidad debería residir en los procesos de negocios en donde se puede gestionar y manejar en forma adecuada.

Creación de servicios web a partir de aplicaciones legadas. Existen tres pasos básicos requeridos para crear un servicio web a partir de aplicaciones o código legado:

- identificar y rescatar la aplicación o código legado,
- "envolver" la aplicación legada,
- publicar y hacer disponible la aplicación legada como servicio web.

Para ser posible rescatar código legado es necesario localizar la aplicación y determinar si es posible su reuso. No es problema analizar y evaluar un pequeño número de programas; sin embargo cuando se examinan cientos de programas diferentes, es necesario recibir asesoría de un experto o utilizar alguna herramienta automatizada de reingeniería.

Una de las claves para descubrir las funcionalidades o ***operaciones de negocios*** son los resultados que éstas producen. Al identificar las variables que se utilizan para retornar las funciones de procesamientos, es posible identificar a través de éstas las operaciones de negocios. Si la aplicación legada está estructurada en una forma tal que las funciones de negocios le fueron asignadas a un único bloque de código, como una función en C, un párrafo en COBOL, un procedimiento interno en PL/1, o una rutina de Natural, la tarea sería sencilla. Sin embargo, en la mayoría de los casos, una función de negocios está dispersa en varios bloques de código en diferentes módulos. Por otro lado, un bloque de código, puede contener y proveer varias funciones de negocios. Por ende existe una relación "varios-a-varios" entre los bloques de código y las operaciones de negocios.

Al hacer un análisis de flujo de datos basándose en los resultados finales, es posible realizar una traza de los resultados hacia las instrucciones que contribuyen a producirlo. Una vez que las instrucciones son identificadas, es posible identificar en qué unidades de códigos están. A partir de allí se pueden copiar del código original solamente aquellas unidades con las variables que se usan de referencia. Esta técnica es conocida como *"code stripping"*[2]; originariamente era una técnica para validar el flujo de programa para lograr una determinada salida; sin embargo, es bastante útil a la hora de extraer las operaciones de negocios fundamentales [439]. De igual forma, además de identificar las unidades de códigos que se corresponden con una operación de negocios, también se deben identificar los objetos de datos que son afectados como resultado de su procesamiento.

Después de identificar el código de una operación de negocios, el siguiente paso es extraer el código y reensamblarlo como un módulo separado con su propia interfaz. Esto se logra copiando las unidades de código involucradas en un marco de trabajo común, reemplazando todos los objetos de datos que éstos referencian, con invocaciones a una interfaz común de datos. En C las interfaces son parámetros de una tipo `structure`, en COBOL los objetos son items de primer nivel de la `Linkage section`, en PL/1 los objetos son basados en estructuras de datos con punteros como parámetros al procedimiento principal. El resultado final será, en todos los casos, una subrutina con una interfaz de llamada; los argumentos originales serán parámetros de entrada y las salidas originales serán parámetros de salida. En este sentido, el código que conforma la lógica de negocios será desacoplado del nivel de datos y se constituirá en un subprograma autocontenido. Este es un prerequisito para realizar la envoltura de la operación de negocios legada para exponerla como servicio web [436].

Un subproducto del proceso de reingeniería de código es la documentación de las operaciones de negocios existentes. Para cada dato resultante de un caso particular de uso, las condiciones, asignaciones, computaciones y operaciones de entrada-salida que lo producen están representadas

[2]Se preserva el nombre en inglés por no encontrar una adecuada traducción.

en la forma de un árbol de flujo de datos *(data flow tree)*. El resultado final establece el nodo raíz del árbol; los otros nodos son los argumentos y variables intermedias las cuales "fluyen" hacia el resultado final. Las ramas del árbol representen las transiciones de estados las cuales son desencadenadas por instrucciones condicionales (`if`, `case` y otras).

Al ser una operación de negocios una intersección de flujos de control y de datos, es necesario representar ambas perspectivas. Con la ayuda de estos diagramas, es posible decidir si una operación existente, implementada en un sistema legado, vale la pena de ser reutilizada como una función púbica en una arquitectura orientada a servicios. Esta decisión requiere compresión de la operación en cuestión de la misma forma que una noción de su valor económico/organizativo. Si una operación tiene un alto valor económico y un bajo valor de implementación, quizás sea mejor reescribir el programa como una entidad separada. Las operaciones con una implementación aceptable y un valor económico significativo son principales candidatas para reusarse mediante envolturas.

Una vez que las operaciones de negocios han sido localizadas, documentadas y consideradas adecuadas candidatas para reutilizarse, el siguiente paso es "envolverlas". El objetivo primordial del proceso de envoltura *(wrapping)* es exponer al componente extraído del código legado con una interfaz WSDL. Esta técnica es usada para transformar cada entrada en un método y cada parámetro en un elemento XML. Las estructuras de datos serán elementos complejos XML con uno o varios subelementos. Los métodos contendrán referencias de sus argumentos y resultados a las descripciones de elementos de datos. En ambos casos, métodos y parámetros serán construidos en base a un esquema XML.

Existen varias herramientas para automatizar el proceso de envoltura, uno de los ejemplos es SoftWrap [446], que automatiza transformaciones para los lenguajes PL/1, COBOL, y C/C++. Además de crear la interfaz WSDL, también dota al componente envuelto con dos módulos adicionales. Un módulo está orientado a reconocer o realizar el *parsing* de un mensaje entrante y extraer la información de éste; el valor extraído es asignado a los argumentos correspondientes del componente envuelto.

El segundo módulo hace la operación complementaria, a partir de los argumentos de salida, crea un mensaje con elementos XML. De esta forma la operación legada puede reutilizarse como servicio web sin cambiar su código. Los dos módulos generados actúan como puente entre la interfaz WSDL y la interfaz de llamada del código legado.

En PL/1 estos dos módulos o subrutinas son implementados como procedimientos externos, en COBOL y C/C++ como clases separadas. El propósito de estos módulos es eliminar la manipulación arbitraria del código legado, debido a que dicha intervención es costosa y propensa a errores. Para lograr un procedimiento efectivo de envoltura, éste debe ser automatizado; hecho que fue ampliamente reconocido por los proveedores principales de software, los cuales ofrecieron soluciones para realizar envolturas de programas completos y bases de datos [210].

El tercer y último paso en la creación de servicios web a partir de código legado es vincular los servicios web al proceso de negocios subyacente. Esto es hecho por medio de un componente *proxy*. El proceso de negocio invoca en realidad a un proxy el cual está disponible en el mismo espacio direccionable que la definición del proceso [23]. Al igual que CORBA (véase Sec. 3.4.2) el proxy o stub controla los parámetros y genera la interfaz WSDL que es despachada por algún servicios de mensajes como el servidor de aplicaciones MQ-Series [232].

En el servidor de aplicaciones hay un planificador, el cual recibe el mensaje entrante, determina a cuál servicio web corresponde y reenvía los contenidos WSDL a ese servicio en particular, en este caso al código legado envuelto. La envoltura del código reconoce los datos de la entrada XML y mueve los valores a la dirección apropiada en el componente reusado.

Una vez que el componente envuelto ha sido ejecutado, su resultado es transformado por la envoltura en una estructura XML; la cual retorna al planificador para que se retransmita a la aplicación consumidora del servicio web. De esta forma, el proceso de negocios puede ser ejecutado a partir de cualquier cliente y aún está disponible para acceder a las funciones legadas

en el servidor de aplicaciones original [437].

7.4. Reutilización de Ontologías de Web Semántica

Otra de las facetas importantes a destacar respecto al empleo de Servicios Web con fines de reuso es el compartir información estructurada en ontologías mediante tecnologías de Web Semántica. Muchos son los problemas que enfrentan las ideas de compartir y reutilizar conocimiento. Actualmente el compartir conocimiento entre entidades es logrado en una forma bastante ad-hoc, careciendo de un adecuado entendimiento del significado de los datos. Por el *"compartir de conocimiento"* se entiende la transferencia o *reutilización* de conocimiento o saberes de una persona a otra, o de una organización a otra, de un grupo a otro grupo, de una persona a una organización, etc. [126].

Cuando el emisor y receptor pueden ser entidades arbitrarias que pueden o no tener el mismo lenguaje, la misma terminología o el mismo contexto de referencia, se deben extremar los cuidados en el mensaje que se envía de una entidad a otra. Un mensaje tiene un emisor y receptor, y es un documento que contiene datos estructurados que son la información que transporta. La información debe ser estructurada en una forma tal que el receptor debe ser capaz de entenderla y, de hecho, capturar el conocimiento relacionado con la información, aún si el emisor utiliza un lenguaje diferente y/o una tecnología diferente. Idealmente, el mensaje es estructurado en una forma tal que una computadora sea capaz de "entender" el documento. Con *"entender"* se hace referencia a la habilidad de la computadora de poder procesar los datos en el documento sobre la base de conocimiento hecha explícita a través del uso de un lenguaje lógico.

El conocimiento explícito consiste en reglas relacionadas a diferentes ítems de conocimiento, de forma tal que la computadora pueda deducir nuevos hechos a partir de los datos en el documento y el conocimiento explícito de antecedentes o de fondo *(background knowledge)*. La falta de estándares y semánticas formales en la Web actual y en los sistemas de Integración de Aplicaciones Empresariales impide que sea posible el compartir y reuso de información entre individuos y organizaciones.

7.4.1. Ontologías y Web Semántica

De acuerdo con el Diccionario de la Real Academia Española, ***ontología*** significa: "Parte de la metafísica que trata del ser en general y de sus propiedades trascendentales" [404]. Es una definición proveniente de la Filosofía. Sin embargo, este término fue rápidamente adoptado por la Inteligencia Artificial en campos de investigación como ingeniería del conocimiento, procesamiento del lenguaje natural, y representación del conocimiento. Tiempo más tarde, en la década de 1990, también fue usado en campos como la integración de información inteligente, recuperación de información desde Internet, y gestión del conocimiento [456].

En Inteligencia Artificial el término *"ontología"* hace referencia a la descripción formal de una porción del mundo o de conocimiento en un programa. Las ontologías fueron desarrolladas para facilitar el compartir y reuso de conocimiento [158]. Una importante definición fue introducida por Gruber [195] que dice: "una ontología es la especificación formal explícita de una conceptualización compartida". Dicha definición fue utilizada por varios autores en trabajos subsiguientes [158, 137, 293, 456, 287].

Una ***conceptualización*** es una vista simplificada y abstracta de la realidad que se desea representar en base a algún objetivo determinado. La ontología es una **especificación** porque representa la conceptualización en una forma concreta. Es **explícita** porque todos los conceptos y restricciones sobre éstos están explícitamente definidos. Por **formal** se entiende que una ontología debería ser comprensible (procesable) por una máquina. El hecho de que sea **compartida** indica que la ontología captura conocimiento consensuado. Las ontologías a través de semánticas formales, basadas en la realidad y con términos consensuados permiten entretejer y relacionar la

comprensión de personas y de máquinas. Además están orientadas a compartir conocimiento y por ende son un artefacto informático con altas posibilidades de reuso [158, 126].

Una computadora puede procesar datos de una ontología, cada dato puede estar vinculado al glosario ontológico, y mediante el conocimiento encapsulado en la ontología, deducir hechos sobre los datos originales. Una computadora puede, por ejemplo, deducir del dato que *"Pedro es un deportista"*, el hecho de que *"Pedro es una persona"*, dado que la ontología establece que *"todo deportista es una persona"*. Si más adelante, se establece en la ontología que *"toda persona tiene corazón"*, se deducirá entonces que *"Pedro tiene corazón"*.

Las ontologías se presentan en diferentes tipos y formas; desde aquellas que solamente representan un léxico con pocas relaciones entre elementos, hasta aquellas mucho más expresivas que intentan capturar todo aspecto relevante de un dominio dando amplio soporte al establecimiento de axiomas. La *expresividad (expressiveness)* de la ontología está limitada por el lenguaje ontológico, el cual es usado para especificarla (construirla). Muchos lenguajes de ontologías han sido desarrollados, abarcando todo el espectro de nivel de expresividad. A continuación se presentarán los lenguajes "clásicos" ontológicos y los actuales lenguajes ontológicos orientados a la Web.

La posibilidad de compartir conocimiento en ontologías no sólo se logra a través de semánticas formales, sino que también es necesaria una ingeniería consensuada de ontologías. Idealmente, las ontologías son vocabularios formales que se comparten entre un grupo de individuos interesados en un dominio específico. La visión de ontología compartida solamente puede ser lograda cuando se utiliza una metodología rigurosa que garantice que el proceso de ingeniería sea colaborativo.

Es importante destacar la complejidad a la que se llega en la ingeniería de ontologías cuando se toman en consideración grandes dominios y, en especial, cuándo se opta por un lenguaje más expresivo para describir la ontología. Existen varias metodologías de ingeniería de metodologías, las cuales intentan facilitar la ingeniería de las ontologías correctas y consensuadas. De la misma forma, existen varios principios de diseño, los cuales pueden ser usados para evaluar la calidad ontológica.

La ontologías fueron adoptadas por varias ramas de las Ciencias de la Computación como ser representación del conocimiento, recuperación de información, diseño de base de datos, etc [200]. En la actualidad la comunidad de investigadores ha prestado especial atención al uso de ontologías en la ***Web Semántica*** *(Semantic Web)*; la cual puede definirse como la gestión de conocimiento a escala global.

Las ontologías buscan entretejer el entendimiento de los símbolos que tienen personas (humanos) y máquinas (computadoras). Estos símbolos son denominados *términos*, los cuales pueden ser interpretados por personas o máquinas. Se entiende por ***término*** *(term)* a una palabra utilizada para identificar específicamente a un concepto. El significado de los términos para las personas es representado por el término en sí, el cual es generalmente una palabra en algún idioma, y por las relaciones entre términos que son comprensibles por personas. Un ejemplo típico de una relación comprensible por personas es la relación *es-un (is-a)*, canónica en el modelamiento de datos y de objetos. La relación establece el hecho que un concepto (el *super*concepto) es más general que otro (el *sub*concepto). Como ejemplo, se podría pensar en el concepto de `estudiante`, el cual es más general que `estudiante de maestría`. En la Fig. 7.1 se presenta dicho ejemplo y se muestra una jerárquia del uso de la relación *es-un*, que comunmente se conoce como ***taxonomía*** *(taxonomy)*. En una taxonomía los términos más generales están situados sobre los más específicos. Usando los pocos términos y relaciones mostrados en la Fig. 7.1, es posible sacar varias conclusiones; por ejemplo, se puede apreciar que un `estudiante de doctorado` (subconcepto) es un `estudiante` y un `investigador` (superconceptos). Estas conclusiones pueden obtenerse por personas y máquinas, debido a la semántica y a la naturaleza formal de la relación, respectivamente.

Los conceptos o términos utilizados en el ejemplo representan en realidad un conjunto de elementos. Es decir, el concepto `estudiante de maestría` representa a todos los estudiantes de maestría existentes (en el dominio de referencia). Uno de los estudiantes de doctorado en

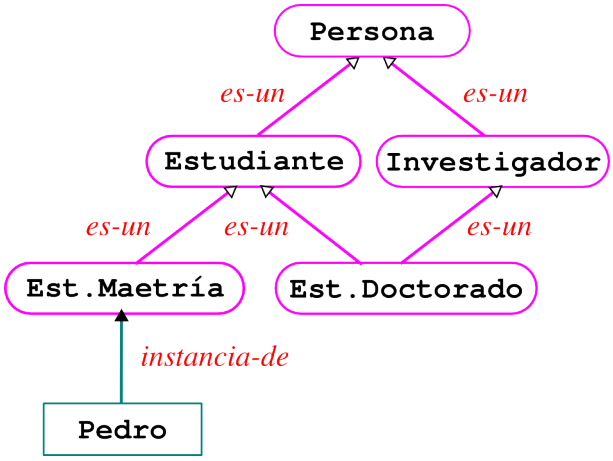

Figura 7.1: Ejemplo de una taxonomía.

particular es `Pedro`, que está modelado en la Fig. 7.1 con un rectángulo. Allí se establece una relación *es-instancia-de (instance-of)* con el concepto `estudiante de maestría`. La semántica de esta relación indica que un determinado objeto es capturado o concuerda con el concepto en cuestión. Dicho de otra forma; el objeto es uno de los elementos del conjunto que este concepto representa. En el caso, `Pedro` es instancia del concepto `estudiante de maestría`, y por las otras relaciones `es-un`, también es instancia de `estudiante` y de `persona`; pero no de `investigador`. Estas relaciones, que son tan fáciles de entender por personas, deben definirse formalmente y sin ambigüedades. De esta forma, una máquina puede razonar con los mismos términos y relaciones, y llegar a la misma conclusión que una persona. Esto se logra por especificar formalmente estas relaciones en una codificación comprensible (computable) por máquinas.

A estas conclusiones, las personas pueden llegar utilizando su sentido común, más precisamente utilizando la implicación lógica. En tanto que una máquina puede arribar a la misma inferencia, si tiene un método tal que pueda obtener las mismas implicaciones a partir de un proceso de derivación utilizando las relaciones descriptas formalmente. Para lograr esto, es necesario que el *proceso de derivación* o *mecanismo de inferencia* cumpla con la propiedad de ***sanidad*** o ***sensatez*** *(soundness)*. Esto significa que la máquina tiene un programa o procedimiento que "emula" al razonamiento o implicación hecha por personas; de forma tal que todas las inferencias que este programa "calcula" representan deducciones a las que una persona puede llegar [419]. En sentido estricto, las máquinas no logran efectivamente "entender"; sino que el entendimiento humano es codificado en una forma que la máquina puede procesarlo y generar las mismas conclusiones logradas por personas, a través del razonamiento lógico formal.

Clasificación de Ontologías

Existen diferentes tipos de ontologías, las cuales fueron construidas para diferentes tipos de aplicaciones, tienen diferentes alcances y niveles de detalle. Existen dos formas fundamentales de categorizar a las ontologías, por un lado si se tiene en cuenta su alcance se clasifican por su *generalidad*, y si se piensa en su nivel de detalle se clasifican según su *expresividad*.

Al ser una ontología una conceptualización compartida, exige que haya acuerdo entre expertos de dominio, usuarios y diseñadores respecto al conocimiento que debe especificarse para que la ontología sea (re)usable. No siempre se logra dicho acuerdo; sin embargo es favorable establecer diferentes niveles o capas de conocimiento representadas en diferentes ontologías basadas en su ***generalidad***. Esto flexibiliza la necesidad de que todos estén de acuerdo en una ontología; sólo las ontologías de más alto nivel de generalidad requieren tal esfuerzo. Existe una clasificación propuesta [126] a la luz de varios trabajos de investigación previos [293, 200, 456, 158]. Dicha clasificación categoriza los tipos de ontologías en:

- ***Ontologías genéricas*** *(generic ontology)* que capturan conocimiento general, independiente del dominio (como ser espacio, clima, empresa, estado, etc.).
- ***Ontologías de dominio*** *(domain ontology)* que capturan conocimiento de un dominio específico. (como ser de agroindustria, política impositiva)
- ***Ontologías de aplicación*** *(application ontology)* que capturan el conocimiento necesario para una aplicación específica (trazabilidad de carnes, impuestos a las ganancias).

Con el propósito de clasificar las ontologías de acuerdo con su ***expresividad***, se distingen diferentes niveles basados en espectro de ontologías [287]:

- ***Vocabulario controlado*** *(Controlled vocabulary)*: una lista de términos
- ***Tesauro*** *(Thesaurus)*: es un conjunto de términos y también se provee algunas relaciones de términos como ser la sinonimia.
- ***Taxonomía informal*** *(Informal taxonomy)*: existe una jerarquía explícita (son soportadas la generalización y especificación), pero no existe herencia estricta; es decir, una instancia de un subconcepto no siempre es instancia del superconcepto.
- ***Taxonomía formal*** *(Formal taxonomy)*: es igual que la anterior salvo que la herencia es estricta. Toda instancia de un subconcepto también es instancia del superconcepto.
- ***Marcos*** *(Frames)*: un *marco* o *clase* contiene un número determinado de propiedades y esas propiedades son heredadas por las subclases e instancias.
- ***Restricción de valores*** *(Value restrictions)*: los valores de las propiedades son restringidos, por ejemplo a un rango o tipo de dato.
- ***Restricciones lógicas generales*** *(General logic constraints)*: los valores que puede tomar una propiedad pueden ser restringidos por una fórmula numérica o expresión lógica utilizando valores de otras propiedades.
- ***Restricciones de lógica de primer orden*** *(First-order logic constraints)*: se pueden representar restricciones del Cálculo de Predicados como ser restricciones entre términos y relaciones como conjuntos disjuntos, relaciones inversas, relaciones parte-todo, etc. Algunos lenguajes que dan soporte a este tipo de ontologías son Ontolingua [254] y CycL [113].

Además de la distinción entre los niveles de expresividad, también se puede clasificar a las ontologías en de *peso liviano* y de *peso pesado* [107] siendo una distinción más simple teniendo en cuenta la expresividad. Las ***ontologías de peso liviano*** *(light-weight ontologies)* incluyen conceptos, propiedades que describen conceptos, relaciones entre conceptos y taxonomía de conceptos. Por otro lado, las ***ontologías de peso pesado*** *(hight-weight ontologies)* además incluyen axiomas y restricciones.

En la Fig. 7.2 se muestra el espectro de ontologías teniendo en cuenta las clasificaciones de expresividad y de "peso" ; también se muestran algunos ejemplos de ontologías a modo de ejemplificar las categorías. A continuación se detallan algunos ejemplo de de ontologías, referenciados en la Fig 7.2, ordenadas de menor a mayor expresividad:

- **Dublin Core** [531, 124] es un vocabulario controlado usado para especificar metadatos de documentos. La ontología Dublin Core contiene términos como "autor", "título", "editorial", etc.
- **WordNet** [399] es un tesauro de todas las palabras en Ingles con algunas relaciones básicas como sinonimia.

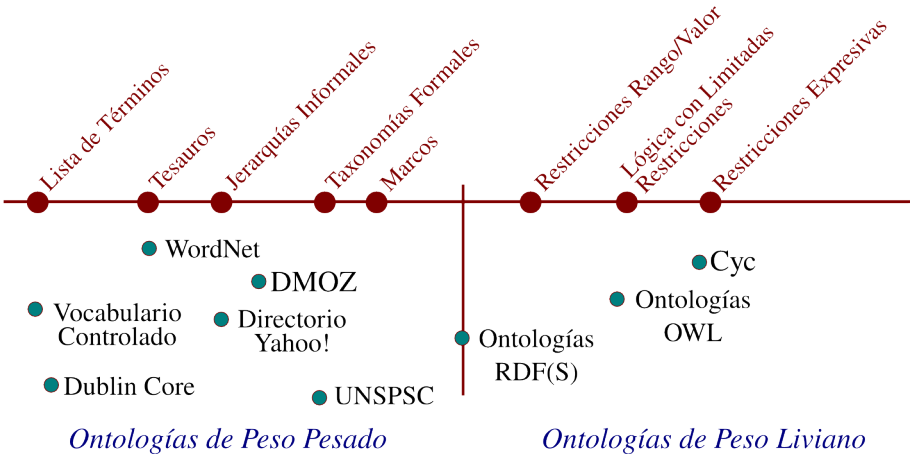

Figura 7.2: Espectro de ontologías segun su expresividad.

- **Directorio Yahoo!** [544] es una jerarquía informal usada para la clasificación de sitios web. Contiene varias categorías de nivel superior como "Negocios y Economía" y subcategorías de estas, como ser "Compras y Servicios".
- **DMOZ** [326] es un proyecto de directorio abierto y también es un jerarquía informal orientada a clasificar sitios web.
- **UNSPSC** [506] es una esquema de clasificación de productos, soportado por las Naciones Unidas teniendo en cuenta el punto de vista del proveedor de producto.
- **eCl@ss** [84], al igual que el anterior, es un esquema de clasificación de productos, pero orientado según el punto de vista del comprador. Es una iniciativa de la industria alemana.
- **RDF(S)** [308, 270, 81] no es realmente una ontología, sino un lenguaje de ontologías. Sin embargo las ontologías descriptas en RDF(S) conforman el límite de expresividad entre las ontologías livianas y pesadas.
- **OWL** [513, 130] también es un lenguaje de ontologías. Es posible describir en OWL ontologías de peso pesado con restricciones lógicas limitadas.
- **Cyc** [114] es una ontología creada con el lenguaje KIF (Knowledge Interchange Format) [187]. Es un lenguaje cuya expresividad soporta relaciones de lógica de primer orden.

La Web Semántica

En la Sec. 4.1.2 se explicó la génesis de la Web, dándole a Tim Berners-Lee el crédito de creador de la misma. Sin embargo, tiempo después Berners-Lee tuvo una visión sobre el futuro de la Web a la que denomino *Web Semántica* [50]. La idea fundamental de la ***Web Semántica*** es presentar la información de la Web en una forma que sea accesible, legible y "comprensible" por computadoras [19].

En la actualidad, una limitante importante es que la mayoría de la información presente en la Web está descripta en lenguaje natural, utilizando solamente metadatos para poder visualizarla; por ejemplo las etiquetas HTML. En sí la información no está orientada al procesamiento de computadoras, sino a la lecto-comprensión humana. Se han desarrollados tecnologías y herramientas sofisticadas, basadas en la inteligencia artificial y reconocimiento lingüista, para capturar el conocimiento actual descripto en la Web y transformarlo en información apta para el procesamiento computacional; sin embargo, estos enfoques fueron descartados por el nivel de ambigüedad encontrado en las fuentes de datos [158].

Es por eso que la adopción de la Web Semántica es la única opción válida para lograr que el conocimiento de la Web sea procesable por computadoras. Una forma de especificar dicho conocimiento es a través de ontologías. Una fuente de información en la Web puede contener una referencia a una ontología, o algún tipo de anotación o vínculo que referencia a una ontología. Esta ontología contiene la definición del concepto representado en la fuente de información de la web. Agentes inteligentes pueden, de esta forma, recopilar información de diferentes fuentes en la Web y combinar el conocimiento que contienen, gracias a las relaciones formales descriptas dentro y entre ontologías. Dos fuentes de datos pueden utilizar la misma ontología para describir su contenido, o utilizar dos ontologías diferentes. En el primer caso, resultaría fácil para el agente combinar los contenidos; en el segundo caso, se requiere que el agente realice un mapeo formal entre términos de ambas ontologías.

La Web Semántica es patrocinada por el W3C [519]. Es importante aclarar que la Web Semántica no intenta ser un medio de información global paralelo al existente hoy; más bien propone que gradualmente los contenidos de la Web actual migren hacia un formato más orientado al significado del conocimiento, más fiel a la visión original de Berners-Lee respecto a la Web.

Las áreas tradicionales para el estudio y aplicaciones de ontologías son la representación de conocimiento [79] y sistemas basados en conocimiento [456]. Sin embargo, gracias al surgimiento de la Web Semántica como campo de interés, la investigación y aplicación de ontologías se ha expandido a nuevas áreas que presentan un gran potencial. A continuación, se realiza una breve descripción de los efectos de las ontologías en diferentes áreas [158, 126].

Gestion de Conocimiento. La ***gestión de conocimiento*** tiene que ver con la adquisición, acceso y mantenimiento del conocimiento dentro de una organización. En los últimos tiempos se ha vuelto una actividad dentro de las empresas por considerar que el conocimiento interno como verdadero activo intelectual el cual conduce a mejorar la productividad e incrementar la competitividad. La gestión de conocimiento es particularmente importante en empresas que están geográficamente distribuidas. La mayoría de la documentación que conforma el conocimiento de una empresa está en formas no muy estructuradas, como ser archivos de texto, páginas HTML, archivo de audio, etc. Estas formas no son adecuadas para el tratamiento automatizado inteligente. Los mayores problemas se ven en la búsqueda y recuperación de información. En general, de los procesos de búsquedas en base a palabras claves y los resultados de las mismas son de acuerdo al número de ocurrencias de la palabra clave que se busca. Es una búsqueda orientada a la sintaxis de los términos, no a su significado. También se ve afectada la tarea de mantenimiento y actualización de información. Al no existir criterios estandarizados de indexación, agregar un nuevo documento puede generar inconsistencia en la terminología. Surge también como limitante, poder tener soporte multilingüe de la información. Otro problema es la integración de la información contenida en fuentes de información no documental, como por ejemplo las bases de datos; si bien en tal sentido, se han desarrollado técnicas de minería de datos, carecen del nivel adecuado de normalización para permitir su evolución y adaptación [19].

La Web Semántica permite lograr mejores sistemas de gestión de conocimiento, utilizando a las ontologías como elemento fundamental. Permite que el conocimiento esté organizado en espacios conceptuales de acuerdo a su significado, permite el empleo de herramientas automatizadas para soportar el mantenimiento, control de inconsistencia y la inferencia de nuevo conocimiento, consultas y resultados orientados a conceptos, multiplicidad de fuentes de información integradas, y personalización y parametrización de distintas vistas de la información recuperada e integrada.

Últimamente, varios proyectos han sido desarrollados para demostrar el uso de las ontologías en gestión de conocimiento. Por ejemplo el lenguaje SHOE *(Simple HTML Ontology Extensions)* [429] es una extensión de HTML, usado para agregar a una página Web etiquetas con información orientada a conceptos, mejorando las oportunidades de consultas orientadas a la semántica [211]. Por otro lado, Ontobroker es una arquitectura con tres elementos fundamentales: una interfaz de consultas, un motor de inferencia usado para derivar respuestas, y un agente de web o *webcrawler*

usado para recoger la información requerida de la web. El proyecto también provee un lenguaje de representación para la formulación de ontologías, con un subconjunto de elementos para realizar las consultas. Además se brinda un lenguaje de anotaciones para que los proveedores de páginas las enriquezcan con contenido ontológico [159, 161].

Un esfuerzo europeo importante de investigación fue el proyecto On-To-Knowledge [373], que se ejecutó desde 2000 a 2002. Su objetivo principal fue tratar la gestión de conocimiento en organizaciones grandes y descentralizadas usando la tecnología de Web Semántica. Como resultado de este proyecto, una gran cantidad de métodos y herramientas fueron desarrollados para facilitar la integración y mediación de la información. Uno de los productos más reconocidos fue *OIL (Ontology Inference Layer)* [372] un lenguaje para expresar ontologías que formó las bases de *DAML+OIL* [174]; el cual a su vez fue el predecesor de OWL (Web Ontology Language) [513], siendo este uno de los estándares más utilizados actualmente.

Integración de Aplicaciones Empresariales. Tanto la Web Semántica y las ontologías pueden cumplir un rol gravitante en los escenarios EAI. Como ya se explicaron en la Sec. 3.6, los entornos de EAI están compuestos por aplicaciones heterogéneras, las cuales almacenan datos independientemente, lo que genera redundancia y conduce a la inconsistencia y confusión respecto al origen de datos. Un ejemplo popular de estos ambientes son los *sistemas de gestión de relación con el cliente (CRM Systems)*. En una empresa, los datos de los clientes son generalmente administrados por varias aplicaciones o sistemas: el de marketing, el de cuenta corriente, el de ventas y facturación, el de despacho y entrega de producto, etc. Generalmente, estos sistemas no fueron desarrollados como un sistema único integral; en su lugar, cada uno fue desarrollado por separado, con tecnologías y equipos diferentes. Un sistema CRM trata de integrar estos sistemas para lograr consistencia entre los datos que maneja cada uno. Sin embargo preservar la integridad a través de este método es un problema importante de mantenimiento; además, se ven muy limitadas las posibilidades de extensión del sistema, agregando otras funciones, como por ejemplo la venta on-line.

Como se vió en la Sec. 7.3.2, una opción sería el re-desarrollo global de todas las aplicaciones, constituyendo un sistema global integral homogéneo. Sin embargo, esta opción tiene la contra de ser muy costosa en tiempo y dinero. En ocasiones, esto puede degenerar en la exigencia que la empresa cambie sus procesos de negocios para adaptarse al sistema informático, cuando lo admisible sería que el sistema informático se adapte a los procedimientos de la empresa. Los procesos de negocios necesitan un enfoque teleológico (orientado a un objetivo), extensible y reutilizable para la integración de aplicaciones. Las ontologías son inherentemente extensibles y reutilizables; estas son capaces de explicar los datos en las aplicaciones y por consiguiente posibilitar la integración dirigida por propósito [158].

Comercio Electrónico. Una forma especial de integración de aplicaciones empresariales es el comercio electrónico, en dónde se deben integrar aplicaciones en diferentes empresas o con entidades arbitrarias en la WEB. El comercio electrónico está dividido en dos áreas: B2C (Business-to-Consumer) y B2B (Business-to-Business) (Véase Secs. 4.2.1 y 4.4.2).

Las ontologías pueden se aplicadas a ambas áreas para incrementar la eficiencia y facilitar la cooperación. En el área B2C, las ontologías pueden ser utilizadas para implementar herramientas de comparación de precios *(shopbots)* para un producto vendido por varios proveedores. Para lograr esto, se pueden utilizar envolturas específicas para cada vendedor; por ejemplo, se pueden usar heurísticas para extraer información desde el sitio web del vendedor en una envoltura que examine las páginas, como una especie de rascado de pantalla (Véase Sec. 7.3.2). Esta es una solución ad-hoc que conlleva problemas de adaptación y mantenimiento; un cambio en la presentación de los productos en el sitio web del vendedor, incompatibiliza la envoltura. Si los vendedores utilizan ontologías, como otra forma de presentar sus productos, el agente comparador de precios usaría las ontologías de los vendedores o un mapeo entre las mismas para obtener

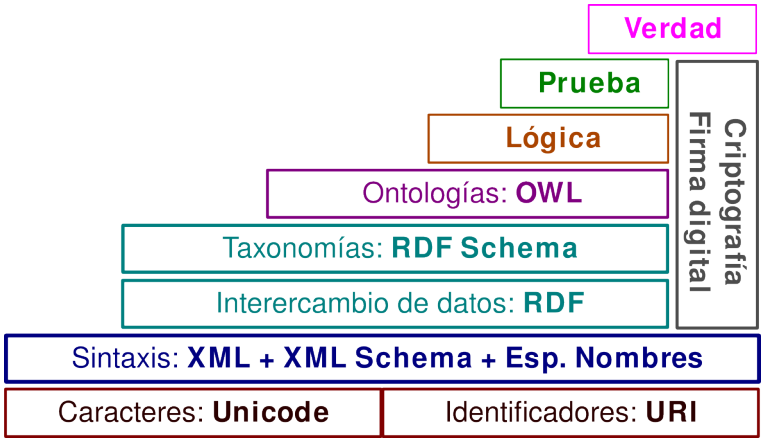

Figura 7.3: Torta de los Lenguajes de la Web Semántica. (Versión Simplificada)

información de una manera formal. De esta forma, el agente puede integrar automáticamente la información de los productos antes de mostrarla al consumidor.

Por otro lado, en el campo de las integraciones B2B, se ha realizado un enorme trabajo para estandarizar los formatos de representación de los mensajes eletrónicos entre comercios, como son el ejemplo de los estándares EDIFACT y XML (Véase Sec. 4.4.4). Además de estandarizar el contenido de los mensajes electrónicos, también se han hecho esfuerzos para describir el contenido como ser RosettaNet [416]. Estas descripciones de contenidos pueden ser utilizadas para un mercado determinado, siendo únicas para ese mercado. Si se utilizan ontologías en esta área, es posible facilitar la traducciones entre diferentes representaciones de documentos, mediar entre diferentes ontologías de productos y servicios, y mediar entre diferentes ontologías generales de comercio electrónico [137].

7.4.2. Lenguajes de la Web Semántica

En la década de 1990 la investigación sobre ontologías en el terreno de Inteligencia Artificial estaba altamente estimulada la idea de la Web Semántica. La convicción general, sostenida por el W3C, es que los lenguajes de ontologías viables para la Web Semántica necesitaban ser desarrollados para ser compatibles con los estándares Web actuales. Las especificaciones estándares relacionadas a ontologías de Web Semántica, conforman capas sobre las tecnologías estándares basada en XML, al igual que las tecnologías de los Servicios Web.

Una versión muy conocida de las relaciones entre lenguajes estándares fue presentada por el mismo Tim Berners-Lee [52], director del W3C, denominada la *Torta de los Lenguajes de la Web Semántica (Semantic Web Language Layer Cake)* y graficada, en una versión simplificada, en la Fig. 7.3. Dicho esquema presenta los diferentes lenguajes o estándares que se usan en la Web Semántica, aumentando en la complejidad de su sintaxis y expresividad semántica a medida que se sube en la torta.

En las capas inferiores, que representan las especificaciones Unicode, URI y XML-Schema, consiste en los estándares existentes que proveen las bases sintácticas para los demás lenguajes de la Semántica Web. ***Unicode*** [504] provee el esquema de codificación de caracteres exigido por XML. El estándar de *Identificador Único de Recurso* o **URI** [49] (véase Sec. 4.1.2) provee el medio para identificar univocamente y direccionar los documentos y otros recursos de la Web. Todos los conceptos usados en los lenguajes superiores de la torta son especificados usando Unicode e identificados usando URIs.

En la Sec. 5.5 se hace un extenso tratado del lenguaje *XML* y sus lenguajes de definición *XMLSchema* y del mecanismo de *Espacio de Nombres*. A continuación se describirá el estándar que conforma la tercera capa de la torta *RDF* y *RDF Schema*. Luego se detalla *OWL* como

tecnología para la cuarta capa relacionada a los lenguajes ontológicos.

RDF

El ***Marco de Trabajo para la Descripción de Recursos*** o ***RDF*** *(Resource Description Framework)* [270, 308] es el primer lenguaje desarrollado específicamente para la Web Semántica. RDF fue pensado como lenguaje para agregar metadatos "comprensible" por computadoras a la información presente en la Web. RDF se basa en XML para realizar la serialización de sus descripciones. También se cuenta con el lenguaje ***RDF Schema*** que extiende a RDF para permitir cierto modelado de ontologías basadas en marcos. Para esto cuenta con primitivas para establecer clases, propiedades e instancias. También permite introducir relaciones del tipo *instancia-de* y *subclase-de* [126, 162]

RDF(S), como se denomina a la combinación de RDF y RDF Schema, no es muy expresivo. En si, las ontologías descriptas con éste están en el límite de las ontologías de peso pesado y liviano (Véase Fig. 7.2). Permite la representación de taxonomías de conceptos y relaciones binarias. Se han creado algunas herramientas para este lenguaje, en especial para la validación de restricciones [107].

Existen elementos básicos en que se deben considerar para entender RDF [19, 546]:

- ***Recurso*** *(Resource)*: es el sujeto de las expresiones de RDF. Es decir, RDF especifica metadatos sobre un determinado ***recurso***. Un recurso es identificado por un URI (único y universal), y ese URI es usado como *nombre* del recurso.

- ***Propiedad*** *(Property)*: es un recurso que tiene un nombre (URI) y que puede ser utilizado como propiedad. Es decir, puede ser utilizado para describir algunos aspectos, características, atributos, o relaciones específicas para un determinado recurso.

- ***Sentencia*** *(Statement)*: una sentencia es usada para describir propiedades de recursos. Una sentencia establece una relación entre un recurso que cumple el rol de *Sujeto*, una propiedad que es el *Predicado* y un valor que es el *Objeto* de la sentencia. El valor de la propiedad puede ser un literal o un recurso. Por ende, una sentencia RDF indica que un recurso (el *sujeto*) es vinculado con otro recurso (el *objeto*) mediante una relación o propiedad (el *predicado*).

En otras palabras, una sentencia RDF puede ser expresada como una tupla *(Sujeto, Propiedad, Objeto)*, y ese orden no debe ser alterado. Una sentencia (o todas las sentencias) de un documento RDF pueden expresarse como un grafo dirigido, en donde el sujeto (recurso) y el objeto (valor o recurso) son nodos y la propiedad es un arco dirigido del primero al segundo. Eventualmente el objeto de una sentencia puede ser el sujeto de otra. Hay que destacar que todas las relaciones en RDF son binarias.

Sintaxis básica. Por ejemplo, se puede expresar conocimiento acerca de cámaras de fotos en una ontología *`OntoFoto`*. Se puede pensar en un determinado modelo de cámara fotográfica *`ACME-1026`*. Este modelo de cámara se lo debe considerar un término o concepto de `OntoFoto`; si se expresa este conocimiento en RDF, el término debe ser un recurso que se identifique unívocamente con un URI, el cual es `http://ontofoto.net/rdf#ACME-1026`. Se puede establecer que toda cámara fotográfica tiene un atributo `peso`, dicho concepto es identificado por el URI `http://ontofoto.net/camara#peso`. El modelo específico `ACME-1026` tiene un valor específico para ese atributo, por ejemplo `700 gr`. Aunque parezca obvio, se debe establecer explícitamente que el recurso `http://ontofoto.net/rdf#ACME-1026` es de hecho una cámara. En otras palabras, se debe indicar que el modelo `ACME-1026` es una instancia de una clase o tipo *Cámara*.

Este conocimiento puede ser expresado por una sentencia:

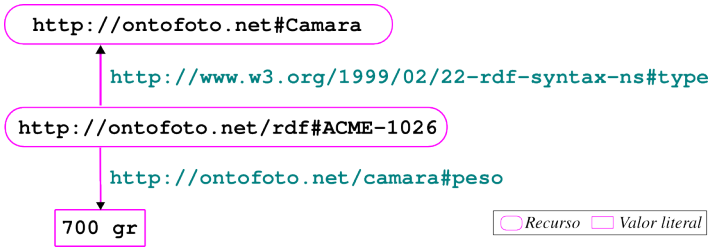

Figura 7.4: Un grafo RDF con uso de type.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns="http://ontofoto.net#">
          <rdf:Description rdf:about="http://ontofoto.net/rdf#ACME-1026">
             <rdf:type rdf:resource="http://ontofoto.net#Camara"/>
             <peso>700 gr</peso>
          </rdf:Description>
</rdf:RDF>
```

Código 7.1: Ejemplo de un documento RDF con relación de tipo.

`http://ontofoto.net/rdf#ACME-1026` es de tipo `http://ontofoto.net#camara` y tiene un **`http://ontofoto.net/camara#peso`** con un valor de **`700 gr`**.

En la Fig. 7.4, se puede apreciar el grafo que representa la sentencia anterior; y en el Cód. 7.1 el documento RDF que la describe. En esencia un documento RDF es un documento XML cuyo elemento raíz es `rdf:RDF` que, junto con los demás elementos del lenguaje RDF, está definido en el espacio de nombre `http://www.w3.org/1999/02/22-rdf-syntax-ns#` asignándole el prefijo *rdf*. También en el Cód. 7.1 de ejemplo se utiliza otro espacio de nombre por defecto (sin prefijo), el cual encierra todas las definiciones propias de la ontología `http://ontofoto.net#`.

En el Cód. 7.1, se utiliza el elemento `camara` para describir el concepto `ACME-1026`, vinculándolo a través del atributo `rdf:about` con el recurso `http://ontofoto.net/rdf#ACME-1026`. Se explicita la relación con la clase o tipo `Cámara` que se asocia con un recurso con el URI `http://ontofoto.net#camara`. Esta clase o tipo tiene varios atributos, uno de ellos es `peso`. Se debe tener en cuenta que la relación *es-de-tipo* o *instancia-de* que se quiere expresar, tambien es un concepto que RDF. Como tal, tiene un elemento **`rdf:type`** que lo representa y que pretenece al espacio de nombre **`http://www.w3.org/1999/02/22-rdf-syntax-ns`**.

Valores Literales y Recursos. En el ejemplo del Cód. 7.1 la propiedad `peso` tiene un valor literal `700 gr`. Sin embargo, la utilización de dicho valor literal es por la suposición que será correctamente interpretado. Sin embargo, para personas o programas que no reconocen el SiMeLa [1, 2] como sistema de medidas, puede ocurrir que el sufijo `gr` no sea correctamente entendido; además el valor expresado de esta forma no está normalizado.

Una solución a esta problemática sería expresar dicho valor y la unidad de medida de peso en forma separada en una estructura de dos componentes. De esta forma, la propiedad `peso` tiene a un recurso como objeto en lugar de un valor literal. En el Cód 7.2 se ve reflejado este cambio, y en la Fig. 7.5, se muestra las nuevas relaciones que conforma este nuevo recurso. En el Cód. se puede observar que el elemento `peso` quedo conformado como una estructura, dentro de la cual está presente el elemento `rdf:Description` encerrando a los dos componentes. Por un lado, el primer componente es un elemento `rdf:value`, que tiene un atributo `datatype` con un valor extraido de XMLSchema `decimal`. Esta definición dice que la magnitud del peso puede ser un valor decimal. El segundo componente es un elemento `uom:unit`, que encierra el concepto de *unidad de medida* definido en una ontología cuyo es-

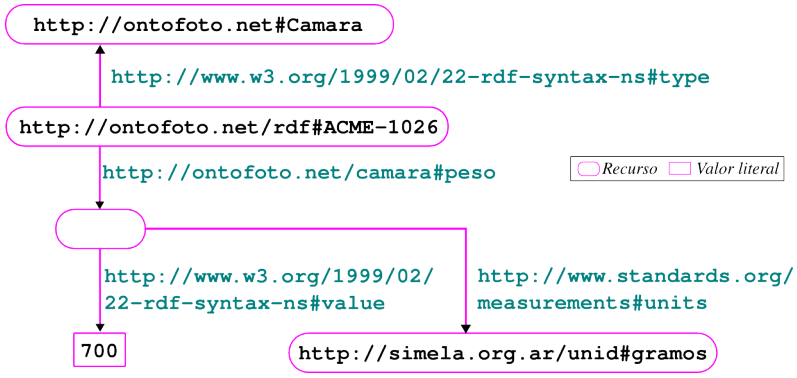

Figura 7.5: Un grafo RDF con un recurso como objeto de sentencia.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3     <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4                xmlns:uom="http://www.standards.org/measurements#"
5                xmlns="http://ontofoto.net#">
6       <rdf:Description rdf:about="http://ontofoto.net/rdf#ACME-1026">
7             <rdf:type rdf:resource="http://ontofoto.net#Camara"/>
8             <peso>
9               <rdf:Description>
10                  <rdf:value rdf:datatype="&xsd;decimal">700</rdf:value>
11                  <uom:units rdf:resource="http://simela.org.ar/unid#gramos"/>
12              </rdf:Description>
13            </peso>
14    </rdf:Description>
15  </rdf:RDF>
```

Código 7.2: Ejemplo de un documento RDF con relación de tipo.

pacio de nombres es `http://www.standards.org/measurements`; además tiene como atributo al recurso que hace referencia a un tipo de unidad en particular identificada por un URI, en este caso `http://simela.org.ar/unid#gramos`.

RDF Schema

No alcanza con las descripciones de RDF para representar conocimiento. Por ejemplo, en el Cód 7.2 se puede ver en la línea 7 que se dice `ACME-2610` es una instancia del tipo/clase `Camara`. Sin embargo, no se sabe dónde está definida dicha clase, y cómo es su estructura. Se desconoce la clase `Camara` tiene subclases o superclases; si tiene más propiedades que las presentes, etc. Estos interrogantes surgen por carecer de un *vocabulario* o *taxonomía* que establezca las relaciones faltantes entre clases o conceptos.

El lenguaje RDF solamente describe recursos en una forma estructurada procesable por computadora; permite establecer relaciones *sujeto-propiedad-objeto* binarias, de forma que se pueda realizar funciones básicas de razonamiento. Sin embargo RDF no puede establecer un vocabulario, solamente describe sentencias únicas, aisladas de cualquier grupo de conocimiento normalizado.

El lenguaje ***RDF Schema*** o ***RDFS*** es usado para crear vocabularios. Estos vocabularios brindan semántica a los documentos planos RDF más allá de la estructuración de conceptos. Esto brinda la posibilidad de conformar información distribuida más amigable y comprensible por máquinas. RDFS es una recomendación del W3C [81] y puede ser visto como un lenguaje de descripción de vocabularios. Estos vocabularios permiten describir clases, subclases y propiedades de los recursos RDF [107]. RDFS también asocia las propiedades con las clases que define. RDFS

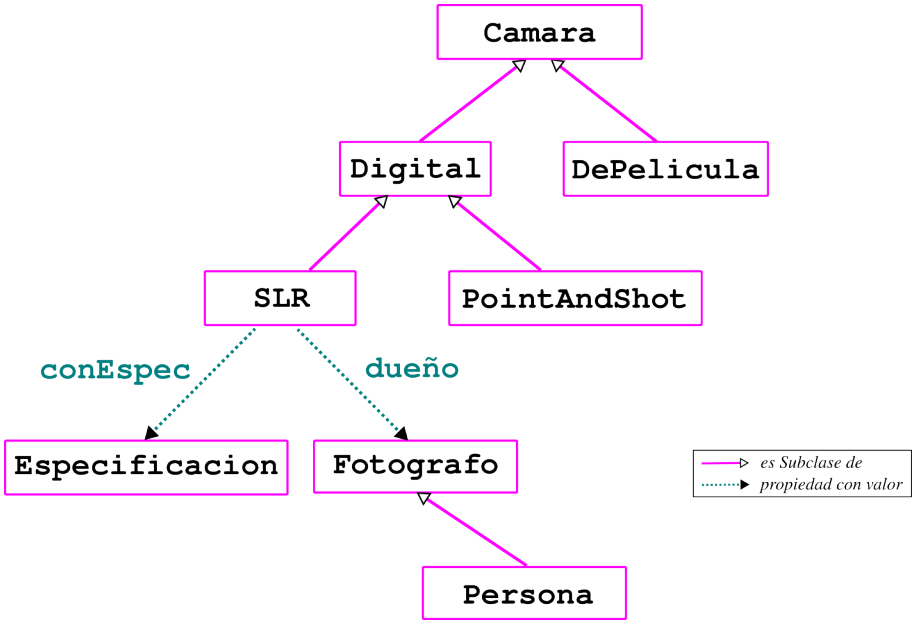

Figura 7.6: Un esquema de vocabulario relacionado con una ontología de cámaras fotográficas.

agrega semántica a los predicados y recursos RDF, define el significado de un determinado término a través de la especificación de sus propiedades y la determinación del dominio de valores válidos de estas propiedades. Una de las cualidades que tiene es que una especificación RDFS está escrita en RDF, usa el mismo modelo de datos; es decir, genera grafos conceptuales o tuplas *sujeto-propiedad-objeto*.

A los efectos de presentar los conceptos de RDFS, se supondrá un vocabulario relacionado al ejemplo de la ontología de cámaras fotográficas, presentado en la sección anterior. Dicho esquema de ejemplo que representa un vocabulario se muestra en la Fig. 7.6. El vocabulario representa los siguientes hechos: Se tiene un recurso (término) denominado *Camara*, que tiene dos subconceptos o subclases `Digital` y `DePelicula`. A su vez, `Digital` tiene dos subconceptos `SLR` y `PointAndShoot`. Por su parte `SLR` tiene una propiedad denominada `dueño`, cuyo valor posible es un recurso de tipo `Fotografo`, que a su vez es un subconcepto de `Persona`; y una propiedad `conEspec` cuyo valor posible es un recurso de tipo `Especificación`.

Para graficar la importancia de la definición semántica de un vocabulario, se presenta en el Cód. 7.3 una descripción RDF. Este documento RDF describe el concepto `ACME-2610` representado por el recurso `http://ontofoto.net/rdf/ACME2610.rdf#ACME-2610`. Se expresa que `ACME-2610` es de tipo `SLR` y tiene una propiedad `dueño` con un valor representado por el recurso `http://ontofoto.net/fotografo#Rosa Flores`. Si no se tuviera el vocabulario, no se podría decir más que eso. Sin embargo, se establece como espacio de nombre por defecto, a un vocabulario definido en `http://ontofoto.net/Camara.rdfs`. Se puede inferir por las relaciones taxonómicas expresadas en el vocabulario `Camara.rdfs` y del documento RDF del Cód. 7.3 que: `ACME-2610` es también del tipo `Digital` y del tipo `Camara`; que `http://ontofoto.net/fotografo#Rosa Flores` es un recurso que representa a un `Fotografo`, y por ende también a una `Persona`; y que `ACME-2610` no tiene definido un valor para la propiedad `conEspec`.

Definición de Clases. Para explicar las estructuras y elementos RDFS, teniendo en cuenta el vocabulario graficado en la Fig. 7.6, se propone una especificación RDFS del mismo en el Cód. 7.4. En este documento `Camara.rdfs` están definidas todas las clases del vocabulario. Se establece el prefijo `rdfs` para el espacio de nombres de RDFS `http://www.w3.org/2000/01/rdf-schema#`. Para definir cada clase, se utiliza el elemento **`rdfs:Class`**, el cual tiene un atributo **`rdf:ID`** que sirve para darle el nombre de la clase, indicando un recurso. En este caso, a diferencia de

```
<?xml version="1.0"?>
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
             xmlns="http://ontofoto.net/Camara.rdfs#">
      <rdf:Description rdf:about="http://ontofoto.net/rdf/ACME2610.rdf#ACME-2610">
            <rdf:type rdf:resource="http://ontofoto.net/Camara#SLR"/>
            <dueño rdf:resource="http://ontofoto.net/fotografo#Rosa Flores"/>
      </rdf:Description>
 </rdf:RDF>
```

Código 7.3: Ejemplo de un documento RDF con uso del vocabulario de la Fig. 7.6.

```
1  <?xml version="1.0"?>
2   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4         xml:base="http://ontofoto.net/Camara.rdfs">
5
6         <rdfs:Class rdf:ID="Camara"></rdfs:Class>
7         <rdfs:Class rdf:ID="Persona"></rdfs:Class>
8
9         <rdfs:Class rdf:ID="Digital">
10             <rdfs:subClassOf rdf:resource="#Camara"/>
11        </rdfs:Class>
12        <rdfs:Class rdf:ID="DePelicula">
13                <rdfs:subClassOf rdf:resource="#Camara"/>
14        </rdfs:Class>
15        <rdfs:Class rdf:ID="SLR">
16                <rdfs:subClassOf rdf:resource="#Digital"/>
17        </rdfs:Class>
18        <rdfs:Class rdf:ID="PointAndShoot">
19                <rdfs:subClassOf rdf:resource="#Digital"/>
20        </rdfs:Class>
21        <rdfs:Class rdf:ID="Fotografo">
22                <rdfs:subClassOf rdf:resource="#Persona"/>
23        </rdfs:Class>
24
25        <rdfs:Class rdf:ID="Especificacion"></rdfs:Class>
26   </rdf:RDF>
```

Código 7.4: Documento RDFS que describe las clases del vocabulario de la Fig. 7.6.

`rdf:Resourse`, `rdf:ID` establece el nombre del recurso a partir del espacio de nombres definido en `xml:base`. Por ejemplo, el recurso de la clase *`Digital`* (línea 9) es **`http://ontofoto.net/Camara.rdfs#Digital`**. Se definen también las clases *`Camara`* (línea 6), *`Persona`* (línea 7) y *`Especificacion`* (línea 25); en estos casos, no se indican que se deriven de otras clases; es por eso que en RDFS, cuando se omite la superclase en el elemento `rdfs:Class`, se asume que se es una subclase de la clase raiz **`rdfs:Resource`**. Para establecer una relación de *subconcepto*, se utiliza el elemento **`rdfs:subClassOf`** como subelemento del elemento `rdfs:Class`. Por ejemplo, al definir la clase *`DePelicula`* (líneas 12 a 14), se indica que es subclase de `Camara` (línea 13). Eventualmente una clase puede ser subclase de varias otras, basta con reiterar distintos elementos `rdfs:subClassOf`, variando su atributo de acuerdo al recurso que representa las diferentes superclases.

Definición de Propiedades. Para definir propiedades se utiliza el elemento **`rdfs:Property`**. En el Cód. 7.5 se muestran la codificación de propiedades basadas en el vocabulario de fotografía, pero estableciendo a modo de ejemplo otras propiedades no incluidas en Fig. 7.6. La primera propiedad que se define es `conEspec` (las líneas 8 a 10) que contiene dos subelementos. El primer subelemento **`rdfs:domain`** indica a qué clase pertenece, en este caso al recurso `#SLR`; el cual

```
1  <?xml version="1.0"?>
2   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4         xml:base="http://ontofoto.net/Camara.rdfs">
5
6         // Todas las definiciones de clases
7
8         <rdf:Property rdf:ID="conEspec">
9             <rdfs:domain rdf:resource="#SLR"/>
10            <rdfs:range rdf:resource="#Especificacion"/>
11        </rdf:Property>
12
13        <rdf:Property rdf:ID="dueño">
14            <rdfs:domain rdf:resource="#SLR"/>
15            <rdfs:domain rdf:resource="#PointAndShot"/>
16            <rdfs:range rdf:resource="#Fotografo"/>
17            <rdfs:range rdf:resource="#Periodista"/>
18        </rdf:Property>
19
20        <rdf:Property rdf:ID="modelo">
21            <rdfs:domain rdf:resource="#Especificacion"/>
22            <rdfs:range rdf:resource="http://www.w3c.org/2001/01/rdf-schema#Literal"/>
23        </rdf:Property>
24        <rdf:Property rdf:ID="modeloOficial">
25            <rdfs:subPropertyOf rdf:resource="#modelo"/>
26            <rdfs:label xml:lang="ES">Nombre de Modelo Oficial</rdfs:label>
27            <rdfs:comment xml:lang="ES"> Este es el nombre oficial de la camara.
28                 Algunos fabricantes pueden tener diferentes nombres para diferentes
29                 regiones o países.
30            </rdfs:comment>
31        </rdf:Property>
32   </rdf:RDF>
```

Código 7.5: Documento RDFS que describe propiedades de las clases

por comenzar con # es una definición local; es decir, se toma el espacio de nombres del propio documento `Camara.rdfs`. El segundo subelemento **`rdfs:range`** sirve para determinar qué tipo de recurso debe usarse como valor posible de esta propiedad, en este caso es un recurso de tipo `#Especificación`. La definición siguiente de **`dueño`** es similar a la anterior (líneas 13 a 18), sólo que en este caso la propiedad se usa en dos dominios (clases) y su valor puede pertenecer a dos tipos de recursos (clases) diferentes.

Un ejemplo interesante encierran la tercera (`modelo`) y la cuarta (`modeloOficina`) definición de propiedades. En `modelo` (líneas 20 a 23) se establece que se usa en el recurso `#Especificación` y que su valor posible es un literal, definido como tipo normalizado de RDFS. Por su parte, `modeloOficial` (líneas 24 a 31) se establece como subpropiedad de la anterior; es decir, como una especie de subclase de la propiedad `modelo`. En la línea 25 se utiliza el subelemento **`rdfs:subPropertyOf`** para indicar esta relación. Una subpropiedad hereda todo lo definido en su propiedad base, como por ejemplo el dominio y rango. Además de esto, también se especifica una etiqueta mediante el subelemento `rdfs:label`, el cual tiene como atributo el idioma y como valor una cadena de caracteres (literal) que representa la etiqueta. De la misma forma también se establece un comentario o anotación con el subelemento **`rdfs:comment`**.

Como se puede ver, las propiedades son declaradas en forma separada de las clases, incluso pueden declararse en otro documento RDFS. A diferencia de lo que se consideraría una clase en orientación a objetos, en RDFS las clases no son "dueñas" de las propiedades, definiendo su visibilidad y alcance. Es importante entender que una propiedad RDFS no tiene el mismo sentido que una variable miembro de una clase de objetos.

```
<?xml version="1.0"?>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xml:base="http://ontofoto.net/Camara.owl">

        <owl:Class rdf:ID="Camara"></owl:Class>
        <owl:Class rdf:ID="Persona"></owl:Class>

        <owl:Class rdf:ID="Digital">
            <rdfs:subClassOf rdf:resource="#Camara"/>
        </owl:Class>
        <owl:Class rdf:ID="DePelicula">
               <rdfs:subClassOf rdf:resource="#Camara"/>
        </owl:Class>
        <owl:Class rdf:ID="SLR">
               <rdfs:subClassOf rdf:resource="#Digital"/>
        </owl:Class>
        <owl:Class rdf:ID="PointAndShoot">
               <rdfs:subClassOf rdf:resource="#Digital"/>
        </owl:Class>
        <owl:Class rdf:ID="Fotografo">
               <rdfs:subClassOf rdf:resource="#Persona"/>
        </owl:Class>

        <owl:Class rdf:ID="Especificacion"></owl:Class>
  </rdf:RDF>
```

Código 7.6: Descripción en OWL de las clases el vocabulario de la Fig. 7.6.

OWL

El ***Lenguaje de Ontologías Web*** o ***OWL*** *(Web Ontology Language)* es una recomendación del W3C [513] y es uno de los lenguajes más utilizados actualmente para la creación y descripción de ontologías. OWL está construido sobre la base de RDF Schema. En una forma simple de verlo, se puede decir que OWL es RDFS con algunos constructores de mayor expresividad, lo que le permite ser capaz de expresar relaciones más ricas y complejas. Por ende todas las clases y propiedades provistas por RDFS pueden ser usadas en un documento OWL.

Definición de clases. En la Sec. 7.4.2 referida a RDFS, se explicaba que la clase raíz de cualquier clase definida en documento RDFS, era `rdfs:resourse`. Esta clase raíz tenía la URI `http://www.w3.org/2001/01/rdf-schema#resource`. En OWL la clase raíz es **`owl:Thing`**, cuya URI es **`http://www.w3.org/2002/07/owl#Thing`**. En la jerarquía de clases de OWL `owl:Thing` la clase raíz y la superclase de *rdfs:Resource*. (Nota: El prefijo **`owl`** representa al espacio de nombre de OWL **`http://www.w3.org/2002/07/owl#`**. Esta sustitución se preservará en el resto de las explicaciones)

Por otro lado, cuando se definen clases en un documento RDFS, se utilizaba el elemento *rdfs:Class* (Véase Cód. 7.4). En un documento OWL, el elemento empleado para tal fin es **`owl:Class`**, que en la jerarquía de clases de OWL es subclase de `rdfs:Class`. Si se tiene en cuenta el ejemplo de la ontología de cámaras presentado en la Fig. 7.6, un documento análogo al propuesto en Cód. 7.4 descripto en OWL sería el presentado en Cód. 7.6. Obsérvese que se preserva a `rdf:RDF` como elemento raíz, lo que significa que una descripción basada en OWL sigue siendo un documento RDF.

Hasta este punto de explicación, OWL no aporta más expresividad que la lograda por RDFS. A continuación se muestran algunas características de OWL que brindan mucha mayor expresividad.

```
<?xml version="1.0"?>
 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
       xmlns:owl="http://www.w3.org/2002/07/owl#"
       xml:base="http://ontofoto.net/Camara.owl">

  // Otras definiciones . . .
       <owl:Class rdf:ID="SLR">
              <rdfs:subClassOf rdf:resource="#Digital"/>
       </owl:Class>
       <owl:Class rdf:ID="SLRAltaRes">
              <rdfs:subClassOf rdf:resource="#SLR"/>
       </owl:Class>
  . . .
       <owl:Class rdf:ID="Fotografo">
              <rdfs:subClassOf rdf:resource="#Persona"/>
       </owl:Class>
       <owl:Class rdf:ID="Profesional">
              <rdfs:subClassOf rdf:resource="#Fotografo"/>
       </owl:Class>
       <owl:Class rdf:ID="Amateur">
              <rdfs:subClassOf rdf:resource="#Fotografo"/>
       </owl:Class>
  . . .
       <rdf:Property rdf:ID="dueño">
           <rdfs:domain rdf:resource="#SLR"/>
           <rdfs:domain rdf:resource="#PointAndShot"/>
           <rdfs:range rdf:resource="#Fotografo"/>
       </rdf:Property>
 </rdf:RDF>
```

Código 7.7: Definiciones OWL relevantes para la propiedad dueño.

Restricciones sobre valores. En las propiedades de la ontología de cámaras de la Fig. 7.6, se puede ver que existe una propiedad `dueño` que asocia a las clases `SLR` y `Fotografo`, para expresar que una cámara digital de tipo SLR tiene a un fotógrafo como dueño. Para explicar una de las extensiones en expresividad de OWL, se supondrá que cierto tipo de cámaras SLR de alta resolución, son tan caras y especiales que solamente las pueden poseer los fotógrafos profesionales. Para esto se debe derivar la clase `SLRAltaRes` de `SLR` para conceptualizar ese tipo especial de cámara; y también se distinguirán dos subclases de `Fotografo`: `Profesional` y `Amateur`.

En el Cód. 7.7 se expresan las definiciones de clases relevantes para el ejemplo, dejando la versión original de la propiedad. En este caso, por ser `SLRAltaRes` una subclase de `SLR` y `Amateur` una subclase de `Fotografo`, se puede lograr la siguiente relación: `Amateur - dueño - SLRAltaRes`; siendo algo que no se quiere concluir en realidad.

Para resolver este problema de especialización y especificar que sólo fotógrafos profesionales pueden tener una cámara SLR de alta resolución, OWL provee el elemento **`owl:allValuesFrom`** para solucionar el problema, como se muestra en el Cód 7.8, en el que sólo se detalla la nueva definición de la clase `SLRAltaRes`. En dicho código, se muestra la definición de la clase `SLRAltaRes` (líneas 4 a 12). Esta clase hereda de dos clases, de `SLR` (línea 5) y de otra clase anónima (líneas 6 a 11). En esta clase anónima, se presenta el elemento **`owl:Restriction`** (líneas 7 a 10) propio de esquema OWL que sirve para establecer una restricción sobre algún elemento; en este caso, sobre la propiedad `dueño`. La propiedad que se restringe se elije mediante el elemento **`owl:onProperty`**. Luego, mediante el elemento **`owl:allValuesFrom`** (línea 9) se indica que todos los valores posibles que puede tomar la propiedad deben ser de la clase `#Profesional`. De esta forma con la fusión de ambas superclases, `#SLR` y la clase anónima, la definición de la propiedad `dueño` de la primera se combina con la definición homónima de la segunda, restringiendo los valores de la

```
<?xml version="1.0"?>
 <rdf:RDF ... >
    . . .
      <owl:Class rdf:ID="SLRAltaRes">
          <rdfs:subClassOf rdf:resource="#SLR"/>
          <rdfs:subClassOf>
              <owl:Restriction>
                  <owl:onProperty rdf:resource="#dueño">
                  <owl:allValuesFrom rdf:resource="#Profesional">
              </owl:Restriction>
          </rdfs:subClassOf>
      </owl:Class>
   . . .
</rdf:RDF>
```

Código 7.8: Ejemplo del uso del elemento `owl:allValuesFrom`.

```
<?xml version="1.0"?>
 <rdf:RDF ... >
  . . .
     <owl:Class rdf:ID="SLRAltaRes">
        <rdfs:subClassOf rdf:resource="#SLR"/>
        <rdfs:subClassOf >
            <owl:Restriction >
                <owl:onProperty rdf:resource="#dueño">
                <owl:allValuesFrom rdf:resource="#Profesional">
                <owl:cardinality
                    rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
                    1
                </owl:cardinality>
            </owl:Restriction >
        </rdfs:subClassOf >
     </owl:Class>
  . . .
</rdf:RDF>
```

Código 7.9: Ejemplo del uso del elemento `owl:cardinality`.

propiedad para la clase `#SLRAltaRes`.

De la misma forma que se usó el elemento `owl:allValuesFrom`, se puede utilizar el elemento **`owl:someValuesFrom`**. En este caso, la semántica cambia de restrigir todos los valores, a restringir al menos uno. Es decir, se indica que al menos una de las instancias de `owl:SLRAltaRes`, debe tener a una instancia de `#Profesional` como valor de la propiedad `dueño`.

Restricciones de Cardinalidad. Otra forma de definir restricciones en propiedades es a través del uso de consideraciones de cardinalidad. Por ejemplo, si se exige sólo un dueño (fotógrafo profesional) para las cámaras digitales de tipo SLR de alta resolución, la definición de la clase `SLRAltaRes` será como se indica en el Cód. 7.9.

Además de restringir exactamente el número de ocurrencias de los valores de una propiedad, también se puede establecer un rango de cantidad de valores permitidos. Para esto se utiliza los elementos **`owl:mincardinality`** y **`owl:maxcardinality`**, como se muestra en el Cód. 7.10. En este código se establece que una cámara digital SLR de alta resolución puede tener entre uno y tres dueños fotógrafos profesionales.

Para completar la semántica de cardinalidad, se puede establecer que una cámara digital SLR de alta resolución puede tener *al menos* un dueno; si en la definición del Cód. 7.10 se preserva el elemento `owl:mincardinality` y se quita `owl:maxcardinality`. Y complementariamente, se

```
<?xml version="1.0"?>
 <rdf:RDF ... >
 . . .
       <owl:Class rdf:ID="SLRAltaRes">
           <rdfs:subClassOf rdf:resource="#SLR"/>
           <rdfs:subClassOf >
               <owl:Restriction >
                   <owl:cardinality rdf:resource="#dueño">
                   <owl:allValuesFrom rdf:resource="#Profesional">
                   <owl:mincardinality
                        rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
                     1
                   </owl:mincardinality>
                   <owl:maxcardinality
                        rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">
                     3
                   </owl:maxcardinality>
               </owl:Restriction >
           </rdfs:subClassOf >
       </owl:Class>
 . . .
</rdf:RDF>
```

Código 7.10: Ejemplo del uso del rango de cardinalidad en propiedades OWL.

puede pedir que exista *como máximo* tres dueños, si se deja `owl:maxcardinality` y se quita `owl:mincardinality`.

Enumeraciones de instancias de una clase. La posibilidad de enumerar los valores (recurso) posibles que pueden ser instancias de una clase, es una característica de expresividad agregada por OWL. En el Cód. 7.11, se muesta la definición de la clase `#SLRAltaRes`, en dónde se indica cuáles son las instancias que satisfacen con esa clase. Para esto, es utilizado el elemento **`owl:oneOf`** (líneas 6 a 13), en el cual se estipula una colección de todos los modelos de fotos (recursos) que pueden ser instancia de la clase.

Definición de propiedades. Además de los elementos de RDFS `rdfs:domain`, `rdfs:range` y `rdfs:subPropertyOf` (explicados en las Sec. 7.4.2), OWL incrementó el número de formas de caracterizar a una propiedad. El primer punto a considerar es que la definición de una pro-

```
<?xml version="1.0"?>
 <rdf:RDF ... >
  . . .
     <owl:Class rdf:ID="SLRAltaRes">
         <rdfs:subClassOf rdf:resource="#SLR"/>
         <rdfs:oneOf rdf:parseType='Collection'>
              <SLR rdf:about="http://acme.net/digital/#ACME-2610" />
              <SLR rdf:about="http://acme.net/digital/#ACME-3610" />
              <SLR rdf:about="http://clickflash.com/slrultra/#CLFL90c" />
              <SLR rdf:about="http://clickflash.com/slrultra/#FLA190e" />
              <SLR rdf:about="http://fuyi.com/digcam/#4588" />
              . . . // otras instancias
         </rdfs:oneOf >
     </owl:Class>
  . . .
 </rdf:RDF>
```

Código 7.11: Ejemplo del uso del elemento `owl:oneOf`.

```
<?xml version="1.0"?>
 <rdf:RDF ... >
    . . .
     <owl:ObjectProperty rdf:ID="dueño">
        <rdfs:domain rdf:resource="#SLR"/>
        <rdfs:range rdf:resource="#Fotografo"/>
     </owl:ObjectProperty>
    . . .
     <owl:DatatypeProperty rdf:ID="modelo">
        <rdfs:domain rdf:resource="#Especificacion"/>
        <rdfs:range rdf:resource="http://www.w3c.org/2001/01/rdf-schema#Literal"/>
     </owl:DatatypeProperty>
   . . .
 </rdf:RDF>
```

Código 7.12: Ejemplo del uso de los elemento `owl:ObjectProperty` y `owl:DatatypeProperty`.

```
<?xml version="1.0"?>
 <rdf:RDF ... >
  . . .
    <owl:ObjectProperty rdf:ID="amigo_de">
      <rdf:type rdf:resource="http://www.w3c.org/2002/07/owl#SymmetricProperty"/>
      <rdfs:domain rdf:resource="#Persona"/>
      <rdfs:range rdf:resource="#Persona"/>
    </owl:ObjectProperty>
  . . .
    <owl:ObjectProperty rdf:ID="mejorPrecio">
      <rdf:type rdf:resource="http://www.w3c.org/2002/07/owl#TransitiveProperty"/>
      <rdfs:domain rdf:resource="#Camara"/>
      <rdfs:range rdf:resource="#Camara"/>
    </owl:ObjectProperty>
  . . .
 </rdf:RDF>
```

Código 7.13: Ejemplo del uso del modificador `SymmetricProperty` en la definición de una propiedad OWL.

piedad OWL es algo diferente a las definiciones de RDFS. RDFS, al definir una propiedad, conecta un recurso con otro recurso, con un literal (valor no-tipado) o con valor tipado; utilizando el elemento **`rdf:Propierty`** para todos los casos. OWL usa diferentes tipos de constructores para implementar diferentes tipos de conexiones expresadas por una propiedad. Por un lado, **`owl:ObjectProperty`** es utilizado para conectar un recurso con un recurso. Por otro lado **`owl:DatatypeProperty`** es utilizado para conectar un recurso, con un valor literal o un valor predefinido en XML Schema.

Tanto **`owl:ObjectProperty`** como **`owl:DatatypeProperty`** son subclases de **`rdf:Property`**. En el Cód. 7.12 se muestra el uso de cada uno de los elementos de OWL destinados a la definición de propiedades.

Propiedades Simétricas. Una propiedad simétrica describe una situación en la cual un recurso `R1` está conectado al recurso `R2` por una propiedad `P`. Si la propiedad `P` fue definida como simétrica, entonces también se expresa que `R2` está conectado a `R1` por la misma propiedad.

Uno de los ejemplos más claros de este tipo de propiedades, es el que se representa como la relación `amigo_de`, expresada en el Cód. 7.13. En este caso, se establece el tipo de propiedad mediante el elemento **`rdf:type`**, el cual tiene un atributo **`rdf:resource`** con el valor **`http://www.w3.org/2002/07/owl#SymmetricProperty`**. Dicho valor es un recurso que indica la calidad de simetría de la propiedad.

```
<?xml version="1.0"?>
 <rdf:RDF ... >
  . . .
   <owl:ObjectProperty rdf:ID="mejorPrecio">
     <rdf:type rdf:resource="http://www.w3c.org/2002/07/owl#TransitiveProperty"/>
     <rdfs:domain rdf:resource="#Camara"/>
     <rdfs:range rdf:resource="#Camara"/>
   </owl:ObjectProperty>
  . . .
 </rdf:RDF>
```

Código 7.14: Ejemplo del uso del modificador `TransitiveProperty` en la definición de una propiedad OWL.

```
<?xml version="1.0"?>
 <rdf:RDF ... >
  . . .
   <owl:ObjectProperty rdf:ID="modelo">
     <rdf:type rdf:resource="http://www.w3c.org/2002/07/owl#FunctionalProperty"/>
     <rdfs:domain rdf:resource="#Specification"/>
     <rdfs:range rdf:resource="http://www.w3c.org/2001/01/rdf-schema#Literal"/>
   </owl:ObjectProperty>
  . . .
 </rdf:RDF>
```

Código 7.15: Ejemplo del uso del modificador `FunctionalProperty` en la definición de una propiedad OWL.

Propiedades Transitivas. Una propiedad transitiva describe una situación en la cual un recurso `R1` está conectado a otro `R2` por una propiedad `P`, y a su vez `R2` está conectado al recurso `R3` por la misma propiedad. Si la propiedad `P` fue definida como transitiva, entonces también se expresa que `R1` está conectado a `R3` por la misma propiedad.

Como ejemplo, existe una propiedad `mejorPrecio` en la ontología de cámaras, que se aplica entre dos recursos de clase `#Camara`. Según el ejemplo, en un documento RDF se establece que la cámara `ACME-2610` tiene `mejorPrecio` que otra `ACME-7010`. A su vez, en otra definición RDF se expresa que `ACME-7010` tiene mejor precio que una cámara `CLFL90c`. Se puede concluir, por carácter transitivo que `ACME-2610` tiene mejor precio que `CLFL90c`. En el Cód. 7.14 se muestra la definición en OWL del esquema de esta propiedad. También se utiliza el `rdf:type`, pero en este caso se lo vincula con el valor **`http://www.w3.org/2002/07/owl#TransitiveProperty`**.

Propiedades Funcionales. Una propiedad funcional describe una situación en la que, dada una instancia, existe un único valor que se corresponde con esa instancia, según dicha propiedad. Por ejemplo de este tipo de relación, se puede pensar en el registro de vehículos, en donde existe una propiedad que establece que para cada vehículo registrado existe un único número de chapa patente (de dominio). Otro ejemplo, tomando la ontología de cámaras, consiste en rever la propiedad `modelo`. Es lógico establecer que cada cámara en particular, tendrá una única descripción de modelo. El Cód. 7.15 muestra esta situación. En la definición de `rdf:type`, se utiliza **`http://www.w3.org/2002/07/owl#FunctionalProperty`** para indicar la calidad funcional de la propiedad.

Propiedades Inversas. Dos propiedades `P,S` son inversas si el recurso `R1` está conectado con el recurso `R2` por la propiedad `P`; y a su vez `R2` está conectado a `R1` por la propiedad `S` que tiene semántica inversa complementaria a `P`. Dentro del ejemplo del vocabulario de cámaras fotográficas se puede establecer que la propiedad `dueño`, que establece una relación entre una instancia de `#Fotografo` y una instancia `#SLR`, puede tener una relación inversa entre una cámara SLR y un fotógrafo, denominada `pertenencia_de`. Estas propiedades inversas son descriptas en el Cód.

```
1  <?xml version="1.0"?>
2   <rdf:RDF ... >
3    . . .
4     <owl:ObjectProperty rdf:ID="dueño">
5       <rdfs:domain rdf:resource="#SLR"/>
6       <rdfs:range rdf:resource="#Fotografo"/>
7     </owl:ObjectProperty>
8    . . .
9     <owl:ObjectProperty rdf:ID="pertenencia_de">
10      <owl:inverseOf rdf:resource="#dueño"/>
11      <rdfs:domain rdf:resource="#Fotografo"/>
12      <rdfs:range rdf:resource="#SLR"/>
13    </owl:ObjectProperty>
14   . . .
15  </rdf:RDF>
```

Código 7.16: Ejemplo del uso del elemento `inverse0f` en la definición de una propiedad OWL.

7.16. En la definición de `rdf:type` de la segunda propiedad, se utiliza el elemento **`owl:inverse0f`** con un atributo de tipo `rdf:resource` en el cual se establece el nombre de la propiedad que se invierte.

Correspondencia entre ontologías. Si se promueve la integración entre ontologías, es necesario que el lenguaje que las describe tenga capacidades para establecer equivalencias entre términos y relaciones de distintas ontologías. La Web Semántica es un ambiente especialmente orientado a la integración de información distribuida, es por eso que una de sus principales tecnologías como OWL permite mecanismos para realizar estas vinculaciones.

Una de las formas que presenta OWL para permitir la correspondencia entre términos y relaciones de distintas ontologías es mediante la utilización del elemento **`owl:equivalentClass`** en la definición de clases. Con este elemento OWL, se puede decir que un término (recurso) de una ontología externa, es equivalente al concepto que se define en dicha clase. Por ejemplo, si existe una ontología que abarca el mismo dominio que `ontofoto`, pero que es gerenciada por otra organización que utiliza términos en inglés. En las defincciones de `Ontofoto` se puede establecer la equivalencia de términos. En el Cód. 7.17 se puede observar la equivalencia del término `Digital` de la ontología local con el término `DigitalCam` de la ontología externa (línea 6); también se ve la equivalencia entre `SLR` y `SingleLensReflex` (línea 11).

Por otro lado, si se desea expresar que dos términos de la misma ontología o de ontologías diferentes son excluyentes uno de otro, se utiliza el elemento **`owl:disjointWith`** en la definición de una de las clases. Con esto se estable que una instancia de una clase nunca será también instancia de la otra en el mismo momento. Es decir, se expresa que el conjunto de instancias de una y otra clase son totalmente disjuntos. En el Cód. 7.17 se expresa que la clase `SLR` es disjunta con la clase `PointAndShot` de la ontología local; y también se indica que es disjunta con la clase `PAS` de la ontología externa (líneas 12 y 13 respectivamente).

Facetas de OWL

La expresividad de RDF(S) es bastante limitada. Viéndolo en una forma simple RDF Schema establece vocabularios normalizados para documentos RDF. RDF Schema es bastante simple, define una jerarquía de clases y una jerarquía de propiedades con restricciones en el dominio y rango de tales propiedades. Sin embargo, luego de liberar RDF Schema, el Grupo de Trabajo de Ontología Web del W3C (Web Ontology Working Group) [530] identificaron prontamente un número de casos de usos característicos en la construcción de ontologías que requerían mayor expresividad que la ofrecida por RDF(S). Por ejemplo [126]:

```
<?xml version="1.0"?>
 <rdf:RDF ... >
  . . .
    <owl:Class rdf:ID="Digital">
      <rdfs:subClassOf|- rdf:resource="#Camara"/>
      <owl:equivalentClass rdf:resource="http://otraOnto.com#DigitalCam"/>
    </owl:Class>

    <owl:Class rdf:ID="SLR">
      <rdfs:subClassOf rdf:resource="#Digital"/>
      <owl:equivalentClass rdf:resource="http://otraOnto.com#SingleLensReflex"/>
      <owl:disjointWith rdf:resource="#PointAndShot"/>
      <owl:disjointWith rdf:resource="http://otraOnto.com#PAS"/>
    <owl:Class>
  . . .
</rdf:RDF>
```

Código 7.17: Ejemplo del uso de los elementos `equivalentClass` y `disjointWith` en la definición de clases OWL.

- No existe forma de declarar clases equivalentes o disjuntas. La definición de clases equivalentes es una situación recurrente cuando se intenta integrar ontologías. En RDF(S) sólo es posible establecer la relación de subclase.
- RDF(S) no permite los conceptos de unión, intersección o complemento. Estos conceptos son bastante útiles cuando se desea construir nuevas clases sin usar herencia.
- RDF(S) no permite restricciones de cardinalidad sobre propiedades.
- RDF(S) no provee mecanismos para determinar el alcance de una propiedad; por ejemplo no se puede enumerar sus posibles instancias que pueden tomarse como valores.
- RDF(S) no permite definir características especiales de las propiedades, como ser simetría, transitividad, inversión, etc.

Habiéndose advertido la necesidad de expresividad, varios grupos en Estados Unidos y Europa emprendieron un esfuerzo conjunto para lograr un lenguaje de modelamiento de ontologías más poderoso. El resultado fue el lenguaje DAML+OIL [174]. En pocas palabras, este lenguaje fue la fusión de una propuesta estadounidense DAML-ONT [288] y una europea OIL [372].

El lenguaje DAML+OIL fue el punto de partida para el Grupo de Trabajo de Ontologías Web del W3C para desarrollar OWL. La principal motivación para el Grupo de Trabajo fue establecer a OWL como lenguaje estándar para la Web Semántica. Una cuestión a tener en cuenta en el diseño de un lenguaje ontológico es el balance que debe lograrse entre expresividad y eficiencia en el proceso de razonamiento. En general, mientras más expresivo sea el lenguaje más ineficiente serán las máquinas de razonamiento que lo soporten. Es por eso que el objetivo del diseño es lograr un lenguaje lo suficientemente expresivo para grandes ontologías y también lo suficientemente simple para soportar una eficiencia razonable. Desafortunadamente, en el caso de OWL, aunque sus constructores tienen alta expresividad, su implementación debe lidiar con complejidades computacionales, en ocasiones incontrolables. En búsqueda de balance entre expresividad y eficiencia, el Grupo de Trabajo responsable de OWL, generó tres diferentes modos de OWL, cada uno de los cuales respeta diferentes niveles de balance entre simpleza y performance. A continuación se explica cada una de estas facetas de OWL [130, 513].

OWL Full. Los ejemplos y explicaciones citados en este trabajo están bajo ***OWL Full***. Todos los elementos tratados están disponibles en este modo de OWL; por ende es el modo que demuestra mayor expresividad. También admite la combinación arbitraria de estos elementos

con constructores de RDF(S). La compatibilidad establece que todo documento RDF legal es un documento OWL Full. La desventaja principal de este modo de OWL es el costo del motor de razonamiento requerido para proveer soporte completo a su expresividad; hecho que está ligado con la eficiencia.

OWL DL. ***OWL DL*** significa *OWL Description Logic*. Es un sublenguaje de OWL Full que tiene restricciones sobre el modo en que deben utilizarse los constructores OWL y RDF(S). En especial se establecen las siguientes reglas:

- No se admiten combinaciones arbitrarias. Cada recurso puede ser solamente una clase, un tipo de dato, una propiedad de dato, una propiedad de objeto, una instancia o un valor dato. Es decir que el rol de un recurso es único.
- Restricciones en las propiedades funcionales e inversas. Estos dos tipos de propiedades son subclases de `rdf:Property`. Por ende, pueden conectar recurso con recurso, o recurso con valor. Sin embargo, en OWL DL, estas propiedades pueden usarse solamente con la propiedad de objeto `owl:ObjectProperty` y no con las de tipo de dato `DatatypeProperty`.
- Restricciones en las propiedades transitivas, ya que no se puede utilizar `owl:cardinality`.

Claramente, las limitaciones impuestas en expresividad, permite que un motor de razonamiento de OWL DL tenga una respuesta rápida. De hecho, el motor es más fácil de construir.

OWL Lite. ***OWL Lite*** es el más restrictivo en expresividad y es un subconjunto de OWL DL. Las restricciones más importantes son:

- No admite `owl:hasValue`, `owl:disjointWith`, `owl:unionOf`, `owl:complementOf`, ni `owl:oneOf`.
- El uso de cardinalidad está acotado. No se admite el uso de `owl:minCardinality` o `owl:maxCardinality`. Se puede utilizar `owl:cardinality`, pero haciendo que su valor sea binario: 0 o 1.
- `owl:equivalentClass` no puede ser utilizado, salvo para indicar conexión entre identificadores de clases.

La ventaja principal de OWL Lite es la eficiencia y el tamaño del motor de razonamiento. La gran desventaja es la pérdida del poder de expresividad.

7.4.3. Ingeniería Ontológica

Las ontologías proveen una teoría respecto a un determinado dominio utilizando un lenguaje expresivo para capturar dicho dominio. Una de las cualidades de las ontologías es que todo el conocimiento relevante debe hacerse explícito. Esto exige que se especifiquen muchas relaciones y conceptos; si esta información se deja implícita, el poder recuperarla e inferirla dependerá de la aplicación que use la ontologías. Esto le quita autonomía a la ontología ya que está estrechamente ligada a una determinada aplicación para que se exprese su conocimiento.

Es por eso que se deben utilizar lenguajes altamente expresivos, pero que puedan ser computables; es decir, tratables y decidibles. Si esa ontología está expresada en lenguajes estándares (computables), es posible compartirla e integrarla con otras ontologías. Incluso puede ser utilizada como recurso de información en sistemas o aplicaciones no previstas. Esto constituye un verdadero y atractivo escenario potencial de reuso; no será de capacidades y funcionalidades como los servicios web, sino de conocimiento.

Una ontología es un artefacto de software, al igual que un servicio web, y como tal debe ser diseñado y construido. Además de la complejidad inherente a la abstracción del conocimiento

a capturar en una ontología, la ingeniería ontológica debe hacer frente a calidades de diseño informático como ser el fácil mantenimiento, la extensibilidad, la integración, la portabilidad y la reutilización. Para comenzar se explicarán los principios de diseño de las ontologías, y se describirán a continuación algunas metodologías que involucran a la reutilización como actividad recomendada. También se puede ver en dichos métodos el soporte al principio de reusabilidad, para aplicarlo a la producción de nuevas ontologías.

Principios de diseño ontológico

El aspecto más importante de una metodología de ingeniería ontológica es el resultado que ésta produce; es decir, la calidad de la ontología como producto final. Con el propósito de establecer cuáles medidas de calidad son necesarias, se citarán algunos criterios de diseño que se deben tenerse en cuenta para considerar a una ontología "buena", apta para el compartir conocimiento y para el reuso. Algunos autores formularon ciertos principios de diseños de ontologías [196, 456, 508], los más importantes son:

- ***Claridad***: el significado estipulado de un término debe ser efectivamente comunicado; se debe definir los términos en función de condiciones necesarias y suficientes. Al respecto se puede decir que las ontologías están codificadas en algún lenguaje lógico. En este contexto, por condiciones necesarias (1) y suficientes (2), se entiende que (1) el término debe ser una consecuencia lógica de su definición y (2) la definición del término debe ser una consecuencia lógica del término. Es decir, un automóvil es un vehículo con motor, habitáculo, motor y cuatro ruedas; de igual forma, un objeto que se mueve, tiene habitáculo, motor y cuatro ruedas debe ser un automóvil.

- ***Coherencia*** *(consistencia)*: Las definiciones que conforman una ontología deben ser consistentes; no sólo lógicamente consistentes, sino que la parte informal de la ontología debe ser consistente con la parte formal. Esto significa que los conceptos, hechos y relaciones del mundo real deben corresponderse con los términos y sentencias expresadas en la lógica. La parte formal de la ontología puede ser verificada usando motores de inferencias. En términos de lógica, para que una teoría sea consistente debe existir al menos una interpretación que haga que la teoría sea verdad.

- ***Extensibilidad***: una ontología debe diseñarse para que pueda ser extensible, ya sea para agregar nuevos términos y relaciones, sin tener que revisar los términos y definiciones existentes. Esta calidad de diseño es una condición para que la ontología pueda evolucionar luego de su desarrollo. Los cambios, agregados o eliminaciones de componentes ontológicos deben causar el menor efecto posible dentro de la ontología, preservando su coherencia.

- ***Tendencia mínima a la codificación***: las opciones de representación no deben elegirse por conveniencia de notación o implementación. Las ontologías deberían ser independientes, dentro de lo posible, de las aplicaciones que las utilizan, del idioma de sus creadores, del ambiente en que se almacenan. Por ejemplo, las decisiones respecto a la forma en que los nombres deben ser escritos (mayúsculas/minúsculas), el formato de fecha y hora, etc. no deberían ser tomadas en la propia ontología.

- ***Mínimos compromisos ontológicos***: con el propósito que una ontología sea compartible y reutilizable lo más posible; se deberían hacer la mínima cantidad de afirmaciones o suposiciones respecto al mundo reservando siempre la cualidad de compartible. Dicho de otra forma, la ontología debe ser agnóstica, pero encerrar conocimiento reutilizable. Una menor cantidad de compromisos ontológicos, hace que la ontología sea mas extensible y reutilizable; es decir, al exigir estar de acuerdo con pocas afirmaciones/suposiciones, el

acuerdo de los usuarios de la ontología es más fácil. Sin embargo, al reducir los compromisos ontológicos, el dominio de la ontología se estrecha en algún sentido. Aquí se plantea conformar un balance entre usabilidad y reusabilidad [456, 126].

Reuso de Ontologías

Como se explicó, para lograr el máximo beneficio en uso de ontologías, éstas necesitan ser compartidas y reutilizadas. Las ontologías existentes pueden ser combinadas para lograr crear nuevas ontologías. Esto ha sido un objetivo en el diseño de ontologías por varios años [508, 158]. La reusabilidad es la principal razón para distinguir diferentes niveles de abstracción de ontologías (veáse Sec. 7.4.1). Una ontología de nivel más superior puede ser reutilizada en diferentes dominios. Si, por otro lado, conocimiento de mayor nivel de abstracción es registrado en una ontología específica de un dominio, este conocimiento no puede ser fácilmente reutilizado en otros dominios, debido a que la distinción entre el conocimiento genérico y específico de dominio no está explicitada. Es muy probable que estos conceptos de nivel superior en cuanto a la abstracción deban ser nuevamente definidos en otra ontología de dominio.

Una nueva ontología puede ser construida a partir de otras mediante el uso de métodos de *inclusión*, *restricción* y *refinamiento polimórfico* [456]. A continuación se detalla cada uno:

- ***Inclusión***: una nueva ontología surge a partir de la inclusión de una ontología existente en otra. El resultado es la unión de dos ontologías. Un problema a resolver en estos casos es el conflicto o "colisión" de nombres.

- ***Restricción***: sólo un subconjunto restringido de conceptos y relaciones de una ontología existente son usados para conformar otra. Este método es utilizado cuando hay interés en sólo una parte de una ontología; o cuando dos ontologías tienen porciones de conocimiento que se sobreponen.

- ***Refinamiento Polimórfico***: consiste en agregar relaciones entre conceptos de una ontología existente y conceptos de la nueva ontología. Por ejemplo, si se tiene una ontología relacionada con números enteros que define la operación de adición "+"', un refinamiento consistiría en tomar este concepto y relacionarlo con una ontología de matrices y conformar la suma de matrices. El proyecto KACTUS [47] fue concebido mediante el refinamiento incremental de ontologías genéricas hacia ontologías más específicas y técnicas.

Por otro lado, si la creación de una nueva ontología surge a partir de la combinación de otras existentes, se pude hacer la distinción entre *fusión*, `alineación` y `relación` de ontologías [137]:

- ***Fusión*** *(merging)*: cuando varias ontologías son fusionadas, las ontologías existentes son reemplazadas por una nueva ontología que unifica los conceptos y relaciones de las ontologías originales.

- ***Alineación*** *(aligning)*: el proceso de alineación consiste en lograr un mutuo acuerdo entre ontologías. Esto significa adaptar a una ontología, a las conceptualizaciones y el vocabulario de la otra; especialmente en las definiciones sobrepuestas de las ontologías a unir. Sin embargo, pueden surgir importantes problemas con este enfoque [252]; como por ejemplo la disparidad en el nivel del lenguaje ontológico, ya sea su sintaxis o expresividad. Además puede darse que los modelos ontológicos respondan a diferentes paradigmas o estilos de construcción.

- ***Relación*** *(relating)*: en el proceso de relación entre dos ontologías, se definen varios axiomas que describen relaciones entre conceptos de sendas ontologías.

En el ambiente de la Web Semántica, altamente desestructurado y distribuido, las ontologías serán utilizadas por varias aplicaciones y reutilizadas en varios escenarios. Esto significa que de optarse por la fusión o alineación de ontologías probablemente se incompatibilicen varias aplicaciones que tienen dependencias de ciertas definiciones en las ontologías existentes; a no ser que se preserve las versiones anteriores, generando un grado importante de redundancia.

Por este motivo, establecer procedimientos de relaciones entre ontologías heterogéneas es la opción viable para la Web Semántica, ya que los conceptos y relaciones propios de cada ontología existente se respetan. Solamente se agregan los axiomas que vinculan los conceptos de una ontología con otra [137]. Sin embargo, cuando las diferencias conceptuales entre las ontologías son profundas y/o son muchas, el proceso de relación sólo puede realizarse en forma parcial; para solucionar este impedimento, es necesario realizar una cierta cantidad de alineamiento entre las ontologías involucradas.

En la mayoría de los métodos tradicionales para crear nuevas ontologías (como los presentados en [456]), las nuevas ontologías son creadas preservando las originales. Esto es por la suposición que las ontologías originales, usadas para crear nuevas, fueron diseñadas con normas de buen diseño y modularidad. Sin embargo en los ambientes amplios como la Web Semántica esto puede no darse [126].

Metodologías en Ingeniería de Ontologías

Con la meta de lograr que el conocimiento que contienen pueda ser compartido y reutilizado, las ontologías necesitan ser reutilizadas en gran escala. Sin embargo, para lograr esto es necesario que las ontologías, al igual que los servicios en una arquitectura SOA (véase Sec. 7.2.5), sean diseñadas con la idea de reuso en mente.

En la Sec. 7.4.1 se presentaron dos clasificaciones importantes de ontologías basadas en la generalidad y expresividad de las mismas. Las ontologías genéricas tienen los más altos requerimientos en reusabilidad. En estos casos el criterio de diseño de mínimos compromisos ontológicos es el más importante de aplicar, debido a la generalidad de los conceptos a aplicar; en tanto que mantener mínima la tendencia de codificación no sería mayor problema porque se diseña sin tener ningún escenario ni aplicación en particular que pueda requerir codificación específica [126].

Cuando se crean ontologías de dominio, como es el común de los casos, estos criterios de diseño son ambos importantes de aplicar. Es importante tener en cuenta a la hora de diseñar estas ontologías de dominio el escenario y la aplicación en los cuales será usada. La calidad y cantidad de compromisos ontológicos exige un balance entre reusabilidad y usabilidad. Es deseable compartir una ontología de un dominio específico entre los actuadores de ese dominio (desarrolladores de ontologías, gerentes de negocios, expertos del dominio, usuarios, etc.). Se espera que la ontología sea utilizada a través de intercambio de mensajes, para poder desacoplarla de cualquier ambiente heterogéneo propio de cada actuador. Estos mensajes deben ser consensuados entre las partes que utilizan la ontología. El significado de cada mensaje debe contar con un compromiso ontológico entre las partes que posibiliten niveles adecuados de usabilidad de la ontología.

Cuando se crea una ontología de aplicación es inevitable tener bastante tendencia a la codificación en los términos y relaciones, ya que deben ser lo suficientemente específicos para una aplicación en particular. Desde esta perspectiva, minimizar los compromisos ontológicos no tiene sentido porque la ontología no está pensada para reuso mas allá de su aplicación principal.

Debe notarse que si bien la discusión anterior se centró en los principios de minimización de compromisos ontológicos y de tendencia a la codificación, no es excluyente el tratamiento de los otros tres principios (claridad, coherencia y extensibilidad). Sin embargo los dos primeros dan un gran impacto a la cualidad de reuso de una ontología. De la misma forma que lograr servicios agnósticos, implica lograr servicios reutilizables, lograr ontologías genéricas sin gran cantidad de compromisos ontológicos aumenta las probabilidades de reuso. Algunas metodologías que abarcan el reuso y la integración con otras ontologías como actividad indicada en su proceso se detallan

a continuación:

El modelo esquemático metodológico Para iniciar el tratamiento de las metodologías, es importante destacar que a mediados de los 90 se presentó el ***modelo esquemático*** *(skeletal model)* de construcción de ontologías. [509] El modelo esquemático consiste en una serie de etapas, cada una de las cuales debe producir un entregable. Las etapas fundamentales son:

- ***Identificación del propósito y alcance***: es esencial determinarlos, fundamentalmente para determinar el nivel de genericidad; es decir, si la ontología a crear va a ser genérica, de dominio o de aplicación. Es importante establecer propósito y alcance entendiendo que no es recomendable desviarse de estos en el proceso metodológico.
- ***Construcción***: la etapa más fundamental del esquema. Se pueden distinguir los siguientes sub-pasos:
  - ***Captura***: de los términos y relaciones del dominio que son identificados y acordados para ser tratados. El resultado de la sub-etapa de captura es una conceptualización consensuada del domino.
  - ***Codificación***: la conceptualización capturada en la etapa previa es efectivamente codificada en un lenguaje ontológico.
  - ***Integración***: la integración con ontologías existentes debe producirse siempre que sea posible. Muchos componentes ontológicos son reutilizables y autocontenidos y pueden ser acoplados a la ontología en desarrollo. Esta sub-etapa debe ser interrelacionada con las dos previas, ya que los resultados de las posibilidades de reuso e integración afectan las decisiones respecto a la captura y codificación de la ontología.
- ***Evaluación***: para lograr una evaluación efectiva de la ontología, es necesario un marco de referencia [382]. Dicho marco de referencia puede contener desde análisis y validación de requerimientos, testeo de la competencia de preguntas, etc.
- ***Documentación***: las descripciones formales producidas deben ser documentadas en forma informal. Es deseable que todas la suposiciones y justificaciones de decisiones de diseño sean debidamente documentadas también.

Algunos trabajos que evolucionaron el concepto del esquema de modelo, plantean técnicas de captura de ontologías. Para definir el alcance de la ontología, se recomienda utilizar la *"tormenta de ideas (brainstorming)"* que produce potenciales términos y relaciones relevantes, y el *agrupamiento* de tales términos y relaciones en áreas de trabajo, necesitando posiblemente un referenciamiento cruzado entre áreas [508]. Por otro lado, para producir las definiciones ontológicas, es esencial (1) determinar la *meta-ontología*. Una ***meta-ontología*** especifica los conceptos necesarios para describir a la ontología en sí. (2) Tratar cada área de trabajo por vez, comenzando con las áreas conflictivas y superpuestas. (3) Definir los términos, primero a un nivel básico cognitivo y luego los más abstracto y detallado. Esto constituye un enfoque *middle-out*; se considera más fácil comenzar a trabajar en un nivel familiar de conocimiento, y luego lograr los términos abstractos genéricos y las definiciones más específicas. Como última tarea de la producción de definición es necesario lograr acuerdo respecto a todas las definiciones. Para culminar la captura de la ontología, es necesario realizar revisiones constantes y documentar el historial de cambios.

Metodología TOVE. Como ejemplo de utilización del modelo esquemático se puede citar a la metodología propuesta en base a las experiencias obtenidas en el proyecto TOVE (Toronto Virtual Enterprise) [198]. Dicha metodología consiste en los siguientes pasos:

- ***Captura de escenarios motivadores***: los escenarios son capturados para clarificar el dominio y utilización de la ontología.

- ***Formulación informal de preguntas de competencia***: estas presuntas son formuladas para determinar la expresividad de la ontología.

- ***Especificación de la terminología en un lenguaje formal***: la metodología recomienda un lenguaje basado en la Lógica de Primer Orden. La especificación debe ser hecha de forma tal, que se pueda contestar las preguntas.

- ***Formulación formal de preguntas de competencia***: las preguntas son reformuladas utilizando el lenguaje ontológico y solamente con los términos definidos para la ontología.

- ***Especificación de los axiomas y definiciones de los términos***: las especificaciones se las realizan en el lenguaje formal elegido para la ontología; es decir, en lógica de primer orden. Es la etapa de codificación de la ontología.

- ***Justificación de axiomas y definiciones***: se prueban los teoremas y se verifican que las preguntas de competencia son debidamente contestadas.

Una desventaja de esta metodología es que utiliza un enfoque bastante rígido, basado en lógica de primer orden, para expresar la ontología. Esto significa que el método es inviable cuando se desea lograr ingeniería de una ontología menos formal. Es importante destacar, que en ningún paso se preveé la integración y reuso de términos ya definidos en otras ontologías.

METHONTOLOGY. Las ideas de ***METHONTOLOGY*** [164] se basan en el ciclo de vida propuesto por el Proceso de Desarrollo de Software del IEEE [233]. En otras palabras trata de utilizar un esquema estándar como directriz de las actividades de la metodología. De una manera sintética, las etapas de METHONTOLOGY son:

- ***Especificación***: Se debe consensuar un documento que establezca el propósito de la ontología, su nivel de formalidad [508] (informal, semi-formal o formal) y el alcance incluyendo el conjunto de términos a representar. Se recomienda en este último caso, utilizar el enfoque middle-out, explicado anteriormente.

- ***Adquisición de conocimiento***: Se deben estipular y capturar todas las fuentes de conocimiento posibles, como ser el análisis de documentos, entrevistas con expertos del dominio, análisis de fuentes de datos normalizadas (base de datos y sistemas documentales), etc.

- ***Conceptualización***: En primer lugar se construirá un completo glosario de términos. Los conceptos son modelados en árboles de clasificación de conceptos, y las relaciones en diagramas de verbos.

- ***Integración***: el propósito fundamental de este paso es reutilizar conceptos definidos en otra ontología. Para lo cual se exploran e inspeccionan los metadatos (los conceptos utilizados para modelar la ontología) de las ontologías existentes para seleccionar aquellos que mejor se ajustan a la conceptualización de la metodología en desarrollo. Luego se buscan los términos coherentes con la conceptualización realizada.

- ***Implementación***: la ontología es codificada en un lenguaje ontológico.

- ***Evaluación***: se verifica la correctitud de la ontología.

- ***Documentación***: en cada etapa previa se deben documentar el desarrollo de la ontología. Los documentos producidos en cada etapa, documentan el propósito, alcance, intensionalidad y expresividad de la ontología.

METHONTOLOGY provee un método bastante comprensible del proceso de construcción de ontologías basado en estándares de IEEE. Es un buen candidato para ser aplicado en la ingeniería de ontologías genéricas; aunque puede abarcar la concreción de ontologías con rango variable en genericidad y expresividad. Establece explícitamente una etapa para la integración y reuso de ontologías existentes.

Metodología On-To-Knowledge. La metodología ***On-To-Knowledge*** [373, 450] está orientada a la concreción de sistemas de gestión de conocimiento empresarial, orientados a integración de varias fuentes de datos, ya sea documentales o informáticamente estructuradas (bases de datos). En la metodología se hace la distinción entre el *proceso de conocimiento* y el `proceso de meta-conocimiento`. El primero tiene relación con el uso real del sistema de gestión de conocimiento; es decir, con las funciones de adquisición y recuperación. El último tiene que ver con el tipo de gestión de conocimiento puesto en el sistema. Viéndolo desde el punto de vista ontológico, el primero se refiere a la utilidad de la ontología, y el segundo se refiere a la construcción, puesta en marcha y mantenimiento de ontologías.

El proceso de meta-conocimiento propuesto por On-To-Knowledge consiste en los siguientes pasos [163]:

- ***Estudio de factibilidad***: que sirve para determinar si corresponde o no construir la ontología. Los autores proponen utilizar técnicas basadas en el método CommonKADS [105].
- ***Desarrollo***: es la primera fase real del proceso de construcción. Esta fase debe lograr como producto una especificación de los requerimientos de la ontología. Es en esta fase cuando deben contemplarse las posibilidades de integración y reuso de otras ontologías. También deben formularse las preguntas de competencia para lograr capturar los requerimientos.
- ***Refinamiento***: en esta fase de desarrollo de la ontología orientada. Dicho desarrollo se lleva a cabo de acuerdo a las especificaciones realizadas en la etapa previa. Esta etapa puede dividirse en: (1) conformación de una taxonomía básica, (2) creación de una ontología primitiva que le agrega a la taxonomía básica las relaciones entre los términos y (3) utilización de la ontología primitiva para lograr una versión definitiva expresada en lenguaje formal.
- ***Evaluación***: es la etapa final del proceso de desarrollo, sin embargo está en continua recursividad con la etapa previa de refinamiento. En este nivel se controla la ontología con sus especificaciones y a través de las preguntas de competencia. Además se corrobora el funcionamiento de la ontología en el ambiente en dónde se la implementará definitivamente.
- ***Mantenimiento***: luego que se conforma la ontología, se inicia la fase de mantenimiento, en la cual los cambios producidos en el dominio objetivo deben verse reflejados en la ontología.

El proyecto On-To-Knowledge está orientado a la construcción de sistemas de gestión de conocimiento. Hace hincapié en la recuperación y estructuración del conocimiento empresarial cuyas fuentes de información son de las más variada naturaleza. Se puede pensar en integrar documentos, bases de datos, información de contacto, etc. Una vez que un sistema de gestión de conocimiento está implementado, el proceso conocimiento tiene cuatros pasos básicos [450]: (1) la creación o importación de datos, (2) la captura del conocimiento (encerrado en esos datos) de acuerdo a los conceptos de la ontología, (3) el acceso y recuperación del conocimiento desde la base de conocimiento con el uso de los conceptos ontológicos, y (4) el uso de ese conocimiento para el logro de objetivos concretos.

Las ontologías son una especialización de una conceptualización [195]. Como tal, describen el conocimiento que esta conceptualización encierra. El ánimo principal de la Web Semántica

es poder exponer tales conceptualizaciones, a través del uso de tecnologías estándares, para que sea más fácil la integración de tales conocimientos. En este sentido, las tecnologías de la Web Semántica, son para la integración de conocimiento, lo que los Servicios Web son para la integración de aplicaciones. Por ser ambos paradigmas promotores de la integración, resolviendo las barreras de interoperatividad, también son grandes promotores de reuso.

Las ontologías definen términos, conceptos, relaciones entre conceptos y reglas. La integración entre diferentes ontologías plantea un desafío en la ingeniería del conocimiento requerido para que dichas ontologías se vinculen en forma coherente. En integración de ontologías, la limitación de representación e implementación es superada por la serialización en lenguajes ontológicos estandarizados, como RDF(S) y OWL. Sin embargo, el principal esfuerzo es el establecer la semántica de vinculación entre ontologías. Este esfuerzo será menor si cada ontología es desarrollada teniendo la idea de que sus términos y definiciones pueden ser reutilizados por otra ontología. Es por eso que el reuso vuelve a ser una cualidad significativa, al igual que en los Servicios Web, para dar oportunidades ciertas para el logro de los objetivos de la Web Semántica.

7.5. Reutilización en Servicios Web Semánticos

Por lo tratado en anteriores capítulos, se estableció que los Servicios Web son el medio para proveer una plataforma interoperable para la integración de aplicaciones mediante el uso de las tecnologías de Web. También se puede ver que las tecnologías de Segunda Generación de Servicios Web proveen soporte a varias funcionalidades esenciales en un marco de integración, siendo las más destacadas las orientadas a la coordinación y composición de servicios. Estas especificaciones brindan el marco de trabajo para que sea posible la integración de varios servicios web, posiblemente de diferentes organizaciones o ambientes. Sin embargo, cada esfuerzo de integración debe ser diseñado específicamente por desarrolladores, indicando el rol de cada servicio participante a integrar, estudiando las interfaces y capacidad de dichos servicios, y estableciendo la secuencia de interacciones entre participantes. Este diseño se puede plasmar en una aplicación cliente que preserva la lógica de integración o en otros servicios web; por ejemplo, un documento WS-BPEL, es un script de ejecución de una secuencia de interacciones entre servicios web. Lo que falta, en este estado de evolución de los Servicios Web, es la capacidad para que las integraciones dinámicas sean posibles. Es decir, dar la posibilidad que las integraciones se generen en forma dinámica sin la intervención de un experto que indique cómo debe ser el diálogo entre Servicios Web.

7.5.1. Concepto de Servicio Web Semántico

Para lograr que los sistemas informáticos y agentes puedan implementar interoperatividad dinámica, confiable y a gran escala de servicios web, es necesario que las descripciones de dichos servicios web sean comprensibles (procesables) por computadoras. De esta forma, se concibe la idea de lograr una Web Semántica de servicios web, cuyas capacidades, propiedades, interfaces y efectos son codificados en una forma no ambigua y orientada al entendimiento de la máquina [162].

Como se explicó en la Sec. 7.4.1, la Web Semántica provee una serie de marcos de trabajos y lenguajes que hacen posible que la semántica encerrada en entidades de la Web sean comprensible por agentes y componentes de software. Con estas tecnologías, en especial con los lenguajes ontológicos, es posible establecer semánticas bien definidas y permitir la manipulación de complejas taxonomías y ontologías que describen las relaciones lógicas entre dichas entidades. De esta forma, surge en forma natural la idea de aplicar la Web Semántica a los Servicios Web.

El término ***Servicios Web Semánticos*** fue concebido al inicio de la década del 2000 [289]. Es un concepto que se utiliza para referenciar al conjunto de principios, tecnologías y métodos para *automatizar* las tareas comunes asociadas a los Servicios Web; como ser descubrimiento, selección, composición y promoción de servicios. Estas tareas logran hacer que los propios ser-

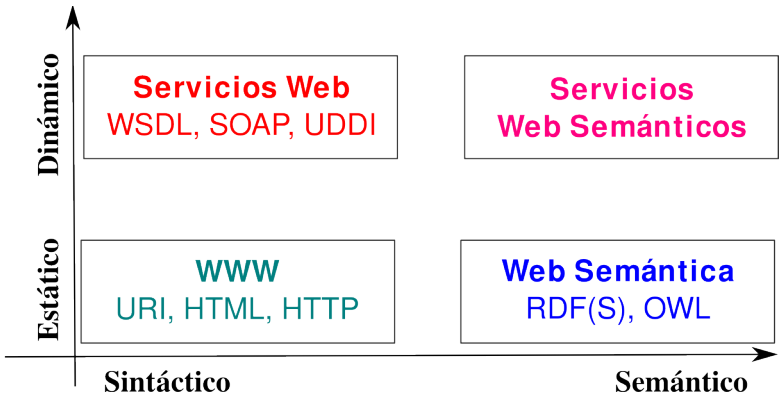

Figura 7.7: La evolución de la Web.

vicios web sean entidades con semánticas procesables por componentes informáticos o agentes; esto es lo que hace muy atractiva a la idea de los Servicios Web Semánticos.

El beneficio potencial de los Servicios Web Semánticos ha permitido el establecimiento de importantes áreas de investigación, tanto en el ámbito académico como en el empresarial. En si, el término ***Servicios Web Semánticos*** compromete dos importantes bloques de construcción que deben ser adecuadamente combinados para lograr la automatización en el uso de servicios web:

- La Semántica Web que agrega semántica procesable a los datos. La computadora puede "entender" esa meta-información y procesarla en representación de un usuario humano.
- Los Servicios Web intentan emplazar a la Web como infraestructura global para la computación distribuida, para la integración de varias aplicaciones y para la automatización de procesos de negocios. La Web no sólo sería el lugar para publicar información legible por personas, sino el marco de trabajo para que la computación sea realizada.

En la Fig. 7.7 se pueden ver los dos aspectos de la evolución natural de la Web. Los Servicios Web, con la capacidad para proveer de abstracción a los recursos computacionales distribuidos, convierten a la Web en un ambiente más dinámico, aunque preserve aún su naturaleza puramente sintáctica. La visión de los Servicios Web Semánticos es describir y especificar varios aspectos del servicio utilizando semántica comprensible por computadoras, para hacer posible la automatización de la ubicación, combinación y uso de servicios web. El trabajo producido en el área de la Web Semántica es aplicado a los Servicios Web para minimizar la intervención de los desarrolladores y usuarios humanos. El **demarcado semántico** es aprovechado para automatizar las tareas de descubrimiento, ejecución e integración natural entre servicios web; de esta forma, se puede disponer de servicios web "inteligentes" [280].

La descripción de los servicios, en una forma comprensible por máquinas, tiene un importante impacto en las áreas de comercio electrónico e integración de aplicaciones empresariales. De esta manera se brinda la posibilidad para la cooperación dinámica y escalable entre los diferentes sistemas y organizaciones. Las descripciones en los estándares de Servicios Web como el WSDL tiene una importante desventaja en este aspecto, ya que están limitadas a cubrir los aspectos sintácticos de la interacción de servicios. Como se vio en Sec. 5.7, las descripciones WSDL establecen cómo los servicios deben ser accedidos, cuáles operaciones pueden ser invocadas, cuáles políticas deben ser soportadas, etc. Sin embargo, lo que hace el servicio y cuál es el orden en que las operaciones deben ser llamadas para obtener cierta funcionalidad, solamente es descripta en lenguaje natural en los comentarios de los documentos WSDL o en las entradas del registro UDDI.

Se puede esperar que la tasa de crecimiento de los servicios web en la Web se incrementará exponencialmente de la misma forma que las páginas web estáticas lo han hecho una década antes.

Y de la misma forma, cuando se intente buscar un servicio web adecuado para determinadas necesidades, se presentarán los mismos obstáculos en el filtrado de información relevante en una masa enorme de información humano-legible, pero sin meta-descripciones que sirva a los agentes y sistemas (informáticos) de búsqueda. Es por eso que las anotaciones orientadas al significado propuestas por la Web Semántica son la gran esperanza. La convergencia de ambas tecnologías es esencial para el éxito de los Servicios Web a escala global.

Los requerimientos para las descripciones de los servicios web semánticos se corresponden con la de los servicios web "tradicionales". Sin embargo, es importante que tales descripciones utilicen a las ontologías como vocabularios comunes, para lograr los altos niveles de automatización. En particular, se pretende tener el mayor potencial de automatización en las siguientes tareas relacionadas a los servicios:

- ***Descubrimiento***: para que un servicio web publicado en la Web o en una intranet empresarial sea utilizado, debe poder localizarse primero. La tecnologías bases de los Servicios Web, en especial UDDI, soportan esta tarea mediante la búsqueda sintáctica por palabras claves. A pesar que existen formas limitadas de estandarización de vocabularios, como el propuesto por UNSPSC [506] (véase Sec. 5.8.2), la búsqueda sintáctica es el método más ampliamente utilizado. Como alternativa opuesta, se proponen anotaciones semánticas de las capacidades de servicios mediante ontologías descentralizadas, interconectadas por axiomas lógicos. Gracias a que los servicios web son expresados en términos ontológicos, es posible determinar, mediante inferencias lógicas, cuáles servicios web se corresponde con los requerimientos. Estas descripciones pueden involucrar nociones más finas como pre y post-condiciones, y la descripción de entradas y salidas basándose en términos ontológicos.

- ***Negociación***: Cuando se determina que una aplicación proveedora de servicios puede brindar los servicios requeridos, es necesario que ésta pueda establecer una instancia de negociación con la aplicación/servicio cliente. La negociación entre servicios web incluye establecer políticas de confianza, determinación de modalidades de pago, selección de ofertas y otras. Para lograr que esta tarea se automatice, las anotaciones semánticas no sólo deben cubrir dichos aspectos, sino que también deben soportar protocolos y políticas de seguridad.

- ***Composición***: en los casos que un objetivo no pueda ser logrado por un único servicio web, las descripciones semánticas pueden ayudar a determinar una combinación de mútiples servicios que logren la funcionalidad requerida. La composición requiere no sólo de anotaciones semánticas de las capacidades integrales del servicio, sino también una descripción del comportamiento que se espera al interactuar con otros servicios.

- ***Invocación***: después que un servicio, o una combinación de servicios, ha sido seleccionada, el paso final es la ejecución. Para lograr esto, los valores posibles de entrada y salida deben ser obtenidos a partir de las descripciones semánticas y adaptadas a los protocolos y formatos de mensajes negociados.

Varios de los avances propuestos hacia el logro de los Servicios Web Semánticos, y hacia las automatizaciones de las tareas anteriormente descriptas, tienen relación con el agregado de anotaciones a las descripciones de cada servicio, extendiendo a las descripciones de las tecnologías fundamentales WSDL, SOAP, UDDI, etc. con anotaciones semánticas.

A continuación se verán algunas propuestas importantes para soportar la definición y desarrollo de Servicios Web Semánticos. En especial se estudiará la propuesta de la Ontología para el Modelado de Servicios Web (WSMO) [162] y se enunciarán otros formalismos comparativamente.

7.5.2. Ontología para el Modelado de Servicios Web (WSMO)

La ***Ontología para el Modelado de Servicios Web*** o ***WSMO*** *(Web Service Modeling Ontology)* [154, 414, 128, 162] es un marco de trabajo que está dirigido a la descripción de

todos los aspectos relevantes de todas las capacidades que son accedidas como servicios web. Su propósito fundamental es permitir la automatización, total o parcial, de las tareas involucradas en la integración de servicios web ya sea inter o intra-empresarial. Con este marco de trabajo, se busca fundamentalmente automatizar las tareas descriptas en la sección anterior; es decir, descubrimiento, selección, composición, mediación, ejecución y monitoreo.

WSMO provee un marco de trabajo conceptual y un lenguaje formal para describir semánticamente todos los aspectos de un servicio web [127]. WSMO tiene su origen conceptual en el Marco de Trabajo para Modelamiento de Servicios Web o ***WSMF*** *(Web Service Modeling Framework)* [160]; y como tal, propone refinar y extender este marco de trabajo para desarrollar una ontología formal y un conjunto de lenguajes. A continuación se describen brevemente los conceptos de WSMO, los principios de diseño que sustenta y sus elementos fundamentales.

Principios de diseño de WSMO

WSMO provee especificaciones ontologícas para los principales elementos de los Servicios Web Semánticos. Como tal, debe sustentar adecuados principios de diseño que se adecuen a los principios básicos del diseño Web, a los principios definidos por la Web Semántica, y a los principios de diseño de la computación distribuida orientada a servicios sobre la Web. Por ende, WSMO se basa en los siguientes principios de diseño:

- ***Compatible con Web***: WSMO adopta a las normas de URIs y Espacio de Nombres (véase Sec. 4.1.2 y 5.5.3). También da soporte a XML y a otras recomendaciones del W3C.
- ***Basado en Ontologías***: las ontologías son usadas como modelos de datos en WSMO. Esto significa que todas las descripciones de recursos y el intercambio de datos es basado en ontologías. Como se vio en la Sec. 7.4.1, las ontologías son un componente principal de la web Semántica y permite mejorar semánticamente el procesamiento de información y el soporte a la interoperatividad. WSMO soporta los lenguajes ontológicos definidos en la Web Semántica.
- ***Desacoplamiento estricto***: la exigencia de desacoplamiento denota que los recursos WSMO son definidos aisladamente, sin importar sus posibles usos e interacciones con otros recursos; en concordancia con la naturaleza de la Web.
- ***Centralización de la mediación***: la mediación está destinada a manejar las heterogeneidades que pueden ocurrir en términos de datos, ontologías subyacentes, protocolos, etc. WSMO reconoce la importancia de la mediación, a partir de los exitosos desarrollos en Servicios Web, estableciéndola como un componente fundamental del marco de trabajo.
- ***Separación ontológica de roles***: WSMO entiende que los clientes de servicios están en un diferente contexto y rol que los servicios web proveedores. WSMO establece diferencia entre los deseos de los clientes y las capacidades disponibles por los servicios.
- ***Descripción vs. Implementación***: WSMO hace la distinción entre las descripciones de los elementos de los servicios web semánticos y sus tecnologías de implementación y ejecución. Mientras que las primeras requieren un marco de trabajo basado en formalismos para lograr descripciones concisas, las últimas se relacionan con el soporte a las tecnologías de ejecución subyacentes (como por ejemplo WSDL). WSMO tiene como propósito proveer un adecuado modelo ontológico descriptor, y ser compatible con las tecnologías de implementación existentes y emergentes (no limitándose a WSDL).
- ***Semántica de Ejecución***: con el propósito de verificar las especificaciones WSMO, existen ambientes de implementación que soportan semánticas formales para la ejecución. Un ejemplo es el ***Ambiente de Ejecución para el Modelamiento de Servicios Web*** o ***WSMX*** *(Web Service Modelling eXecution environment)* [542, 89].

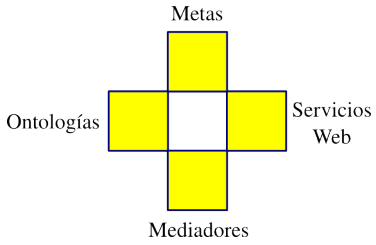

Figura 7.8: Componentes Principales de WSMO.

- ***Servicios vs Servicios Web***: un servicio web es una entidad computacional capaz de lograr determinadas metas por invocación. Un servicio, por el contrario, es el valor real provisto por tal invocación [30, 395]. WSMO posibilita describir servicios web que proveen acceso a servicios.

Componentes de WSMO

WSMO identifica cuatro componentes como conceptos principales en su marco de trabajo: *ontologías*, `servicios`, `metas` y `mediadores` (Véase Fig. 7.8).

- ***Ontologías*** *(Ontologies)*: como se explicó, las ontologías proveen la terminología a ser utilizada y son el elemento esencial para el éxito de los Servicios Web Semánticos. WSMO describe una *epistemología* de ontologías; es decir, define sus elementos constitutivos fundamentales como ser conceptos, relaciones, etc.

- ***Servicios Web*** *(Web Services)*: describen a las entidades computacionales que proveen determinado valor a un cierto dominio. Las descripciones de servicios web consisten en sus capacidades, interfaces, y manejo interno.

- ***Metas*** *(Goals)*: describen los aspectos relacionados a los deseos de los usuarios (consumidores de servicios) respecto a una funcionalidad requerida. En este punto, se utilizan ontologías para describir los aspectos relevantes de las metas. De alguna manera, las metas modelan el punto de vista de un cliente respecto al proceso de uso de un servicio web, por ende son entidades separadas en WSMO.

- ***Mediadores*** *(Mediators)*: describen elementos encargados de gestionar los problemas de interoperatividad entre diferentes elementos, como ser diferentes ontologías o servicios. Los mediadores son utilizados para resolver incompatibilidades entre terminologías (a nivel de datos), para la comunicación de servicios y la correspondencia entre servicios web y metas requeridas.

Definición de Ontologías

Las ontologías definen una terminología acordada común para proveer conceptos y relaciones entre conceptos. Para lograr capturar las propiedades semánticas de relaciones y conceptos, una ontología generalmente también provee un conjunto de axiomas. Las ontologías de WSMO son clases de primera categoría, y en su definición se pueden describir los siguientes elementos:

Propiedades No-funcionales. Las ***propiedades no-funcionales*** son aplicables a todas las definiciones de elementos WSMO. Son usadas para describir aspectos como el nombre del creador y de la fecha de creación de la definición, y son provistas generalmente en lenguaje natural. Los elementos definidos por la iniciativa de metadatos de *Dublin Core* [531, 124] son tomados como recomendación para definir estas propiedades (véase Sec. 7.4.1). Dubin Core es un conjunto de

atributos que define un estándar para información descriptiva de recursos. También se utiliza la norma de URIs [49] para identificar elementos.

Ontologías Importadas. Como se explicó en las Sec. 7.4.1, una forma para manejar la complejidad en la definición de ontologías es la modularización. Poder importar ontologías permite un enfoque modular en el diseño de ontologías. Cuando WSMO incluye en la definición de alguno de sus elementos otras ontologías, significa que asume las definiciones expresadas en esa ontología para utilizar su vocabulario.

Mediadores. Cuando se importan metodologías en escenarios reales, se requieren de algunos pasos para alinear, fusionar o transformar la ontología importada para salvar las incompatibilidades (véase Sec.7.4.3). En estos casos WSMO utiliza un tipo especial de mediadores denominados *mediadores ontológicos* o *Mediadores OO* que sirven cuando se requiere cierta alineación con las ontologías importadas. Dicha alineación puede darse a través del renombre de conceptos, atributos o similares. Las sentencias de mediador también pueden aplicarse a cualquiera de los cuatro elementos fundamentales de WSMO.

Conceptos. Los ***conceptos*** constituyen los elementos básicos de la terminología ontológica. La descripción de un concepto provee atributos con nombres y tipos. También es posible que un concepto sea directamente un *sub*concepto de ninguno, uno o varios *super*conceptos. La herencia así establecida en WSMO, hace que un concepto herede la signatura del superconcepto y las restricciones correspondientes, ya que refleja la relación *"es-un"*. Adicionalmente un concepto puede ser definido por una expresión lógica; es decir, se puede establecer un axioma que refina o restrinja el significado heredado de los superconceptos.

Relaciones. Las ***relaciones*** modelan interdependencias entre varios conceptos, o instancias de los mismos. La aridad de las relaciones no está limitada en WSMO. Una relación puede también heredar de una *super*relación sus signaturas y restricciones. Es por eso que el conjunto de tuplas que pertenecen a una relación, son un subconjunto de cada una de las superrelaciones. De igual forma que los atributos en los conceptos, una relación puede tener un conjunto de nombres de parámetros. Cada parámetro es un valor único y puede establecerse una restricción de rango en la forma de concepto. También, al igual que los conceptos, es posible establecer axiomas para restringir o refinar la definición de la relación.

Funciones. Las ***funciones*** son un tipo especial de relación, las cuales tienen un rango unario y un dominio n-ario; dónde el rango especifica un valor retornado. Las funciones pueden ser utilizadas para representar y exponer los predicados predefinidos de los tipos de datos comunes. Su semántica puede ser capturada externamente, o mediante restricciones a relaciones ya definidas.

Instancias. Las ***instancias*** son definidas explícitamente o por medio de un vínculo a un repositorio de instancias [251]. Esto intenta integrar grandes conjuntos de instancias que preexisten en un almacén de datos mediante el envío de consulta a dicho almacén o repositorio.

Definición de Servicios Web

Al definir un servicio web, WSMO provee un modelo conceptual para describir, en una forma explícita y unívoca, todos los aspectos de un servicio, que incluyen las propiedades no-funcionales, su funcionalidad y las interfaces necesarias para accederlo. Un modelo sin ambigüedades de servicios con una semántica bien formada puede ser procesado e interpretado por agentes y sistemas informáticos, sin la intervención de personas, permitiendo la automatización de las

tareas involucradas en el uso de servicios web. De esta forma se logra el principal objetivo de los Servicios Web Semánticos.

Como aspecto central de la definición de servicios, WSMO provee un punto de vista unificado para servicios. El valor que los servicios pueden proveer es capturado como su *capacidad*. El medio para interactuar con éste y requerir su ejecución, o los modos para negociar aspectos relacionados con su provisión, son características capturados en su *interface*. Los principales elementos de una definición WSMO de un servicio web se detallan a continuación.

Propiedades No-funcionales. Al igual que la descripción de ontologías, es posible agregar información no-funcional a la descripción de un servicio web en WSMO. Algunas de las descripciones no-funcionales para un servicio web pueden ser: *exactitud (accurancy)* indicando precisión y tasa de error generados por el servicio, *financiación (financial)* representando precio y costo asociado a cada capacidad del servicio, *dueño (owner)* indicando la organización o persona que tiene la propiedad del servicio, *rendimiento (performance)*, *fiabilidad (reliability)*, *escalabilidad (escalability)*, *seguridad (security)*, *versión (version)*, etc. Estas definiciones no-funcionales son usadas principalmente para el descubrimiento y selección de servicios; sin embargo, alguna información es adecuada para procesos de negociación.

Ontologías Importadas. Ontologías externas, o definidas en otro módulo WSMO, que son utilizadas para proveer vocabulario formal en la definición del servicio web.

Mediadores. Un servicio web puede utilizar mediadores en las siguientes situaciones:

1. cuando terminología heterogénea es utilizada y se producen conflictos, una definición WSMO de servicio web puede utilizar mediadores ontológicos (Mediadores OO), iguales a los explicados en la definición de ontologías; o

2. cuando un servicio web necesita lidiar con heterogeneidad relacionada a los protocolos o procesos al interaccionar con otros servicios. En este caso, se utiliza un tipo especial de mediador, denominado *Mediador WW*, cuya descripción se verá más adelante.

Capacidades. Las ***capacidades*** son parte esencial en las descripciones WSMO de servicios web y tratan de capturar la funcionalidad del mismo. Fundamentalmente se expresan mediante la configuración del mundo antes que el servicio sea ejecutado y el estado del mundo luego de la ejecución exitosa de la capacidad/funcionalidad del servicio. El detalle de las capacidades es utilizado en la automatización del descubrimiento y selección de servicios; es decir, sirve para que un servicio consumidor pueda determinar si un servicio cumple o no sus necesidades.

Las capacidades son descriptas en WSMO en base a:

- ***propiedades no-funcionales***, ***ontologías importadas*** y ***mediadores***, cuyas fundamentaciones son las mismas que las explicadas para sus respectivos usos en la definición de ontologías.

- ***precondiciones***: son condiciones, materializadas por axiomas, del estado requerido del espacio de información antes de la ejecución de la capacidad de un servicio web. Las precondiciones restringen el conjunto de estados de espacio de información posibles para ejecutar la capacidad en la manera definida.

- ***suposiciones***: describen el estado del mundo que es supuesto antes de la ejecución de una capacidad. De otra forma, no es garantizada la adecuada provisión del servicio. A diferencia de las precondiciones, las suposiciones no exigen que sean corroboradas. Es necesaria esta distinción para que pueda ser posible explicitar condiciones del mundo que estan fuera del espacio de información.

- ***postcondiciones***: describen el estado del espacio de información que se garantiza alcanzar luego de la ejecución exitosa de la capacidad. Además, es una forma de describir la relación que existe entre la información provista para la ejecución del servicio y los resultados obtenidos.

- ***efectos***: describen el estado del mundo, más allá del espacio de información, que se alcanzará si se ejecuta con éxito la capacidad de servicios y si se cumplen las suposiciones.

Interfaces. Una definición de ***interfaz*** en WSMO describe la manera en la cual la funcionalidad de un servicio puede ser invocada y llevada a cabo. Para esto, se proveen dos puntos de vistas complementarios para describir la competencia operacional del servicio: (1) *Coreografía (choreography)* que descompone la capacidad en términos de interacciones con el servicio, y (2) *orquestación (orchestration)* que descompone una capacidad en términos de las funcionalidades requeridas de otros servicios. Esta distinción en el modelo WSMO refleja la diferencia entre comunicación y cooperación (véase Sec. 6.4 y 6.5). La coreografía define cómo comunicarse con el servicio para consumir su funcionalidad. La orquestación define cómo la funcionalidad integral es lograda a partir de la cooperación de proveedores de servicios más elementales.

Las interfaces de servicios web descriptas en WSMO son presentadas de forma que puedan ser comprensibles por agentes de software para que determinen el comportamiento del servicio y puedan razonar sobre éste. También son utilizadas para propósitos de descubrimiento y selección de servicios. Eventualmente, se las puede vincular con especificaciones de servicios web existentes, como pueden ser documentos WSDL. Las interfaces son descriptas en WSMO en base a:

- ***propiedades no-funcionales***, ***ontologías importadas*** y ***mediadores***, cuyas fundamentaciones son las mismas que las explicadas para sus respectivos usos en la definición de ontologías.

- ***coreografía*** que describen el comportamiento de un servicio desde el punto de vista del servicio cliente. Esta definición están de acuerdo con la brindada por el Glosario del W3C [206], que estipula que la coreografía de servicios web tiene que ver con las interacciones del servicio con sus usuarios o clientes.

- ***orquestación*** que describen cómo el servicio web hace uso de otros para lograr la capacidad descripta. Un ejemplo de orquestación fue presentado en la Sec. 6.1.

WSMO describe las coreografías y orquestaciones según un mecanismo basado en estados, el cual se fundamenta en máquinas de estados finitos [203].

Definición de Metas

Las descripciones WSMO de metas son utilizadas para plasmar los deseos del usuario o aplicación cliente. Son el medio para especificar los objetivos del lado del consumidor de servicios; requisitos que deben cumplirse mediante la ejecución de capacidades de servicios web. Una cuestión importante, que fundamenta esta característica de WSMO, es la búsqueda del completo desacople de las exigencias del consumidor con las funcionalidades que los servicios web deben llevar a cabo para cumplirlas. Los componentes de una descripción WSMO de metas se presentan a continuación.

Propiedades no-funcionales. Debido a que las metas descriptas pueden representar a un servicio que potencialmente satisface determinados requerimientos, el conjunto de propiedades no-funcionales son las mismas que se aplican a las descripciones de servicios web. Un tipo especial de propiedad no-funcional puede ser el *tipo de correspondencia (type of match)*, que refleja el grado de correspondencia exigido entre la meta y las capacidades del los servicios web candidatos.

Ontologías importadas. Una meta puede utilizar ontologías importadas como una terminología para definir otros elementos que son parte de la meta.

Mediadores. La definición de una meta puede utilizar mediadores en las siguientes situaciones:

1. cuando terminología heterogénea es utilizada y se producen conflictos, una definición de metas web puede utilizar un mediador ontológico (Mediador OO), iguales a los explicados en la definición de ontologías; o

2. cuando una meta reutiliza alguna de las definiciones de metas ya descriptas; por ejemplo con el objeto de refinarlas. En este caso se utilizan un tipo especial de mediador, denominado *Mediador GG*, cuya descripción se verá más adelante.

Capacidad requerida. Sirve para indicar las capacidades buscadas en los servicios web.

Interfaz requerida. Sirven para definir las interfaces del servicio web con el que se desea interactuar.

Definición de Mediadores

La mediación se relaciona con manejar la heterogeneidad; es decir, resolver las posibles incompatibilidades entre recursos que se necesitan que interoperen. Por ser ésta, una problemática inherente a sistemas distribuidos, y por ende a las áreas de aplicaciones de los Servidores Web Semánticos, WSMO define el concepto de *mediadores* como conceptos fundamentales.

Las arquitecturas orientadas a la mediación fueron introducidas en [537]. Definen al ***mediador*** como una entidad que establece interoperatividad entre recursos que tiene incompatibilidades antes de resolverlas en tiempo de ejecución. Este enfoque hacia la mediación se sustenta en la descripción declarativa de recursos, con lo cual deben trabajar mecanismos para resolver incompatibilidades en un nivel estructural y semántico. Esto permite definir capacidades de mediación que son genéricas e independientes de dominio; y así también, reutilizables. En lo que respecta a la necesidad de mediación en los Servicios Web Semánticos, WSMO identifica tres niveles de mediación:

- ***Mediación a nivel de datos*** *(Data-level mediation)*: es la mediación entre fuentes de datos o protocolos de transferencias heterogéneos; en el contexto de WSMO este tipo de mediación está estrechamente ligado a la integración de ontologías (véase Sec. 7.4.1)

- ***Mediación a nivel funcional*** *(Functional-level mediation)*: es la mediación entre funcionalidades requeridas y provistas. En WSMO, se relaciona con la vinculación de descripciones de metas y servicios web que son similares pero no exactas.

- ***Mediación a nivel de proceso*** *(Process-level mediation)*: es la mediación para heterogeneidad entre procesos de negocios o protocolos de comunicación. En WSMO, este tipo de mediación trata los escenarios de coreografía y orquestación de servicios.

Los mediadores en WSMO permiten el logro de una arquitectura orientada a mediación para Servicios Web Semánticos. Una definición de mediador WSMO tiene los siguientes componentes.

Propiedades no-funcionales. Un mediador puede ser visto como un servicio en sí; por ende, se aplican todas las propiedades no-funcionales permitidas para descripciones de servicios web.

Ontologías importadas. La fundamientación del modo de uso es la misma que explica su uso en las definiciones de ontologías.

Origen (Source). Sirve para definir el recurso para el cual se resuelve la heterogeneidad. Un mediador puede tener varios componentes `origen`.

Destino (Target). Sirve para definir el recurso que recibe al recurso origen mediado; es decir, al recurso origen transformado para salvar la heterogeneidad.

Servicio mediador. Especifica el servicio mediador con la capacidad que se aplica para resolver la inconsistencia. Puede ser definido de diferentes formas: en forma directa por una definición explicita del vínculo a un servicio de mediación; mediante una meta que especifica la capacidad de mediación requerida y es detectada mediante un proceso de descubrimiento de servicio; y mediante otro mediador, cuando el servicio de mediación no es interoperable con el mediador que se describe.

Como pueden darse problemas de heterogeneidad entre varios tipos de componentes WSMO, se definen varios tipos de mediadores para conectar tipos específicos de recursos: *mediadores OO*, *mediadores GG*, *mediadores WG* y *mediadores WW*. Las siglas en mayúsculas indican los componentes que vinculan.

- ***Mediadores OO*** *(OO Mediators)* resuelven las incompatibilidades entre ontologías y proveen especificaciones relacionadas con el dominio al componente destino de la mediación. Los mediadores OO son utilizados para importar la terminología requerida para la descripción de un recurso en cualquier situación que exista inconsistencia entre las ontologías a utilizar. Las técnicas de mediación OO son las comúnmente usadas para integrar ontologías (véase Sec. 7.4.3).

- ***Mediadores GG*** *(GG Mediators)* conectan definiciones de metas para poder crear nuevas metas a partir de las existentes. Eventualmente un mediador GG puede necesitar usar un mediador OO para resolver incompatibilidades ontológicas en su definición.

- ***Mediadores WG*** *(WG Mediators)* vinculan una definición de un servicio web con una meta, resuelve las incompatibilidades terminológicas y establecen las diferencias entre ambos (si existen). Están destinados a manejar heterogeneidad de términos y sirven para vincular de antemano servicios web con metas definidas, y para el manejo de correspondencias parciales en el descubrimiento de servicios.

- ***Mediadores WW*** *(WW Mediators)* establecen interoperatividad entre servicios web que de otra forma no son interoperables. Un mediador WW se establece entre los servicios web que conforman coreografías, con el propósito de mediar sobre datos, protocolos o procesos.

Lenguaje de Modelamiento de Servicios Web WSML

El ***Lenguaje de Modelamiento de Servicios Web*** o ***WSML*** *(Web Service Modeling Language)* [153] es un lenguaje formal para la descripción de ontologías y servicios web semánticos que tiene en cuenta todos los aspectos de las descripciones semánticas identificadas por WSMO [129]. El lenguaje WSML contiene varios formalismos, basados en la Lógica Descriptiva y en la Programación Lógica, que lo hacen aplicable en Servicios Web Semánticos. Dichos formalismos están restringidos, y en ocasiones no están presentes en otros lenguajes ontológicos como OWL [513, 130].

El lenguaje WSML está en desarrollo, todavía no se lo puede considerar una tecnología madura, como puede ser OWL. Como requerimiento de diseño WSML busca establecer métodos formales en tres principales áreas: (1) para la descripción de ontologías, (2) para la descripción declarativa funcional de servicios web y de metas y (3) para definición de dinámica de servicios web.

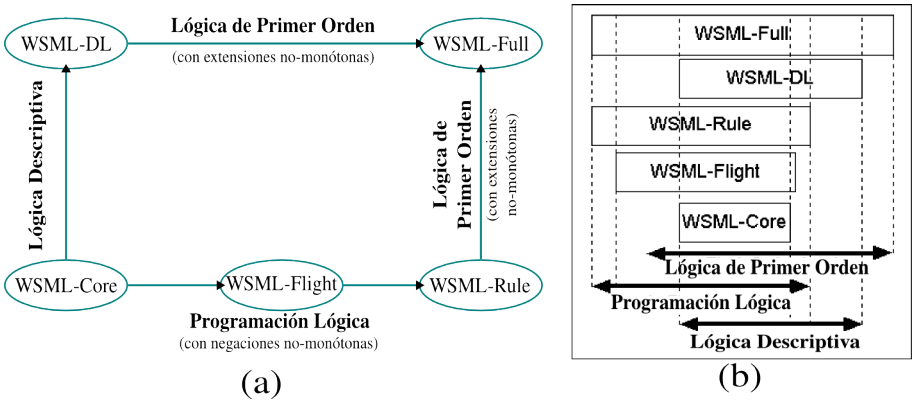

Figura 7.9: (a) Relaciones entre variantes de WSML. (b) Relaciones de varientes WSML con formalismos.

La versión actual de WSDL define la sintaxis y semántica para la definición de ontología. En la Versión 1.0 propuesta en agosto de 2008 [541] se admite la importación de ontologías definidas en RDF(S) y OWL. La versión 1.0 propone cambios conceptuales respecto a la versión anterior 0.21, a tal punto que presenta incompatibilidades. Con respecto a la definición de descripciones funcionales para metas y servicios web, simplemente se propone un marco de trabajo, sin estipular ninguna semántica en particular. Por su parte, la descripción del comportamiento dinámico de servicios web está bajo plena investigación y no conforma parte integral de WSML.

WSML al igual que OWL, tiene diferentes modos que encierran diferentes niveles de expresividad, los cuales están asociados a diferentes paradigmas descriptivos/lógicos. Las variantes de WSML son las siguientes:

- ***WSML-Core*** que está basado en la intersección de dos formalismos: la Lógica Descriptiva *SHIQ* [25] y la Lógica de Horn [202], denominada *DLP (Description Logic Programs)* [194]. Es la variante de WSML que tiene menor poder expresivo, la cual se limita a la descripción de conceptos, atributos, relaciones binarias e instancias. Soporta la jerarquía de conceptos y relaciones, y también tipo de datos.
- ***WSML-DL*** que captura la Lógica Descriptiva *SHIQ(D)*, que en su mayor parte se corresponde con OWL-DL [130].
- ***WSML-Flight*** es una extensión de WSML-Core con un poderoso lenguaje de reglas. Agrega características como metamodelado, restricciones y negación no-monótona. Está basado en una variación de F-Logic [250].
- ***WSML-Rule*** extiende a WSML-Flight con más características de Programación Lógica. Agrega semántica de buena formación [512].
- ***WSML-Full*** unifica WSML-DL y WSML-Rule en el ámbito de Lógica de Primer Orden con extensiones de razonamiento no-monótono. La semántica de WSDL-Full está en proceso de desarrollo actualmente.

En la Fig. 7.9 se puede ver (a) las relaciones entre las diferentes variantes y (b) la relaciones de las variantes con el formalismo que adoptan.

7.5.3. Otros Marcos de Trabajos para Servicios Web Semánticos

OWL-S

OWL-S [86, 280] es un ontología de servicios descripta en OWL que sirve para especificar varios aspectos de los servicios web. La iniciativa tiene sus raíces en la Ontología de Servicios

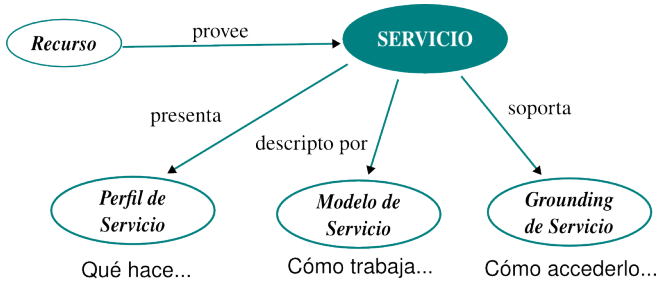

Figura 7.10: Esquema de Modelo Conceptual de OWL-S.

DAML (DAML-S) [117]. Al mismo tiempo fue el primer esfuerzo progresivo para que los servicios web adquieran anotación semántica. Gracias a que migró desde DAML+OIL [174] hacia OWL, OWL-S trata de adoptar las recomendaciones existentes relacionadas con la Web Semántica. La ontología en sí, define el concepto fundamental de *Servicio* y tres OWL-S subontologías conocidas como el *"Perfil de Servicio"*, el *"Modelo de Servicio"* y el *"Grounding*[3] *de Servicio"* (Véase Fig. 7.10).

Perfil de Servicio. Cada instancia de la clase *Servicio* puede presentar varios *perfiles de servicios*. Un ***perfil de servicio*** *(service profile)* expresa lo que hace el servicios, para propósitos de promoción de sus capacidades. Un perfil de servicio también sirve como plantilla para peticiones al servicios; de esta forma permite el descubrimiento y la correspondencia con requerimientos de clientes. El perfil de servicio puede estar conformado por aspectos no-funcionales como referencias a esquemas de clasificación estandarizados (como UNSPC [506]) u ontologías externas, información respecto al proveedor y dueño, tasa de disponibilidad de servicio, etc.

Sin embargo, la información más importante descripta en el perfil es la especificación sobre la funcionalidad que provee el servicio. Dicha descripción se realiza en base a: *entradas*, *salidas*, *precondiciones* y *efectos*. Las ***entradas*** y ***salidas*** se refieren a clases OWL que describen los tipos de las instancias de la información a ser enviada al servicio y su correspondiente respuesta. Las ***precondiciones*** y ***efectos*** no tiene un formato fijo. Un problema al respecto es que las semánticas de dichas condiciones no son cubiertas por la expresividad de OWL-S, basada en Lógica Descriptiva. Para cubrir esta falencia OWL-S admite varios esquemas para importar en su ontologías OWL. Preferentemente se utiliza expresiones en SWRL [223], y como segundas opciones, en DRS [285] y KIF [187]. Por ende, cuando se intercambian descripciones de perfiles en OWL-S o se utiliza OWL-S para el descubrimiento, se necesita consensuar el lenguaje para expresar las condiciones y la noción de correspondencia; ya que tales características no son cubiertas por el estándar ontologóco OWL.

Modelo de Servicio. Un servicio web debe ser descripto por un ***modelo de servicio*** *(service model)* que exprese cómo trabaja. El principal uso de un modelo de servicio es permitir la invocación, promulgación, composición, monitoreo y recuperación del servicio. El modelo de servicio entiende a las interacciones con el servicio web como un proceso; que antes de asociarse con un programa a ejecutar, el concepto debe entenderse como las formas en las que un cliente consumidor puede interactuar con el servicio. OWL-S distingue entre *procesos atómicos*, *procesos simples* y *procesos compuestos*. Los ***procesos atómicos*** son operaciones simples; mientras que los ***procesos simples*** son vistos como procesos atómicos o complejos que preservan encapsulamiento y abstracción. Los ***procesos compuestos*** son construidos a partir de procesos simples o atómicos mediante constructores estándares de flujo de trabajo, como `sequence`, `split` o `join` para describir la información del flujo de control y del flujo de datos del proceso.

[3]Se preserva el término en inglés por no encontrar una adecuada traducción.

El resultado y las salidas depende de una condición expresada en lenguajes lógicos externos. De nuevo, un posible problema, además de las condiciones, es que la semántica de los constructores de flujo de trabajo no puede ser expresada en la Lógica Descriptiva subyacente en OWL; por esta razón, dicha semántica debe ser externamente definida [323].

Grounding de Servicio. Al hablarse de *grounding*, se hace referencia a la cualidad de vincular las definiciones semánticas con el o los componentes que implementan dicho conceptos. De esta forma, definir el ***Grounding de Servicio*** *(Service Grounding)* implica establecer mapeos entre los constructores del modelo de proceso de OWL-S con las especificaciones detalladas de formato de mensajes, protocolos y similares. OWL-S permite establecer correspondencia entre procesos atómicos con operaciones WSDL, y sus entradas y salidas con los mensajes descriptos WSDL; sin embargo no se limita a esa única tecnología subyacente.

Comparación con WSMO. Una de las diferencias que citan los autores de WSMO [162, 268], es el meta-lenguaje para definir los conceptos fundamentales de cada marco de trabajo. Por un lado, OWL-S de describe en base al mismo OWL; en tanto que WSMO se describe utilizando el meta-lenguaje *MOF (Meta-Object Faclitiy* [355]. Es decir, OWL-S define sus meta-modelos en el mismo lenguaje que utiliza para definir la descripciones concretas de servicios; por otro lado, WSMO está diseñado en base a un lenguaje específico para definir lenguajes ontológicos.

Cuando se trata de precondiciones o suposiciones, OWL-S delega la descripción de las mismas a otros lenguajes, si especificar el modo de combinación entre varios formalismos. Por su parte, WSMO describe todo componente (ontología, mediador, servicio web, meta) y condición en un mismo lenguaje, WSML. Si bien OWL-S orienta en la elección del lenguaje para describir suposiciones y condiciones, todas las sugerencias están basadas en Lógica Proposicional o de Primer Orden. Por el contrario y como se explicó, WSML admite el uso de características no-monótonas provenientes de la Programación Lógica.

Tanto OWL-S como WSMO tiene similitudes en su modelo conceptual. Un perfil de servicio de OWL-S es similar a la capacidad de un servicio web o meta de WSMO. Sin embargo, WSMO hace explícita la diferencia entre la vista del proveedor de servicio (*capacidad* en la definición de servicio) y los requerimientos del consumidor (*capacidad requerida* en la definición de una meta). El modelo de proceso de OWL-S es similar a las interfaces de servicios web o metas de WSMO. Sin embargo en WSMO también existe una distinción que lo destaca: la posibilidad de tratamiento diferente del comportamiento externo (interfaz de coreografía) y del comportamiento interno (interfaz de orquestación).

Los mediadores de WSMO no tienen una contraparte específica en OWL-S. OWL-S los trata como un tipo especial de servicio [375]. Es único el tratamiento que le da WSMO como componentes fundamentales orientados a resolver heterogeneidades.

Por otro lado, los elementos de *grounding* que OWL-S los considera fundamentales en su modelo conceptual, no recibien la misma jerarquía en WSMO; aunque la vinculación que denota el *grounding* se especifica en las interfaces WSMO. Al respecto también se puede decir, que ni OWS-L ni WSMO están limitados al tipo de tecnología subyacente para la descripción de servicios web, como puede ser WSDL.

SWSF

El ***Marco de Servicios Web Semánticos*** o ***SWSF*** *(Semantic Web Services Framework)* [36] es un intento reciente de marco de trabajo para anotaciones semánticas en servicios web que se nutre de los trabajos previos en OWL-S y el *Lenguaje de Especificación de Proceso* o *PSL (Process Specication Language)*, estandarizado por la norma ISO 18269 [199]. SWSF está basado en dos principales componentes: una *ontología* o *modelo conceptual* y un *lenguaje* para axiomatizarla.

Modelo conceptual. La ***ontología SWSO*** *(Semantic Web Services Ontology)* ha sido influenciada por OWL-S y comparte sus tres conceptos: *perfil*, *modelo* y *grounding*. De esta forma, SWSO puede verse como un refinamiento de OWL-S. A pesar que existen similitudes con la ontologías OWL-S, una diferencia sustancial es la expresividad del lenguaje subyacente; el cual, en vez de ser OWL, es *SWSL*.

Lenguaje. El lenguaje ***SWSL*** *(Semantic Web Service Language)* tiene como objetivo describir los conceptos de servicios web y cada servicios web en particular. SWSL viene en dos variantes: *SWSL-FOL* y *SWSL-Rules*. El diseño de ambas variantes buscó compatibilidad con las tecnologías Web como ser el uso de URIs, integración con los tipos predefinidos de XML, uso de espacios de nombres y los mecanismos de importación de los mismos. Ambos lenguajes están conformados por capas y con cada capa se incluyen nuevos conceptos que mejoran la expresividad del lenguaje. Esto hace que ambos lenguajes sean más expresivos que OWL-S.

En el enfoque adoptado por el marco de trabajo SWSF, se dan dos conceptualizaciones (ontologías) independientes para el modelo conceptual; ambas son expresadas como variantes del lenguaje SWSL: (1) la ***ontología FLOWS*** *(First-order Logic Ontology for Web Services)* basada en el lenguaje SWSL-FOL, y (2) la ***ontología ROWS*** *(Rule Ontology for Web Services)* basada en SWSL-Rules; que, si bien comparte el mismo modelo conceptual, provee una semántica diferente.

La ontología FLOWS, con sustento en la Lógica de Primer Orden, hace uso de los predicados lógicos para modelar el estado del mundo. Han sido introducidas características propias del Cálculo de Predicados [407] como el uso de predicados fluentes, para representar cambios en el mundo, de manera compatible con PSL [36, 197]. El modelo de proceso es parte de la ontología FLOWS la cual ofrece constructores para describir el comportamiento de servicios, según el enfoque de PSL. Por su parte SWSL-Rules es un lenguaje de Programación Lógica, y provee soporte para las tareas relacionadas con los servicios como ser descubrimiento, vinculación y especificación de políticas.

Comparación con WSMO. Como modelo conceptual, SWSF comparte las mismas características de OWL-S, por ende son similares las diferencias con WSMO. Sin embargo, algunas limitaciones de OWL-S fueron solucionadas en SWSF. SWSF por ejemplo, no se limita a la expresividad a la Lógica Descriptiva y provee definiciones robustas respecto a la semántica de condiciones y a los aspectos dinámicos de los servicios web mediante la reutilización de PSL. De esta forma, al igual que WSML se constituye en un único lenguaje para el marco de trabajo.

WSDL-S

En comparación con WSMO, OWL-S y SWSF, el lenguaje ***WSDL-S*** [7] es el enfoque más simple, ya que se constituye como una extensión directa de la especificación WSDL (véase Sec. 5.7) a la cual le agrega manejo de semántica.

La especificación WSDL-S tiene sus orígenes en el proyecto METEOR-S [379, 516]. En 2005 WSDL-S ha sido puesto en consideración para su tratamiento en el W3C [8]. La propuesta de WSDL-S consiste en un mecanismo liviano para incrementar las definiciones WSDL con semántica, mediante el agregado de etiquetas con anotaciones al esquema de WSDL. El objetivo del uso de estas etiquetas es poder describir la semántica que se captura con las definiciones de capacidades de WSMO o los perfiles de OWL-S.

Haciendo uso de la extensibilidad de los elementos WSDL, se crearon un conjunto de anotaciones que describen semánticamente las entradas, salidas y operaciones de un servicio web. A la vez, estas anotaciones WSDL-S sirven para categorizar a un servicio en referencia a una ontología externa. Esta metodología mantiene al modelo semántico por fuera de WSDL, haciendo este enfoque imparcial en relación al lenguaje utilizado para representar dicha ontología. Se han

propuestos los lenguajes WSML, OWL, y UML como posibles candidatos para cumplir el rol de descriptor ontológico.

Las principales características de diseño de WSDL-S son:

- WSDL-S se construye sobre un estándar industrial como WSDL y promueve un mecanismo compatible para agregar semántica a los servicios web.
- Las anotaciones WSDL-S son abstractas respecto al lenguaje ontológico, dando libertad de acción para elegir.
- Las anotaciones WSDL-S son definidas en base a tipos de XML Schema, siendo éste el más importante formato de definición de tipos de datos en el mundo de Servicios Web y Web Semántica.

El último item está orientado principalmente a estrechar el vínculo entre las definiciones de mensajes WSDL en términos de XML Schema y sus contrapartes semánticos. En propuestas como WSMO y OWL-S esto se logra en forma desacopalda, a través de la definición de componentes de *grounding*. WSDL-S es esencialmente diferente en este aspecto, por establecer directamente en la definición WSDL las anotaciones semánticas. Este hecho puede considerarse como una desventaja respecto a los otros formalismos, ya que fija implícitamente la tecnología subyacente de servicio web a definiciones WSDL.

Comparación con WSMO. WSDL-S con su enfoque minimalista puede verse como ortogonal a WSMO. Se pueden usar anotaciones WSML, o incluso recursos definidos en WSMO, para definir condiciones o suposiciones en WSDL-S. WSDL-S no se establece como un completo marco de trabajo para ladefinición de servicios web semánticos como intenta serlo WSMO.

Una de las críticas profundas a WSDL-S es su imparcialidad respecto al lenguaje a utilizar para definir las anotaciones semánticas y ontológicas. Sin lograr un cierto acuerdo respecto al uso de un determinado lenguaje ontológico para tal fin; o estipular al menos cómo pueden relacionarse las diferentes semánticas, es imposible que se establezcan mecanismos de consulta, vinculación, o noción respecto a la correspondencia entre servicio requerido y las descripciones de servicios actuales.

Los principios propuestos por los Servicios Web Semánticos, tiene inherentemente la idea de reuso presente. La idea de automatizar las tareas relacionadas con el uso de servicios web mediante descripciones semánticas tiene como objetivo implícito automatizar el reuso de servicios. Los formalismos y lenguajes para describir a lo servicios web como conceptos ontológicos para posibilitar el razonamiento automático sobre los mismos varían en expresividad y performance. La mayoría de estos lenguajes tiene sus raíces en aquellos orientados a ontologías de datos de Web Semántica. Se puede ver que el marco de trabajo propuesto por WSMO es una de las ideas más evolucionadas; sin embargo, puede considerarse un campo de investigación con final abierto.

7.6. Resumen

En este capítulo se describieron las principales vinculaciones que tiene la Reusabilidad de Software con los Servicios Web. Se pudo ver que la problemática de reusabilidad no depende exclusivamente de la adopción de una determinada tecnología informática. Existen impedimentos al reuso que no tienen raíz tecnológica. Para concretar un escenario de reuso sistémico, es necesario que determinadas prácticas de diseño y producción sean revisadas para establecer la Reusabilidad de Software como un principio y una actividad con significación e importancia propia dentro de la estructura de cualquier empresa.

En tal sentido, adoptar los principios de las Arquitecturas Orientadas a Servicios como marco para la práctica de la informática es una decisión significativamente en favor de la Reusabilidad.

El paradigma SOA establece como principio fundamental a la Reusabilidad de Servicios, considerándolo medio y fin último en la producción de software. En la creación de cada servicio, se acepta la idea de servicio agnóstico, como cualidad intrínseca para asegurar sus posibilidades de reutilización futura; por otro lado, también se estipula el mantenimiento de un Inventario Centralizado de Servicios para tener a disposición componentes reusables que puedan aportar parte de la lógica de aplicación subyacente.

Luego el capítulo trato el uso de Servicios Web para la modernización de aplicaciones legadas. Entendiendo la problemática de los sistemas legados, los cuales encierran un conocimiento esencial para determinados procesos de negocios, la adopción de los Servicios Web para reciclarlos es una de las variantes de modernización menos riesgosas. Se presentaron también lineamientos metodológicos para llevar a cabo la tarea de conversión de componentes legados en servicios web expuestos mediante envolturas.

Otras de las vinculaciones tratadas fue la reutilización de ontologías mediante tecnologías de Web Semántica. El paradigma de la Web Semántica tiene como objetivo hacer que los recursos de la Web sean conceptualizados semánticamente. Dicha conceptualización tiene su adecuada realización en las ontologías. Se pudieron desarrollar diferentes clasificaciones de ontologías teniendo en cuenta ejes de expresividad y genericidad. Se explicó la noción de lenguaje ontológico y se describieron dos ejemplos de estándares para definición de ontologías en la Web Semántica: RDF(S) y OWL. Se concluyó el estudio de reuso de ontologías con una breve descripción de metodologías para la construcción de ontologías que tiene a la reutilización y el reuso como actividades normadas.

Para finalizar el capítulo, se estudio en reuso en los Servicios Web Semánticos. Se presentó la idea de Servicio Web Semántico como fusión de los principios de Servicios Web y Web Semántica. La principal motivación es poder conceptualizar a los servicios web mediante ontologías que expliquen la semántica de sus capacidades e interfaces y así poder automatizar las tareas de descubrimiento, negociación, composición e invocación. Se presentó un marco de trabajo orientado a Servicios Web Semánticos: WSMO, para poder explicar las características semánticas relacionadas a servicios web. También se hizo una breve descripción de otras alternativas en comparación con WSMO, concluyendo que es un área en desarrollo y no existen estándares impuestos.

Capítulo 8

Conclusiones

8.1. Conclusiones

La realización del presente trabajo de Tesis ha permitido al autor cumplir con los objetivos planteados. Al respecto se pueden enunciar algunas conclusiones a partir de los mismos:

Comprender la problemática de la reusabilidad y sus efectos concretos en el desarrollo de software.

Respecto a la problemática de reusabilidad se debe entender que su definición, ámbito y efectos fueron adaptándose a lo largo del tiempo. Las exigencias actuales respecto a la reusabilidad distan bastante de sus orígenes en dónde simplemente se planteaba la necesidad de conformar librerías de componentes o módulos reutilizables. En las últimas décadas la necesidad de lograr reuso es cada vez más grande, no sólo para dotar a los desarrolladores de un grupo de componentes reutilizables. Las nuevas exigencias abarcan diferentes situaciones, como por ejemplo:

- la modernización y reuso de aplicaciones legadas, por considerarlas bienes esenciales en una empresa y no existir otra forma viable de modernizarlas,
- la integración de aplicaciones, que impone situaciones en donde el reuso a través de plataformas de middleware es la única vía posible, porque las partes a integrar son gerenciadas por diferentes entidades y no es posible ningún tipo de rediseño o acuerdo en la elección de plataformas de desarrollo,
- el reuso de activos de intelectuales, como ontologías que describen conocimiento de un determinado dominio, en donde se plantea la interrelación de bases de conocimiento de diferente índole y con diferente naturaleza y composicion de recursos,
- la automatización del reuso, a partir de definiciones semánticas de cada artefacto reutilizable y la implementación de mecanismos de descripción, descubrimiento, composición e invocación.

Es concluyente decir que cada uno de estos escenarios nuevos aumentan la definición de Reusabilidad de Software, esencialmente porque se ha incrementado la definición de software. La aparición de la Web y su posterior adopción como medio de computación y gestión de conocimiento llevo la concepción de computación, software, aplicación, sistema, dato, información, recurso y distribución a niveles nunca pensados en los inicios de la Ingeniería en Software. Es lógico concluir que la idea de Reusabilidad de Software, también haya sido renovada.

Sin embargo, a pesar que la problemática del reuso aumentó en complejidad, se ven requerimientos más concretos y desafíos más motivadores a vencer. Algunas soluciones planteadas, como la integración de ontologías, logran instaurar los procesos de reutilización como un efecto

colateral a la integración. Otras soluciones, buscan al reuso como requerimiento esencial, como ser la modernización de aplicaciones legadas.

Comprender la problemática de la integración de aplicaciones distribuidas como escenario elegido para visualizar las necesidades de reuso de software.

En una forma simple de verlo, se puede concluir que la integración de aplicaciones distribuidas le dió un rosto al problema de reusabilidad, un lugar por donde atacar la carencia de ésta. Los desafíos planteados por la integración de aplicaciones en sistemas distribuidos obligaron a prácticar el reuso y la reutiliación.

La práctica del reuso está asociaba fuertemente con la conformación de una estructura empresarial humana y tecnológica que lo soporte, con la buena praxis de los desarrolladores promoviendo el reuso y la reutiliación en cada proyecto, y con la financiación y apoyo gerencial que se exige para establecer tal actividad organizativa. Como en los inicios de la Ingeniería en Software, una empresa dependía de sí misma para lograr tener una infraestructura informática, el apoyo al reuso sistémico (no trivial) era visto como una pérdida de tiempo y recursos. Esto se producía porque los resultados de un plan de reuso, que es costoso de implementar, se verían en un futuro a mediano y largo plazo. A estos impedimentos políticos/culturales, se le debían agregar la falta de maduración que tenian los principios y herramientas tendientes a favorecer la construcción para reuso y con reusabilidad. Esencialmente no hay diferencia en el aspecto de una aplicación desarrollada desde cero y otra que reutilizó componentes; los beneficios del reuso se ven en el proceso de desarrollo y en la calidad final del software. Es por eso que en ocasiones, por no obtener resultados concretos inmediatamente, se desestimaban o abortaban las iniciativas de reuso.

Los requerimientos de integración de aplicaciones precipitaron los tiempos, y surgió la real necesidad de tener en cuenta a la reusabilidad en el desarrollo de aplicaciones. Sencillamente porque un equipo de desarrollo no tenía control sobre los componentes distribuidos a integrar, y mucho menos sobre las decisiones respecto a las herramientas de desarrollo y plataforma de implementación. Surgen así la necesidad de establecer estándares de interoperatividad y respetarlos. De exponer los componentes heterogénesos de software con interfaces abiertas que se vinculen a través de redes intraempresariales o incluso de Internet.

La integración de aplicaciones presentó situaciones en donde el reuso era la única opción viable, conformar procesos de negocios con soporte de flujos de trabajos es esencialmente una tarea de reutilización. En ocasiones estos nuevos retos, originó que las organizaciones entren en planes de reorganización de sus activos informáticos, y recursos humanos calificados. El hecho de que las operaciones comerciales electrónicas se hicieran cada vez más recurrentes, obligó a las empresas y su grupo de desarrollo optar por tecnologías abiertas y métodos de producción de software que generen aplicaciones con alto grado de adaptabilidad, extensibilidad, interoperatividad y posibilidades de reuso en esenarios no previstos.

Comprender los principios sostenidos por las tecnologías de los Servicios Web y los efectos de su adopción en la construcción de sistemas informáticos distribuidos.

La tecnología de Servicios Web es, en su concepción, una evolución en las tecnologías de middleware que tiene como propósito fundamental solucionar las barreras de heterogeneidad en la integración y desarrollo de aplicaciones distribuidas.

Sin embargo, la noción de Arquitecturas Orientadas a Servicios, que es el marco teórico de Servicios Web, expresamente habla de la Reusabilidad como principio rector. Se puede concluir que la tecnología de Servicios Web surge para solucionar problemas que las tecnologías anteriores de middleware no solucionaron. Sin embargo la metodología de desarrollo de software que posibilita su adopción, la Orientación a Servicios, establece al reuso como uno de sus condicionantes

para su realización efectiva. Con esta distinción entre tecnología y método, se deja en claro que la solución a la reusabilidad no es una cuestión de adopción tecnológica, sino una práctica que debe involucrar a todos los actuadores de una empresa para poder concretar los beneficios y ventajas competitiva que viene con ésta. Sin embargo, es concluyente que los Servicios Web, han provisto el marco tecnológico adecuado para hacer realidad los principios del reuso; a diferencia del relativo o poco éxito de las tecnologías previas.

Se puede concluir que los Servicios Web son un marco tecnológico que soporta la reutilización (construir software a partir de software existente) y el reuso (construir software con posibilidades de reutilizarse). Y complementariamente, la Orientación a Servicios es el marco teórico y metodológico que hace posible la reusabilidad sistémica.

Relacionar los conceptos anteriores, describiendo escenarios concretos de vinculación, comprendiendo las problemáticas particulares de Reusabilidad e Integración y su solución a través de la tecnologías de Servicios Web.

Han sido exitosos la utilización de los Servicios Web en escenarios concretos que requieren integración o reuso. Si bien toda solución informática requiere que los ingenieros tomen decisiones, y la calidad de esas decisiones puede variar de acuerdo a la idoneidad y preferencias particulares de los expertos, el uso de los Servicios Web es adaptable a un amplio aspecto dando posibilidades concretas de éxito.

La utilización de Servicios Web favoreció el reciclado de aplicaciones legadas; evitando de esta forma el riesgo y costo que exigiría reemplazar una aplicación legada por una completamente nueva. El uso de las envolturas de servicios web para la modernización de aplicaciones, aumenta el tiempo de vida de los sistemas legados, preserva su lógica de aplicación depurada durante años y le da la posibilidad de extender sus capacidades. Este escenario de uso de Servicios Web permite integrar componentes legados a una Arquitectura Orientada a Servicios.

Otro escenario que se consideró es la Web Semántica, como evolución de la Web actual. La principal motivación de la Web Semántica es la integración y la gestión automatizada del conocimiento contenido de la Web. La Web Semántica propone describir formalmente dichos recursos mediante el uso de lenguajes y marcos de trabajos ontológicos. Esencialmente una ontología describe sin ambigüedades, y con reutilización de términos, cualquier concepto de la Web. Una ontología no tiene por qué ser totalmente autodescriptiva, puede integrarse con otras ontologías para incrementar su base de términos.

El escenario final tratado, los Servicios Web Semánticos, es la fusión de la tecnología madura de los Servicios Web con el reciente desarrollo de la Web Semántica. Se plantea describir semánticamente a los servicios web como artefactos de software para automatizar las tareas asociadas; es decir, el descubrimiento, la descripción, la negociación, la composición y la invocación. Se puede concluir que los Servicios Web Semánticos son un medio para automatizar el reuso de software, y es lógico concluir que serán el nuevo paso en el desarrollo de las tecnologías de integración y un campo de desarrollo e investigación para la Reusabilidad de Software.

Por lo descripto en cada punto de conclusión se puede decir en forma de resumen que:

- Existe una vinculación natural entre la Reusabilidad y los Servicios Web, especialmente potenciada por la integración de aplicaciones, de datos y de conocimiento.
- Los Servicios Web y la Orientación a Servicios son fuertes paliativos para la falta de reusabilidad. Sin embargo la adopción de cada uno requiere un gran esfuerzo, que puede involucrar a transformaciones significativa en la infraestructura informática de una empresa.
- No se puede llegar a una conclusión definitiva respecto a que si los Servicios Web y la Orientación a Servicio son la "cura" a la falta de reusabilidad. Por un lado, es necesario que determinadas tecnologías maduren, como los Servicios Web Semánticos, para valorar su

éxito en la automatización del reuso. Por otro lado, no se pude hablar de "cura" definitiva, ya que la complejidad de la reusabilidad, como se explicó, va generando nuevos requisitos y escenarios.

- Los Servicios Web, y en especial su aporte a la integración e interoperatividad, favorecieron a brindar requerimientos realistas y soluciones viables a cuestiones sobre reuso. Ambos conceptos presentan una sinergia conjunta. A la luz de lo investigado, los Servicios Web y la Orientación a Servicios resultan tener altas probabilidades de éxito para la adopción y práctica de la Reusabilidad de Software.

8.2. Resultados Logrados

El autor del presente trabajo, además de reconocer el aprendizaje y asimilación de los muchos y variados conocimientos generados por el proceso investigación y redacción de esta Tesis, quiere destacar también algunos resultados visibles:

- Presentación de un artículo denominado *"Middleware support for semantic web services: a survey of current approaches"*, en co-autoria con el Director de la Tesis Dr. Pablo Fillottrani, a los efectos de ser publicado en el Congreso de Inteligencia Computacional Aplicada (CICA) 2009 que se llevará a cabo el 23 y 24 de julio de 2009, en la ciudad de Buenos Aires.
- Por el carácter de docente de la Facultad de Ciencias de la Administración de la Universidad Nacional de Entre Ríos, el autor de la Tesis ha dictado y presentó propuestas de cursos de créditos válidos para la carrera Lic. en Sistemas que se dicta en dicha Facultad, sobre temáticas desarrolladas en la tesis:
  - *"Fundamentos de Programación Web"* - Dictado en el año 2004
  - *"Fundamentos para Desarrollo en Web"* - Propuesta presentada y aprobado su dictado para el año 2009
  - *"Tecnologías de Middleware"* - Propuesta presentada y aprobado su dictado para el año 2009
- El autor de la Tesis ha dictado y tiene planificado dictar las siguientes charlas relacionadas de las temáticas desarrolladas en la tesis:
  - *"Principios de Reusabilidad de Software"* - III Jornadas de Administración e Informática. Fac. de Ciencias de la Administración. Universidad Nacional de Entre Ríos. 29 de Octubre de 2004.
  - *"Servicios Web para la integración de Aplicaciones"* - Fac. de Ciencias de la Administración. Universidad Nacional de Entre Ríos. Propuesta aceptada para dictarse en Abril 2009.
- En relación a las herramientas utilizadas para la redacción de la presente Tesis, el autor dictó el curso *"Tecnología Educativa y Elaboración de Materiales Didácticos para la Enseñanza de la Matemática"*. En el cual se trató la utilización de LaTeXy PowerPoint para la contrucción de material didáctico. El curso fue dictado en la Licenciatura en Enseñanza de la Matemática en el marco del Convenio: Universidad Nacional de Lomas de Zamora - Fac. de Ingeniería y Universidad Nacional de Entre Ríos - Fac. de Ciencias de la Administración.

8.3. Futuras Líneas de Investigación

Con la concreción de cada capítulo, se le han planteado posibles caminos para proseguir el proceso de investigación. A continuación se describen, a criterio del autor de la Tesis, las temáticas que merecen mayor profundización en investigaciones futuras.

Sistemas Informáticos Distribuidos

En el contexto de los diseños de aplicaciones distribuidas se propone profundizar la técnica *middle-out*. Esta técnica de diseño es utilizada en la construcción de envolturas, como una combinación de *top-down* y *bottom-up*.

También se sugiere realizar un estudio comparativo de las técnicas y productos de integración que implementen el método *bottom-up*. Especialmente aquellas que soporten la integración de servicios web de forma automatizada.

Middleware

Se propone continuar con el estudio exploratorio de las tecnologías que dan soporte a la Computación Ubicua. En especial aquellas que se basen en el paradigma de agente inteligente, ya que muestran ser las variantes válidas de acuerdo a los principios de la Web Semántica.

Otra posible temática a profundizar es la Brokers de Agentes, como infraestructura de middleware que brinda un ambiente de ejecución de Servicios Web Semánticos.

Tecnologías Web

Sería adecuado continuar con el estudio exploratorio de las plataformas de desarrollo de aplicaciones distribuidas en la web. En especial estudiar las cuestiones fundamentales para la configuración y optimización de Servidores de Aplicaciones.

Tecnologías Fundamentales de Servicios Web

Se recomienda hacer un estudio comparativo entre las tecnologías de Servicios Web fundamentales presentadas en esta tesis (XML, WSDL, SOAP. UDDI), con otras alternativas. En especial con el marco de trabajo REST (Representational State Transfer).

Se propone profundizar el estudio de las extensiones de las tecnologías bases (XML, WSDL, SOAP. UDDI). Como puede ser aquellas relacionada de XML al manejo de datos multimediales y se seguridad. También se recomienda el estudio de estándares orientados a la correcta utilización de las especificaciones, como el Perfil Básico (Basic Profile) propuesto por la Organización WS-I (Web Services Interoperability Organization).

Segunda Generación de Tecnologías Web

Se sugiere profundizar en las especificaciones no cubiertas en este trabajo. En especial, realizar un estudio más pormenorizado de los estándares WS-* relacionados con seguridad, como ser WS-Federation, WS-SecureConversation, WS-Trust, XML Encryption, XML Signature, XKMS, SAML y otros.

Se recomiendo hacer un estudio más detallado del modelo coreográfico propuesto por WS-CDL. En especial relevar los campos de aplicación y realizar una evaluación comparativa con otras tecnologías de composición como ser WS-BPEL.

Servicios Web y Reusabilidad

Se propone profundizar sobre la adopción y aplicación del principio de Reusabilidad en arquitecturas SOA. En especial realizar una recopilación de casos de éxito, de la cual se puedan extraer técnicas y procedimientos exitosos.

Otra línea de investigación a seguir se relaciona con el estudio de metodologías para utilizar ontologías para integrar bases de datos legadas a través de la Web. En especial los formalismos y métodos utilizados para acceder a consultas de base de datos o información de minería de datos a través de recursos ontológicos.

Respecto a los Servicios Web Semánticos, están abiertas las líneas de investigación para recomendar mecanismos para determinar la correspondencia entre especificaciones y requisitos de servicios. También son necesarias la definición de formalismos para capturar la dinámica de servicios. También es importante poder realizar aportes respecto a la vinculación entre los marcos de trabajo de los Servicios Web Semánticos. Como se vio, algunas especificaciones pueden combinarse.

Bibliografía

[1] *Ley 19.511. Ley de metrología. Argentina.* B.O. 11 de Mayo de 1972, 1972. http://www.inti.gov.ar/metrologia/pdf/19511.pdf .

[2] *Decreto 0878/1989. Argentina. SiMeLa.* B.O. 27 de junio de 1989., 1989.

[3] Abbate, Janet: *From ARPANET to Internet : a history of ARPA-sponsored computer networks, 1966-1988 /.* History fast, University of Pennsylvania, 1994. http://worldcat.org/oclc/43188487.

[4] Abernethy, Randy, Randy Morin y Jesus Chahin: *COM/Dcom Unleashed.* Sams, Indianapolis, IN, USA, 1999, ISBN 0672313529.

[5] Abie, Habtamu: *An Overview of Firewall Technologies*, Noviembre 23 2000. http://citeseer.ist.psu.edu/452437.html;http://www.nr.no/publications/FirewallTechnologies.pdf .

[6] AFIP - Administración Federal de Ingresos Públicos.: *La AFIP - ¿Qué es el CUIT?* [En Línea]. http://www.afip.gov.ar/ET/chicos/chicos_temas_cuit.htm, [Consulta: mayo de 2008].

[7] Akkiraju, Rama, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc Thomas Schmidt, Amit Sheth y Kunal Verma: *Web Service Semantics - WSDL-S.* Technical Note v. 1.0, IBM and University of Georgia, April 2005.

[8] Akkiraju, Rama, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc Thomas Schmidt, Amit Sheth y Kunal Verma: *Web Service Semantics - WSDL-S.* W3C Member Submission, W3C, nov 2005. http://www.w3.org/Submission/2005/SUBM-WSDL-S-20051107/.

[9] Allen, Paul y Stuart Frost: *Component-based development for enterprise systems: applying the SELECT perspective.* Cambridge University Press, New York, NY, USA, 1998, ISBN 0-521-64999-4.

[10] Alonso, Gustavo, Fabio Casati, Harumi Kuno y Vijay Machiraju: *Web Services: Concepts, Architecture and Applications.* Springer Verlag, 2004, ISBN 3-540-44008-9.

[11] Altinel, Mehmet y Michael J. Franklin: *Efficient Filtering of XML Documents for Selective Dissemination of Information.* En *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, páginas 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc., ISBN 1-55860-715-3.

[12] Alvestrand, H.: *Tags for the Identification of Languages.* RFC 1766 (Proposed Standard), Marzo 1995. http://www.ietf.org/rfc/rfc1766.txt, Obsoleted by RFCs 3066, 3282.

[13] Alvestrand, H.: *Tags for the Identification of Languages.* RFC 3066 (Best Current Practice), Enero 2001. http://www.ietf.org/rfc/rfc3066.txt, Obsoleted by RFCs 4646, 4647.

[14] Alvestrand, H.: *Content Language Headers.* RFC 3282 (Draft Standard), Mayo 2002. http://www.ietf.org/rfc/rfc3282.txt.

[15] Ambler, Scott W.: *Building object applications that work: your step-by-step handbook for developing robust systems with object technology.* Cambridge University Press, New York, NY, USA, 1998, ISBN 0-521-64826-2.

[16] Andersson, Christoffer: *GPRS and 3G Wireless Applications: Professional Developer's Guide.* John Wiley & Sons, Inc., New York, NY, USA, 2002, ISBN 0471189758.

[17] Andrews, Tony, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic y Sanjiva Weerawarana: *Business Process Execution Language for Web Services (BPEL4WS) Version 1.1.* Informe técnico, Mayo 2003. http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[18] Anklesaria, F., M. McCahill, P. Lindner, D. Johnson, D. Torrey y B. Albert: *The Internet Gopher Protocol (a distributed document search and retrieval protocol).* RFC 1436 (Informational), Marzo 1993. http://www.ietf.org/rfc/rfc1436.txt.

[19] Antoniou, Grigoris y Frank van Harmelen: *A Semantic Web Primer.* Cooperative Information Systems. MIT Press, April 2004, ISBN 0262012103. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0262012103.

[20] ASC Accredited Standards Committee X12: *ASC X12 - Home.* [En Línea]. http://www.x12.org/x12org/index.cfm, [Consulta: marzo de 2008].

[21] Austin, Daniel, Sharad Garg, Abbie Barbir y Christopher Ferris: *Web Services Architecture Requirements.* W3C Note, W3C, Febrero 2004. http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211.

[22] Austin, J.: *How to Do Things With Words (William James Lectures).* Harvard University Press, September 1975, ISBN 0674411528. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0674411528.

[23] Aversano, Lerina, Gerardo Canfora, Aniello Cimitile y Andrea De Lucia: *Migrating Legacy Systems to the Web: An Experience eport.* En *CSMR*, páginas 148–157, 2001.

[24] Aversano, Lerina y Maria Tortorella: *An assessment strategy for identifying legacy system evolution requirements in eBusiness context.* Journal of Software Maintenance, 16(4-5):255–276, 2004, ISSN 1040-550X.

[25] Baader, Franz, Diego Calvanese, Deborah Mcguinness, Daniele Nardi y Peter Patel-Schneider (editores): *The Description Logic Handbook - Cambridge University Press.* Cambridge University Press, spanishfirst edición, January 2003. http://dx.doi.org/10.2277/0521781760.

[26] Bacon, Jean, John Bates, Richard Hayton y Ken Moody: *Using Events to Build Distributed Applications.* sdne, 00:148, 1995.

[27] Bacon, Jean y Tim Harris: *Operating Systems: Concurrent and Distributed Software Design.* Pearson Education, 2002, ISBN 0321117891.

[28] Bagci, Faruk, Holger Schick, Jan Petzold, Wolfgang Trumler y Theo Ungerer: *Ubiquitous Mobile Agent System in a P2P-Network.* En *Workshop at the Fifth Annual Conference on Ubiquitous Computing*, Seattle, USA, 2003.

[29] Bagci, Faruk, Holger Schick, Jan Petzold, Wolfgang Trumler y Theo Ungerer: *Communication and security extensions for a ubiquitous mobile agent system (UbiMAS).* En *CF '05: Proceedings of the 2nd conference on Computing frontiers*, páginas 246–251, New York, NY, USA, 2005. ACM, ISBN 1-59593-019-1.

[30] Baida, Ziv, Jaap Gordijn y Borys Omelayenko: *A shared service terminology for online service provisioning*. En Janssen, Marijn, Henk G. Sol y René W. Wagenaar (editores): *ICEC*, volumen 60 de *ACM International Conference Proceeding Series*, páginas 1–10. ACM, 2004, ISBN 1-58113-930-6. http://doi.acm.org/10.1145/1052220.1052222.

[31] Ballinger, Keith, David Ehnebuske, Christopher Ferris, Martin Gudgin, Canyang Kevin Liu, Mark Nottingham y Prasad Yendluri: *Basic Profile Version 1.1*. Final Material, Web Services Interoperability Organization, Abril 2004. http://ws-i.org/Profiles/BasicProfile-1.1.html.

[32] Banavar, Guruduth, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman y Deborra Zukowski: *Challenges: an application model for pervasive computing*. En *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, páginas 266–274, New York, NY, USA, 2000. ACM, ISBN 1-58113-197-6.

[33] Bapat, Subodh: *Object-oriented networks: models for architecture, operations, and management*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994, ISBN 0-13-031097-2.

[34] Barkley, John: *NISTIR 5277 - Comparing Remote Procedure Calls*. Informe técnico, National Institute of Standards and Technology (NIST), 1993. http://hissa.ncsl.nist.gov/rbac/nistir/5277/.

[35] Bates, John, Jean Bacon, Ken Moody y Mark Spiteri: *Using events for the scalable federation of heterogeneous components*. En *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, páginas 58–65, New York, NY, USA, 1998. ACM.

[36] Battle, Steve, Abraham Bernstein, Harold Boley, Benjamin Grosof, Michael Gruninger, Richard Hull, Michael Kifer, David Martin, Sheila McIlraith, Deborah McGuinness, Jianwen Su y Said Tabet: *Semantic Web Services Framework (SWSF) Overview*. W3C Member Submission, W3C, sep 2005. http://www.w3.org/Submission/2005/SUBM-SWSF-20050909/.

[37] BEA Systems Inc: *BEA Tuxedo. High-Performance Transaction Processing Monitor*. [En Línea]. http://www.bea.com/framework.jsp?CNT=index.htm\&FP=/content/products/tux, [Consulta: febrero de 2008].

[38] BEA Systems Inc.: *BEA WebLogic Integration 10.2*. [En Línea]. http://www.bea.com/framework.jsp?CNT=index.htm\&FP=/content/products/weblogic/integrate, [Consulta: marzo de 2008].

[39] BEA Systems Inc.: *BEA WebLogic Product Family*. [En Línea]. http://www.bea.com/framework.jsp?CNT=index.htm\&FP=/content/products/weblogic/, [Consulta: marzo de 2008].

[40] BEA Systems Inc.: *Dev2Dev Online: WebLogic Integration*. [En Línea]. http://dev2dev.bea.com/wlintegration/, [Consulta: marzo de 2008].

[41] BEA Systems Inc.: *Introduction to the BEA M3 System*. [En Línea]. http://e-docs.bea.com/wle/m322in/inhtml/insttitl.htm, [Consulta: junio de 2007].

[42] Bellifemine, Fabio, Agostino Poggi y Giovanni Rimassa: *Developing Multi-Agent Systems with a FIPA-compliant Agent Framework*. Software: Practice and Experience, 31(2), 2001. http://jmvidal.cse.sc.edu/library/bellifemine01a.pdf.

[43] Bellifemine, Fabio, Agostino Poggi y Giovanni Rimassa: *JADE: A White Paper*. EXP in search of innovation, 3(3):6–19, 2003. jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf.

[44] Belokosztolszki, András, David M. Eyers, Peter R. Pietzuch, Jean Bacon y Ken Moody: *Role-based access control for publish/subscribe middleware architectures*. En *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*, páginas 1–8, New York, NY, USA, 2003. ACM, ISBN 1-58113-843-1.

[45] Belushi, Wesal Al y Youcef Baghdadi: *An Approach to Wrap Legacy Applications into Web Services*. En *Service Systems and Service Management, 2007 International Conference*, volumen 9-11, páginas 1–6, Washington, DC, USA, 2007. IEEE Computer Society.

[46] Benatallah, Boualem, Remco M. Dijkman, Marlon Dumas y Zakaria Maamar: *Service-Oriented Software System Engineering: Challenges and Practices*, capítulo Cap. 3: Service Composition: Concepts, Techniques, Tools and Trends, páginas 48–66. IDEA GROUP PUBLISHING, 2005.

[47] Benjamin, Jan, Pim Borst, Hans Akkermans y Bob J. Wielinga: *Ontology Construction for Technical Domains*. En *EKAW '96: Proceedings of the 9th European Knowledge Acquisition Workshop on Advances in Knowledge Acquisition*, páginas 98–114, London, UK, 1996. Springer-Verlag, ISBN 3-540-61273-4.

[48] Bennett, Keith: *Legacy Systems: Coping with Success*. IEEE Software, 12(1):19–23, 1995, ISSN 0740-7459.

[49] Berners-Lee, T., R. Fielding y L. Masinter: *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Standard), Enero 2005. http://www.ietf.org/rfc/rfc3986.txt.

[50] Berners-Lee, T., J. Hendler y O. Lassila: *The semantic web*. Scientific American, 284(5):34–43, 2001.

[51] Berners-Lee, T., L. Masinter y M. McCahill: *Uniform Resource Locators (URL)*. RFC 1738 (Proposed Standard), Deciembre 1994. http://www.ietf.org/rfc/rfc1738.txt, Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986.

[52] Berners-Lee, Tim: *Semantic Web in XML200*. [En Línea]. http://www.w3.org/2000/Talks/1206-xml2k-tbl/, [Consulta: enero 2009].

[53] Berners-Lee, Tim: *CERN experience*. En *ACM SIGOPS European Workshop*. ACM, 1988. http://doi.acm.org/10.1145/504092.504100.

[54] Berners-Lee, Tim: *Information Management: A Proposal*. CERN document, 1989.

[55] Berners-Lee, Tim: *The WorldWideWeb browser*. [En Línea], 1990. http://www.w3.org/People/Berners-Lee/WorldWideWeb, [Consulta: enero de 2008].

[56] Berners-Lee, Tim: *World Wide Web (First Pages)*. [En Línea], 1990. http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html, [Consulta: enero de 2008].

[57] Berners-Lee, Tim: *WWW Daemon user guide*. [En Línea], 1990. http://www.w3.org/History/19921103-hypertext/hypertext/WWW/Daemon/User/Guide.html, [Consulta: enero de 2008].

[58] Berners-Lee, Tim y Robert Cailliau: *World-Wide Web*, Enero 07 1994. http://citeseer.ist.psu.edu/342644.html;ftp://gatekeeper.dec.com/pub/net/infosys/www/doc/chep92www.ps.Z.

[59] Berners-Lee, Tim, Robert Cailliau y Bernd Pollermann: *World-Wide Web: The Information Universe*, Marzo 07 1992. http://citeseer.ist.psu.edu/98059.html;ftp://ftp.irit.fr/pub/logiciels/network/www/cern/doc/ENRAP_9202.ps.gz.

[60] Bernstein, Philip y Eric Newcomer: *Principles of transaction processing: for the systems professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, ISBN 1-55860-415-4.

[61] Bernstein, Philip A., Meichun Hsu y Bruce Mann: *Implementing recoverable requests using queues*. SIGMOD Rec., 19(2):112–122, 1990, ISSN 0163-5808.

[62] Birman, Kenneth P.: *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005, ISBN 0387215093.

[63] Birrell, Andrew D. y Bruce Jay Nelson: *Implementing remote procedure calls*. ACM Transactions on Computer Systems, 2(1):39–59, February 1984. http://citeseer.ist.psu.edu/birrell84implementing.html.

[64] Bisbal, Jesús, Deirdre Lawless, Bing Wu y Jane Grimson: *Legacy Information Systems: Issues and Directions*. IEEE Software, 16(5):103–111, 1999. http://citeseer.ist.psu.edu/bisbal99legacy.html.

[65] Bisbal, Jesus, Deirdre Lawless, Bing Wu, Jane Grimson, Vincent Wade, Ray Richardson y D O'Sullivan: *An Overview of Legacy System Migration*. En *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, página 529, Washington, DC, USA, 1997. IEEE Computer Society, ISBN 0-8186-8271-X.

[66] Blair, Gorden S. y Jean Bernard Stefani: *Open Distributed Processing and Multimedia*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998, ISBN 0201177943.

[67] Bluetooth SIG Inc.: *Bluetooth.com | The Official Bluetooth Technology Info Site*. [En Línea]. http://www.bluetooth.com/bluetooth/, [Consulta: marzo de 2008].

[68] Bodhuin, T., E. Guardabascio y M. Tortorella: *Migrating COBOL Systems to the WEB by Using the MVC Design Pattern*. En *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, página 329, Washington, DC, USA, 2002. IEEE Computer Society.

[69] Bodoff, David, Mordechai Ben-Menachem y Patrick C. K. Hung: *Web Metadata Standards: Observations and Prescriptions*. IEEE Softw., 22(1):78–85, 2005, ISSN 0740-7459.

[70] Bonsma, Erwin y Cefn Hoile: *A Distributed Implementation of the SWAN Peer-to-Peer Look-Up System Using Mobile Agents*. En *AP2PC*, páginas 100–111, 2002. http://springerlink.metapress.com/openurl.asp?genre=article\&issn=0302-9743\&volume=2530\&spage=100.

[71] Booth, David y Canyang Kevin Liu: *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. W3C Recommendation, W3C, Junio 2007. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626.

[72] Borland Software Corporation: *Borland VisiBrokerÂ®. A Robust CORBAÂ® Environment for Distributed Processing*. [En Línea]. http://www.borland.com/us/products/visibroker/index.html, [Consulta: marzo de 2008].

[73] Borland Software Corporation: *VisiBrokerÂ® ITS*. [En Línea]. http://techpubs.borland.com/am/visibroker/its/, [Consulta: marzo de 2008].

[74] Bosworth, Adam: *Developing Web Service*. En *Proceedings of the 17th International Conference on Data Engineering*, páginas 477–481, Washington, DC, USA, 2001. IEEE Computer Society, ISBN 0-7695-1001-9.

[75] Boubez, Toufic, Maryann Hondo, Asir S Vedamuthu, Prasad Yendluri, Ümit Yalçinalp, David Orchard y Frederick Hirsch: *Web Services Policy 1.5 - Guidelines for Policy Assertion Authors.* W3C Note, W3C, Noviembre 2007. http://www.w3.org/TR/2007/NOTE-ws-policy-guidelines-20071112.

[76] Boucher, Karen y Fima Katz: *Essential guide to object monitors.* John Wiley & Sons, Inc., New York, NY, USA, 1999, ISBN 0-471-31971-6.

[77] Bovenzi, Diego, Gerardo Canfora y Anna Rita Fasolino: *Enabling Legacy System Accessibility by Web Heterogeneous Clients.* En *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, página 73, Washington, DC, USA, 2003. IEEE Computer Society, ISBN 0-7695-1902-4.

[78] Box, Don, Aaron Skonnard y John Lam: *Essential XML: Beyond Markup.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000, ISBN 0201709147.

[79] Brachman, Ronald J. y James G. Schmolze: *An overview of the KL-ONE knowledge representation system.* Cognitive Science, 9(2):171–216, 1985.

[80] Bray, Jennifer y Charles Sturman: *Bluetooth 1.1: Connect Without Cables, Second Edition.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001, ISBN 0130661066.

[81] Brickley, Dan y Ramanathan V. Guha: *RDF Vocabulary Description Language 1.0: RDF Schema.* W3C Recommendation, W3C, Febrero 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[82] Brodie, Michael L. y Michael Stonebraker: *Migrating legacy systems: gateways, interfaces & the incremental approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, ISBN 1558603301. http://portal.acm.org/citation.cfm?id=208444.

[83] Bryan, John: *Build a Firewall: Protect your networks from unwanted intruders.* Byte Magazine, 20(4):91–??, Abril 1995, ISSN 0360-5280.

[84] Bundesministerium für Wirtschaft und Technologie: *An open and future-oriented community | eCl@ss, the international standard for the classification of products and services.* [En Línea]. http://www.eclass-online.com/, [Consulta: enero 2009].

[85] Burdett, David y Nickolas Kavantzas: *WS Choreography Model Overview.* W3C Working Draft, W3C, Marzo 2004. http://www.w3.org/TR/2004/WD-ws-chor-model-20040324/.

[86] Burstein, Mark, Jerry Hobbs, Ora Lassila, Drew Mcdermott, Sheila Mcilraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan y Katia Sycara: *OWL-S: Semantic Markup for Web Services.* Website, November 2004. http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.

[87] Buschmann, Frank, Kevlin Henney y Douglas Schmidt: *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series).* John Wiley & Sons, 2007, ISBN 0470059028.

[88] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal: *Pattern-oriented Software Architecture: A System ofPatterns*, volumen 1. Wiley, 1996.

[89] Bussler, Christoph, Emilia Cimpian, Dieter Fensel, Juan Miguel Gomez, Armin Haller, Thomas Haselwanter, Michael Kerrigan, Adrian Mocan, Matthew Moran, Eyal Oren, Brahmananda Sapkota, Ioan Toma, Jana Viskova, Tomas Vitvar, Maciej Zaremba y Michal Zaremba: *Web Service Execution Environment (WSMX).* W3C Member Submission, June 2005. http://www.w3.org/Submission/WSMX/.

[90] Cabrera, Felipe, George Copeland, Tom Freund, Bill Cox, Johannes Klein, Tony Storey y Satish Thatte: *Web Services Transactions (WS-Transactions)*. Informe técnico, Agosto 2002. http://www.ibm.com/developerworks/library/ws-transpec/.

[91] Cabrera, Felipe, George Copeland, Tom Freund, Johannes Klein, David Langworthy, David Orchard, John Shewchuk y Tony Storey: *Web Services Coordination (WS-Coordination) Version 1.0*. Informe técnico, Agosto 2002. http://www.ibm.com/developerworks/library/ws-coor/.

[92] Cabrera, Luis Felipe, George Copeland, Jim Johnson y David Langworthy: *Coordinating Web Services Activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity*. [En Línea], Febrero 2004. http://msdn.microsoft.com/en-us/library/ms996526.aspx, [Consulta: abril de 2008].

[93] Carzaniga, Antonio, David S. Rosenblum y Alexander L. Wolf: *Challenges for distributed event services: Scalability vs expressiveness*, 1999. http://citeseer.ist.psu.edu/carzaniga99challenges.html.

[94] Carzaniga, Antonio, David S. Rosenblum y Alexander L. Wolf: *Design and evaluation of a wide-area event notification service*. ACM Trans. Comput. Syst., 19(3):332–383, 2001, ISSN 0734-2071.

[95] Cerami, Ethan: *Web Services Essentials*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002, ISBN 0596002246.

[96] Cervantes, Humberto y Richard S. Hall: *Service-Oriented Software System Engineering: Challenges and Practices*, capítulo Cap. 1: Technical Concepts of Service Orientation, páginas 1–26. IDEA GROUP PUBLISHING, 2005.

[97] Chaffey, Dave: *E-Business and E-Commerce Management: Strategy, Management, and Applications*. Financial Times/Prentice Hall, 2001, ISBN 0273651889.

[98] Chester, Timothy M.: *Cross-Platform Integration with XML and SOAP*. IT Professional, 3(5):26–34, 2001, ISSN 1520-9202.

[99] Chikofsky, Elliot J. y James H. Cross II: *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Softw., 7(1):13–17, 1990, ISSN 0740-7459.

[100] Cilia, M., L. Fiege, C. Haul, A. Zeidler y A. P. Buchmann: *Looking into the past: enhancing mobile publish/subscribe middleware*. En *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*, páginas 1–8, New York, NY, USA, 2003. ACM, ISBN 1-58113-843-1.

[101] Clabby, Joe: *Web Services Explained: Solutions and Applications for the Real World*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002, ISBN 0130479632.

[102] Clark, James y Murata Makoto: *RELAX NG Tutorial*. Committee Specification, OASIS, Deciembre 2001. http://relaxng.org/tutorial-20011203.html.

[103] Cobb, Edward E.: *TP Monitors and ORBs: A Superior Client/Server Alternative*. Object Magazine., 4(9):57–61, 1995.

[104] Comer, Douglas E. y David L. Stevens: *Internetworking with TCP/IP, Vol. 3: Client-Server Programming and Applications, Linux/Posix Sockets Version*. P T R Prentice-Hall, pub-PHPTR:adr, 2001, ISBN 0-13-032071-4. http://www.phptr.com/ptrbooks/esm_0130320714.html.

[105] CommonKADS: *What is CommonKADS?* http://www.commonkads.uva.nl/frameset-commonkads.html, [Consulta: marzo 2009].

[106] Contributors: IBM, BEA Systems, Microsoft, SAP AG, Sun Microsystems: *Web Services Addressing. BEA, IBM, Microsoft, SAP and Sun submit WS-Addressing Specification to W3C for standardization.* Informe técnico, IBM Corporation, Octubre 2007. http://www.ibm.com/developerworks/library/specification/ws-add/, [Consulta: mayo de 2008].

[107] Corcho, Oscar, Mariano Fernández-López y Asunción Gómez-Pérez: *Methodologies, tools and languages for building ontologies: where is their meeting point?* Data Knowl. Eng., 46(1):41–64, 2003, ISSN 0169-023X.

[108] Cougaar Project: *CougaarForge: Bienvenid@s.* [En Línea]. http://cougaar.org/, [Consulta: marzo de 2008].

[109] Coulouris, George, Jean Dollimore y Tim Kindberg: *Sun XDR - Archive Material From Distributed Systems: Concepts and Design.* [Pdf], 1994, ISBN 0201624338. http://www.cdk3.net/ipc/Ed2/SunXDR.pdf.

[110] Coulouris, George, Jean Dollimore y Tim Kindberg: *Distributed systems (3rd ed.): concepts and design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, ISBN 0-201-61918-0.

[111] Coyle, Frank P.: *Xml, Web Services, and the Data Revolution.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002, ISBN 0201776413.

[112] Curbera, Francisco, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi y Sanjiva Weerawarana: *Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI.* IEEE Internet Computing, 6(2):86–93, March 2002, ISSN 1089-7801. http://portal.acm.org/citation.cfm?id=613693.

[113] Cycorp: *OpenCyc.org HomepageThe Syntax of CycL.* [En Línea]. http://www.cyc.com/cycdoc/ref/cycl-syntax.html, [Consulta: enero 2009].

[114] Cycorp: *what does Cyc know?* [En Línea]. http://cyc.com/cyc/technology/whatiscyc_dir/whatdoescycknow, [Consulta: enero 2009].

[115] Daconta, Michael C., Kevin T. Smith y Leo J. Obrst: *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management.* John Wiley & Sons, Inc., New York, NY, USA, 2003, ISBN 0471432571.

[116] Dakhli, Salem y Claudine Toffolon: *A Three Layers Software Development Method: Foundations and Definition.* iceccs, 00:162, 1997.

[117] DAML-S Coalition: *DAML-S 0.9 draft release.* http://www.daml.org/services/daml-s/0.9/, [Consulta: enero 2009].

[118] Davenport, Thomas H.: *Process innovation: reengineering work through information technology.* Harvard Business School Press, Boston, MA, USA, 1993, ISBN 0875843662. http://portal.acm.org/citation.cfm?id=171556.

[119] Davenport, Thomas H. y James E. Short: *The New Industrial Engineering: Information Technology and Business Process Redesign.* Sloan Management Review, 31(4):11–27, 1990. http://sloanreview.mit.edu/smr/issue/1990/summer/1/.

[120] Davies, Saida y Peter Broadhurst: *WebSphere MQ V6 Fundamentals.* IBM Red Books. IBM Corporation, International Technical Support Organization, spanishfirst edition edición, Noviembre 2005. http://www.redbooks.ibm.com/abstracts/sg247128.html.

[121] Davis, Doug: *The hidden impact of WS-Addressing on SOAP. Is the SOAP standard in for a shake-up.* Informe técnico, IBM Corporation, Abril 2004. *http://www.ibm.com/developerworks/webservices/library/ws-address.html*, [Consulta: mayo de 2008].

[122] Dayal, U., A. P. Buchmann y D. R. McCarthy: *Rules are objects too: A knowledge model for an active, object-oriented databasesystem.* En *Lecture notes in computer science on Advances in object-oriented database systems*, páginas 129–143, New York, NY, USA, 1988. Springer-Verlag New York, Inc., ISBN 0-387-50345-5.

[123] Dayem, Rifaat A.: *Mobile data and wireless LAN technologies.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997, ISBN 0-13-839051-7.

[124] DCMI: *Dublin Core Metadata Initiative (DCMI).* [En Línea]. *http://dublincore.org/*, [Consulta: enero 2009].

[125] De, Bart, Win Frank, Piessens Wouter Joosen y Tine Verhanneman: *On the importance of the separation-of-concerns principle in secure software engineering.* 2002.

[126] De Bruijn, Jos: *Using Ontologies - Enabling Knowledge Sharing and Reuse on the Semantic Web.* Technical Report DERI-2003-10-29, DERI, 2003. *http://www.debruijn.net/publications/DERI-TR-2003-10-29.pdf*.

[127] De Bruijn, Jos, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta König-Ries, Jacek Kopecky, Rubén Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna y Michael Stollberg: *Web Service Modeling Ontology (WSMO).* W3C Member Submission, June 2005. *http://www.w3.org/Submission/WSMO/*.

[128] De Bruijn, Jos, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Michael Kifer, Birgitta König-Ries, Jacek Kopecky, Rubén Lara, Eyal Oren, Axel Polleres, James Scicluna y Michael Stollberg: *D2v1.2. Web Service Modeling Ontology (WSMO) - Final Draft.* WSMO Final Draft, 2005. *http://www.wsmo.org/TR/d2/v1.2/*.

[129] De Bruijn, Jos, Holger Lausen, Reto Krummenacher, Axel Polleres, Livia Predoiu, Michael Kifer y Dieter Fensel: *The Web Service Modeling Language WSML.* Informe técnico, DERI, October 2005.

[130] Dean, Mike y Guus Schreiber: *OWL Web Ontology Language Reference.* W3C Recommendation, W3C, February 2004.

[131] Delamer, Ivan M. y Jose L. Martinez Lastra: *Self-Orchestration and Choreography: Towards Architecture-Agnostic Manufacturing Systems.* En *AINA*, páginas 573–582. IEEE Computer Society, 2006, ISBN 0-7695-2466-4. *http://doi.ieeecomputersociety.org/10.1109/AINA.2006.301*.

[132] Deutsch, L. P.: *Design Reuse and Frameworks in the Smalltalk-80 System.* En Biggerstaff, T. J. y C. Richter (editores): *Software Reusability*, volumen II — Applications and Experience, capítulo 3, páginas 57–71. acm press, 1989.

[133] Dierks, T. y C. Allen: *The TLS Protocol Version 1.0.* RFC 2246 (Proposed Standard), Enero 1999. *http://www.ietf.org/rfc/rfc2246.txt*, Obsoleted by RFC 4346, updated by RFC 3546.

[134] Dierks, T. y E. Rescorla: *The Transport Layer Security (TLS) Protocol Version 1.1.* RFC 4346 (Proposed Standard), Abril 2006. *http://www.ietf.org/rfc/rfc4346.txt*, Updated by RFCs 4366, 4680, 4681.

[135] Dignum, Frank y Mark Greaves: *Issues in Agent Communication: An Introduction.* En *Issues in Agent Communication*, páginas 1–16, 2000.

[136] Dijkstra, Edsger Wybe: *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1976, ISBN 013215871X.

[137] Ding, Ying, Dieter Fensel, Michel Klein y Borys Omelayenko: *The Semantic Web: Yet Another Hip?* Data and Knowledge Engineering, forthcoming. http://www.cs.vu.nl/~dieter/research.html.

[138] Dorda, Santiago C., Kurt Wallnau, Robert C. Seacord y John Robert: *A Survey of Legacy System Modernization Approaches.* Technical Note CMU/SEI-2000-TN-003, Software Engineering Institute of Carnegie Mellon University, April 2000.

[139] Draves, Richard P., Michael B. Jones y Mary B. Thompson: *MIG — The Mach Interface Generator*, 1989.

[140] Dun & Bradstreet, Inc.: *Information on the D&B D-U-N-S Number and how to request one at dnb.com.* [En Línea]. http://www.dnb.com/US/duns_update/, [Consulta: mayol de 2008].

[141] Eastlake, Donald y Joseph Reagle: *XML Encryption Syntax and Processing.* W3C Recommendation, W3C, Deciembre 2002. http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

[142] Eckerson, Wayne W.: *Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications.* Open Information Systems, 10, 1:3(20), 1995.

[143] Eisler, M.: *XDR: External Data Representation Standard.* Informe técnico 4506, Internet Engineering Task Force, Mayo 2006. http://www.ietf.org/rfc/rfc4506.txt.

[144] Emtage, Alan y Peter Deutsch: `archie` — *An Electronic Directory Service for the Internet.* Technical report, McGill University, Montréal, Québec, Canada, 1992.

[145] Encyclopædia Britannica: *automated teller machine.* [En Línea], 2008. http://www.britannica.com/EBchecked/topic/44844/automated-teller-machine, [Consulta: abril de 2008].

[146] Endrei, Mark, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo y Tony Newling: *Patterns: Service-Oriented Architecture and Web Services.* IBM Red Books. IBM Corporation, International Technical Support Organization, spanishfirst edition edición, Abril 2004. http://www.redbooks.ibm.com/abstracts/SG246303.html.

[147] Eppinger, Jeffrey L. y Scott Dietzen: *Encina: modular transaction processing.* En *COMPCON '92: Proceedings of the thirty-seventh international conference on COMPCON*, páginas 378–382, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press, ISBN 0-8186-2655-0.

[148] Erl, Thomas: *Second-Generation (WS-*) Web Services.* [En Línea]. http://www.soaspecs.com/page2.asp, [Consulta: abril de 2008].

[149] Erl, Thomas: *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004, ISBN 0131428985.

[150] Erl, Thomas: *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005, ISBN 0131858580.

[151] Erl, Thomas: *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007, ISBN 0132344823.

[152] Esprit Project.: *Renaissance Project-Methods and Tools for the Evolution and Reengineering of Legacy Systems.* [En Línea], 1997. *http://www.comp.lancs.ac.uk/computing/research/cseg/projects/renaissance/RenaissanceWeb*, [Consulta: enero 2009].

[153] ESSI WSML working group.: *WSML Web Service Modeling Language - Home Page.* [En Línea]. *http://www.wsmo.org/wsml/index.html*, [Consulta: marzo 2009].

[154] ESSI WSMO working group.: *WSMO Web Service Modeling Ontology - Home Page.* [En Línea]. *http://www.wsmo.org*, [Consulta: marzo 2009].

[155] Eugster, Patrick Th., Rachid Guerraoui y Christian Heide Damm: *On objects and events.* En *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, páginas 254–269, New York, NY, USA, 2001. ACM, ISBN 1-58113-335-9.

[156] FAQ eD2k-Kademlia: *What is ED2K?* . [En Línea]. *http://www.amule.org/wiki/index.php/FAQ_ed2k*, [Consulta: marzo de 2008].

[157] Farrel, Willy: *Introducing the Java Message Service.* [Pdf], Enero 2004. *https://www6.software.ibm.com/developerworks/education/j-jms/index.html*.

[158] Fensel, Dieter: *Ontologies: a silver bullet for knowledge management and electronic commerce.* Springer-Verlag New York, Inc., New York, NY, USA, 2001, ISBN 3-540-41602-1.

[159] Fensel, Dieter, Jürgen Angele, Stefan Decker, Michael Erdmann, Hans Peter Schnurr, Rudi Studer y Andreas Witt: *Lessons Learned from Applying AI to the Web.* Int. J. Cooperative Inf. Syst, 9(4):361–382, 2000. *http://www.worldscinet.com/journals/ijcis/09/0904/S021884300000017X.html*.

[160] Fensel, Dieter y Christoph Bussler: *The Web Service Modeling Framework WSMF.* Electronic Commerce Research and Applications, 1(2):113–137, 2002. *http://dx.doi.org/10.1016/S1567-4223(02)00015-7*.

[161] Fensel, Dieter, Stefan Decker, Michael Erdmann y Rudi Studer: *Ontobroker: The Very High Idea.* En Cook, Diane J. (editor): *FLAIRS Conference*, páginas 131–135. AAAI Press, 1998, ISBN 1-57735-051-0.

[162] Fensel, Dieter, Holger Lausen, Axel Polleres, Jos de Bruijn, Michael Stollberg, Dumitru Roman y John Domingue: *Enabling Semantic Web Services: The Web Service Modeling Ontology.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006, ISBN 3540345191.

[163] Fernandez-Lopez, Mariano, Asun Gomez-Perez, Jerome Euzenat, Aldo Gangemi, Y. Kalfoglou, D. Pisanelli, M. Schorlemmer, G. Steve, Ljiljana Stojanovic, Gerd Stumme y York Sure: *A survey on methodologies for developing, maintaining, integrating, evaluating and reengineering ontologies.* OntoWeb deliverable 1.4, Universidad Politecnia de Madrid, 2002. *http://www.aifb.uni-karlsruhe.de/WBS/ysu/publications/OntoWeb_Del_1-4.pdf*.

[164] Fernandez-Lopez, Mariano, Asuncion Gomez-Perez y Natalia Juristo: *METHONTOLOGY: from Ontological Art towards Ontological Engineering.* En *Proceedings of the AAAI97 Spring Symposium Series on Ontological Engineering*, páginas 33–40, Stanford, USA, March 1997.

[165] Ferris, Chris: *Web Services Reliable Messaging reloaded. Get updated on the new WS-ReliableMessaging specification.* Informe técnico, IBM Corporation, Febrero 2005. *http://www.ibm.com/developerworks/webservices/library/ws-rmreload/*, [Consulta: mayo de 2008].

[166] Fielding, R., J. Gettys, J. Mogul, H. Frystyk y T. Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2068 (Proposed Standard), Enero 1997. http://www.ietf.org/rfc/rfc2068.txt, Obsoleted by RFC 2616.

[167] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach y T. Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard), Junio 1999. http://www.ietf.org/rfc/rfc2616.txt, Updated by RFC 2817.

[168] Fiorano Software Technologies P Ltd.: *FioranoMQ(R) 2008.* [En Línea]. http://www.fiorano.com/products/fmq/products_fioranofmq.php, [Consulta: marzo de 2008].

[169] FIPA - Fundation of Intelligent Physical Agents: *FIPA Agent Communication Language Specifications.* [En Línea]. http://www.fipa.org/repository/aclspecs.html, [Consulta: marzo de 2008].

[170] FIPA - Fundation of Intelligent Physical Agents: *FIPA Standard Status Specifications.* [En Línea]. http://www.fipa.org/repository/standardspecs.html, [Consulta: marzo de 2008].

[171] FIPA - Fundation of Intelligent Physical Agents: *Welcome to FIPA!* [En Línea]. http://www.fipa.org/, [Consulta: marzo de 2008].

[172] Forrester, Justin: *An Overview of Secure Sockets Layer.* citeseer.ist.psu.edu/forrester00overview.html.

[173] Fowler, Martin: *Analysis Patterns: Reusable Objects Models.* Addison Wesley, 1997, ISBN 0-201-89542-0.

[174] Frank van Harmelen, Peter F. Patel Schneider y Ian Horrocks: *Reference description of the DAML+OIL (March 2001) ontology markup language.* [En Línea]. http://www.daml.org/2001/03/reference.html, [Consulta: enero 2009].

[175] Freed, N. y N. Borenstein: *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples.* RFC 2049 (Draft Standard), Noviembre 1996. http://www.ietf.org/rfc/rfc2049.txt.

[176] Freed, N. y N. Borenstein: *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* RFC 2045 (Draft Standard), Noviembre 1996. http://www.ietf.org/rfc/rfc2045.txt, Updated by RFCs 2184, 2231.

[177] Freed, N. y N. Borenstein: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.* RFC 2046 (Draft Standard), Noviembre 1996. http://www.ietf.org/rfc/rfc2046.txt, Updated by RFCs 2646, 3798.

[178] Freed, N. y J. Klensin: *Media Type Specifications and Registration Procedures.* RFC 4288 (Best Current Practice), Deciembre 2005. http://www.ietf.org/rfc/rfc4288.txt.

[179] Freed, N. y J. Klensin: *Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures.* RFC 4289 (Best Current Practice), Deciembre 2005. http://www.ietf.org/rfc/rfc4289.txt.

[180] Freeman, P.: *Reusable Software Engineering: Concepts and Research Directions.* En Biggerstaff, T. y T. E. Cheatham, Jr. (editores): *Proceedings of the ITT Workshop on Reusability in Programming*, páginas 129–137. ITT, 1983.

[181] Freeman, P.: *A perspective on reusability.* En Freeman, Peter (editor): *Tutorial: Software Reusability*, páginas 2–8. IEEE Computer Society Press, 1987.

[182] Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides: *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Lecture Notes in Computer Science, 707:406–431, 1993. http://citeseer.ist.psu.edu/gamma93design.html.

[183] Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[184] Ganti, Narsim y William Brayman: *The transition of legacy systems to a distributed architecture*. Wiley-QED Publishing, Somerset, NJ, USA, 1995, ISBN 0-471-06080-1.

[185] Garcia-Molina, Hector, Dieter Gawlick, Dieter Gawlick, Johannes Klein, Johannes Klein, Karl Kleissner, Karl Kleissner, Kenneth Salem y Kenneth Salem: *Coordinating multi-transaction activities*. Informe técnico, 1990.

[186] Garcia-Molina, Hector y Kenneth Salem: *Sagas*. SIGMOD Rec., 16(3):249–259, 1987, ISSN 0163-5808.

[187] Genesereth, Michael R.: *Knowledge Interchange Format. Draft proposed American National Standard (dpANS)*. [En Línea]. http://logic.stanford.edu/kif/dpans.html, [Consulta: enero 2009].

[188] German Research Center for Artificial Intelligence: *DIET Decentralised Information Ecosystems Technology*. [En Línea]. http://www.dfki.uni-kl.de/IVS/IVSDeutsch/Projects/diet.html, [Consulta: marzo de 2008].

[189] Gibson, Bill: *Tecnical Note 1111: Top-Down System Design*. http://msdn2.microsoft.com/en-us/teamsystem/aa718891.aspx.

[190] Goodchild, Andrew, Charles Herring y Zoran Milosevic: *Business Contracts for B2B*. En Ludwig, Heiko, Yigal Hoffner, Christoph Bussler y Martin Bichler (editores): *ISDO*, volumen 30 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2000. http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-30/paper8.pdf.

[191] Gray, Jim: *Notes on Data Base Operating Systems*. En Flynn, Michael J., Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer y Hans Rüdiger Wiehle (editores): *Operating Systems, An Advanced Course*, volumen 60 de *Lecture Notes in Computer Science*, páginas 393–481. Springer, 1978, ISBN 3-540-08755-9.

[192] Gray, Jim y Andreas Reuter: *Transaction Processing : Concepts and Techniques (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, San Francisco, CA, USA, October 1992, ISBN 1558601902. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/1558601902.

[193] Gregor, Alex, Dave Johnson y Carl Wohlers: *IBM Component Broker on System/390*. IBM Red Books. IBM Corporation, International Technical Support Organization, spanishfirst edition edición, Noviembre 1998. http://www.redbooks.ibm.com/abstracts/sg245127.html.

[194] Grosof, Benjamin N., Ian Horrocks, Raphael Volz y Stefan Decker: *Description logic programs: combining logic programs with description logic*. En *WWW*, páginas 48–57, 2003. http://doi.acm.org/10.1145/775152.775160.

[195] Gruber, Thomas R.: *A translation approach to portable ontology specifications*. Knowl. Acquis., 5(2):199–220, 1993, ISSN 1042-8143.

[196] Gruber, Thomas R.: *Toward principles for the design of ontologies used for knowledge sharing*. Int. J. Hum.-Comput. Stud., 43(5-6):907–928, 1995, ISSN 1071-5819.

[197] Grüninger, Michael: *A guide to the ontology of the process specification language*. International Handbooks on Information Systems. Springer, 2004, ISBN 3-540-40834-7.

[198] Grüninger, Michael y Mark S. Fox: *Methodology for the Design and Evaluation of Ontologies*, Junio 08 1995. http://citeseer.ist.psu.edu/217283.html;http://quebec.ie.utoronto.ca/EIL/public/method.ps.

[199] Grüninger, Michael y Christopher Menzel: *The process specification language (PSL) theory and applications*. AI Mag., 24(3):63–74, 2003, ISSN 0738-4602.

[200] Guarino, N.: *Formal Ontology in Information Systems: Proceedings of the 1st International Conference June 6-8, 1998, Trento, Italy*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1998, ISBN 9051993994.

[201] Gudgin, Martin, Marc Hadley y Tony Rogers: *Web Services Addressing 1.0 - Core*. W3C Recommendation, W3C, Mayo 2006. http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.

[202] Gupta, Gopal: *Horn Logic Denotations*. En *IJCSLP*, páginas 357–358, 1998.

[203] Gurevich, Yuri: *Evolving Algebras 1993: Lipari Guide*. En *Specification and Validation Methods*, páginas 9–36. Oxford University Press, 1995.

[204] Guttag, John: *Abstract data types and the development of data structures*. Commun. ACM, 20(6):396–404, 1977, ISSN 0001-0782.

[205] Haag, Stephen, Donald J. McCubbrey y Maeve Cummings: *Management and Information Systems for the Information Age,3e*. McGraw-Hill Higher Education, 2001, ISBN 0072458720.

[206] Haas, Hugo y Allen Brown: *Web Services Glossary*. W3C Note, W3C, Febrero 2004. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/.

[207] Hamzeh, K., G. Pall, W. Verthein, J. Taarud, W. Little y G. Zorn: *Point-to-Point Tunneling Protocol (PPTP)*. RFC 2637 (Informational), Julio 1999. http://www.ietf.org/rfc/rfc2637.txt.

[208] Hansmann, Uwe, Martin S. Nicklous y Thomas Stober: *Pervasive computing handbook*. Springer-Verlag New York, Inc., New York, NY, USA, 2001, ISBN 3-540-67122-6.

[209] Harold, Elliote Rusty y W. Scott Means: *XML in a nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002, ISBN 0-596-00292-0.

[210] Hasselbring, W.: *Information System Integration*. Communications of the ACM, 43(6):32–36, 6 2000. http://se.informatik.uni-oldenburg.de/pubdb_files/pdf/CACM-ISI2000.pdf.

[211] Heflin, Jeff y James Hendler: *Dynamic ontologies on the web*. En *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, Menlo Park, California, 2000. AAAI/MIT Press.

[212] Heimbigner, Dennis y Dennis McLeod: *A federated architecture for information management*. ACM Trans. Inf. Syst., 3(3):253–278, 1985, ISSN 1046-8188.

[213] Heinlein, Paul: *FastCGI*. 1998. http://portal.acm.org/citation.cfm?id=327524\&coll=portal\&dl=ACM.

[214] Held, Gilbert: *Understanding Cookies*. Sys Admin: The Journal for UNIX Systems Administrators, 7(4):51–54, Abril 1998, ISSN 1061-2688. http://www.samag.com/.

[215] Hemenway, Kevin y Tara Calishain: *Spidering Hacks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003, ISBN 0596005776.

[216] Hewlett-Packard Company: *Standard Template Library Programmer's Guide*. [En Línea]. http://www.sgi.com/tech/stl/index.html, [Consulta: junio 2008].

[217] Hewlett-Packard Development Company, L.P.: *TIBCO ActiveEnterprise platform solution*. [En Línea]. http://h71028.www7.hp.com/enterprise/cache/9813-0-0-0-121.html, [Consulta: marzo de 2008].

[218] Hickman, Kipp E. B. y Taher ElGamal: *The SSL Protocol*. RFC draft, Netscape Communications Corp., Junio 1995. http://home.netscape.com/newsref/std/SSL.html, Version 3.0, expires 12/95.

[219] Hill, N. C. y D. M. Ferguson: *Electronic Data Interchange: A Definition and Perspective*. EDI Forum: The Journal of Electronic Data Interchange, páginas 5–12, 1988. citeseer.ist.psu.edu/hill89electronic.html.

[220] Hoare, C. A. R.: *An axiomatic basis for computer programming*. Commun. ACM, 12(10):576–580, 1969, ISSN 0001-0782.

[221] Hoare, C. A. R.: *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985, ISBN 0-13-153271-5.

[222] Hohpe, Gregor: *Hub and Spoke [or] Zen and the Art of Message Broker Maintenance*. [En Línea], 2003. http://www.enterpriseintegrationpatterns.com/ramblings/03_hubandspoke.html, [Consulta: marzo de 2008].

[223] Horrocks, Ian, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof y Mike Dean: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. Informe técnico, May 2004.

[224] Hors, Arnaud Le, David Raggett y Ian Jacobs: *HTML 4.01 Specification*. W3C Recommendation, W3C, Deciembre 1999. http://www.w3.org/TR/1999/REC-html401-19991224.

[225] Horswill, John y Members of the CICS Development Team at IBM Hursley: *Designing and Programming CICS Applications*. O'Reilly & Associates, Inc.O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Julio 2000, ISBN 1-56592-676-5. http://www.oreilly.com/catalog/cics.

[226] Hower, Chad Z.: *Dude, where's my business logic?* [En Línea], 2005. http://www.codeproject.com/gen/design/DudeWheresMyBusinessLogic.asp, [Consulta: 13-septiembre-2007].

[227] Hunter, Jason: *Java Servlet Programming*. O'Reilly & Associates, Inc, 1998. http://www.servlets.com/index.html.

[228] IBM Corporation: *CICS Family. Customer Information Control System (CICS)*. [En Línea]. http://www-306.ibm.com/software/htp/cics/, [Consulta: febrero de 2008].

[229] IBM Corporation: *Encina Transactional Programming Guide. Version 5.1. Third Edition*. [En Línea]. http://publib.boulder.ibm.com/infocenter/txformp/v5r1/topic/com.ibm.txseries510.doc/aetgpt0002.htm, [Consulta: febrero de 2008].

[230] IBM Corporation: *IBM DCE for AIX, Version 2.2 Documentation. Introduction to DCE*. [En Línea]. http://www.univie.ac.at/dcedoc/, [Consulta: febrero de 2008].

[231] IBM Corporation: *WebSphere Application Server Toolkit. Developing Web services.* [En Línea]. http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/org.eclipse.jst.ws.doc.user/tasks/toverws.html, [Consulta: abril de 2008].

[232] IBM Corporation: *WebSphere MQ.* [En Línea]. http://www-306.ibm.com/software/integration/wmq/, [Consulta: marzo de 2008].

[233] IEEE Institute of Electrical and Electronics Engineers: *IEEE Standard for Developing a Software Project Life Cycle Process.* Informe técnico, IEEE Institute of Electrical and Electronics Engineers, 2006. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1665059.

[234] IONA Technologies: *OrbixOTM 3.0 Documentation.* [En Línea]. http://www.iona.com/support/docs/orbixotm/orbixotm30.xml, [Consulta: marzo de 2008].

[235] ISO - International Organization for Standardization: *ISO - Maintenance Agency for ISO 3166 country codes - English country names and code elements.* [En Línea]. http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm, [Consulta: mayo de 2008].

[236] Jacobsen, O.J. y D.C. Lynch: *A Glossary of Networking Terms.* RFC 1208 (Informational), Marzo 1991. http://www.ietf.org/rfc/rfc1208.txt.

[237] Jacobson, Ivar: *Object-Oriented Software Engineering: a Use Case driven Approach.* Addison–Wesley, Wokingham, England, 1995.

[238] java.net: *JAX-RPC Reference Implementation Project.* [En Línea]. https://jax-rpc.dev.java.net/, [Consulta: abril de 2008].

[239] java.net: *JAX-WS Reference Implementation Project.* [En Línea]. https://jax-ws.dev.java.net/, [Consulta: abril de 2008].

[240] java.net: *SAAJ - SOAP with Attachments API for JavaTM 1.3.* [En Línea]. https://saaj.dev.java.net/, [Consulta: abril de 2008].

[241] jBoss.Org: *JBossWiki : JBossMQ.* [En Línea]. http://wiki.jboss.org/wiki/JBossMQ, [Consulta: marzo de 2008].

[242] Jennings, Nicholas R.: *Agent-Oriented Software Engineering.* En Garijo, Francisco J. y Magnus Boman (editores): *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volumen 1647, páginas 1–7. Springer-Verlag: Heidelberg, Germany, 30– 2 1999. citeseer.ist.psu.edu/article/jennings00agentoriented.html.

[243] Jennings, Nicholas R.: *On agent-based software engineering.* Artificial Intelligence, 117(2):277–296, March 2000. http://dx.doi.org/10.1016/S0004-3702(99)00107-1.

[244] Jennings, Nicholas R.: *An agent-based approach for building complex software systems.* Commun. ACM, 44(4):35–41, April 2001, ISSN 0001-0782. http://portal.acm.org/citation.cfm?id=367250.

[245] Jennings, Nicholas R. y Michael J. Wooldridge (editores): *Agent technology: foundations, applications, and markets.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998, ISBN 3-540-63591-2.

[246] Jia, Weijia y Wanlei Zhou: *Distributed Network Systems: From Concepts to Implementations (Network Theory and Applications).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006, ISBN 0387238395.

[247] Kaaranen, Heikki, Siamak Naghian, Lauri Laitinen, Ari Ahtiainen y Valtteri Niemi: *UMTS Networks: Architecture, Mobility and Services*. John Wiley & Sons, Inc., New York, NY, USA, 2001, ISBN 047148654X.

[248] Karlsson, E. A. (editor): *Software Reuse: A Holistic Approach*. John Wiley & Sons, 1995.

[249] Kaye, Doug: *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003, ISBN 1881378241.

[250] Kifer, Lausen y Wu: *Logical Foundations of Object-Oriented and Frame-Based Languages*. JACM: Journal of the ACM, 42, 1995.

[251] Kiryakov, A., D. Ognyanov y V. Kirov: *A Framework for Representing Ontologies Consisting of Several Thousand Concepts Definitions*. DIP Project Deliverable D2.2, 2004. http://dip.semanticweb.org/deliverables/D22ORDIv1.0.pdf .

[252] Klein, Michel: *Combining and Relating Ontologies: An Analysis of Problems and Solutions*, Abril 2001. http://citeseer.ist.psu.edu/463094.html;http://www.cs.vu.nl/~mcaklein/papers/IJCAI01-ws.pdf .

[253] Klensin, J.: *Simple Mail Transfer Protocol*. RFC 2821 (Proposed Standard), Abril 2001. http://www.ietf.org/rfc/rfc2821.txt .

[254] Knowledge Systems Laboratory. Stanford University: *Ontolingua Home Page*. [En Línea]. http://www.ksl.stanford.edu/software/ontolingua/ , [Consulta: enero 2009].

[255] Kohl, J. y C. Neuman: *The Kerberos Network Authentication Service (V5)*. RFC 1510 (Proposed Standard), Septiembre 1993. http://www.ietf.org/rfc/rfc1510.txt, Obsoleted by RFC 4120.

[256] Kong, Mike, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato y Geoffrey L. Wyant: *Network Computing System Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990, ISBN 0-13-617085-4.

[257] Kounev, Samuel y Alejandro Buchmann: *Improving data access of J2EE applications by exploiting asynchronous messaging and caching services*. En *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, páginas 574–585. VLDB Endowment, 2002.

[258] Krafzig, Dirk, Karl Banke y Dirk Slama: *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004, ISBN 0131465759.

[259] Krill, Paul: *Microsoft, IBM, SAP To Discontinue UDDI Web Services Registry Effort*. [En Línea], 16 de diciembre del 2005. http://www.infoworld.com/article/05/12/16/HNuddishut_1.html, [Consulta: mayo 2008].

[260] Kristol, D. y L. Montulli: *HTTP State Management Mechanism*. RFC 2965 (Proposed Standard), Octubre 2000. http://www.ietf.org/rfc/rfc2965.txt .

[261] Krueger, Charles W.: *Software Reuse*. ACM Computing Surveys, 24(2):131–183, Junio 1992.

[262] Kurose, James F. y Keith Ross: *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002, ISBN 0201976994. http://portal.acm.org/citation.cfm?id=549735 .

[263] Labrou, Yannis y Tim Finin: *Semantics and Conversations for an Agent Communication Language.* En Pollack, Martha E. (editor): *Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence (IJCAI-97)*, páginas 584–591, Nagoya, Japan, 1997. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA. *citeseer.ist.psu.edu/article/labrou97semantics.html*.

[264] Lafon, Yves y Nilo Mitra: *SOAP Version 1.2 Part 0: Primer (Second Edition).* W3C Recommendation, W3C, Abril 2007. *http://www.w3.org/TR/2007/REC-soap12-part0-20070427/*.

[265] Lakhotia, Arun y Jean Christophe Deprez: *Restructuring Functions with Low Cohesion.* En *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, página 36, Washington, DC, USA, 1999. IEEE Computer Society, ISBN 0-7695-0303-9.

[266] Lamb, David Alex: *IDL: sharing intermediate representations.* ACM Trans. Program. Lang. Syst., 9(3):297–318, 1987, ISSN 0164-0925.

[267] Lampson, B. W. y H. E. Sturgis: *Crash recovery in a distributed data storage system.* Informe técnico, Xerox Palo Alto Research Center, 1979. *http://citeseer.ist.psu.edu/lampson79crash.html*.

[268] Lara, Rubén, Dumitru Roman, Axel Polleres y Dieter Fensel: *A Conceptual Comparison of WSMO and OWL-S.* En *ECOWS 2004*, volumen 3250 de *LNCS*, páginas 254–269. Springer, 2004. *http://www.springerlink.com/content/p8358uyre5kw3h7h*.

[269] Larman, Craig: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001, ISBN 0130925691.

[270] Lassila, Ora y Ralph R. Swick: *Resource Description Framework (RDF) Model and Syntax Specification.* supersed Work, W3C, Febrero 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

[271] Laurent, S. St.: *Describing Your Data: DTDs and XML Schemas.* [En Línea], Deciembre 1999. *http://www.xml.com/pub/1999/12/dtd/*, [Consulta: abril de 2008].

[272] Leymann, Frank: *Web Services Flow Language (WSFL 1.0).* Informe técnico, IBM Corporation, Mayo 2001.

[273] Linthicum, David S.: *Enterprise application integration.* Addison-Wesley Longman Ltd., Essex, UK, UK, 2000, ISBN 0-201-61583-5.

[274] Lowe, Doug y David Helda: *Client/Server Computing for Dummies (3rd Ed).* IDG Books Worldwide, 1999, ISBN 0764504762.

[275] Maler, Eve, David Orchard y Steven DeRose: *XML Linking Language (XLink) Version 1.0.* W3C Recommendation, W3C, Junio 2001. *http://www.w3.org/TR/2001/REC-xlink-20010627/*.

[276] Maler, Eve, Jean Paoli, C. M. Sperberg-McQueen, François Yergeau y Tim Bray: *Extensible Markup Language (XML) 1.0 (Fourth Edition).* W3C Recommendation, W3C, Agosto 2006. *http://www.w3.org/TR/2006/REC-xml-20060816*.

[277] Manual de PHP: *¿Qué se puede hacer con PHP? . Introducción.* [En Línea], 2006. *http://www.php.net/manual/es/intro-whatcando.php*, [Consulta: 21-septiembre-2007].

[278] Marks, Eric A.: *Service-Oriented Architecture (SOA) Governance for the Services Driven Enterprise.* Wiley Publishing, 2008, ISBN 0470171251, 9780470171257.

[279] Marrow, P., E. Bonsma, F. Wang y C. Hoile: *DIET – A Scalable, Robust and Adaptable Multi-Agent Platform for Information Management.* BT Technology Journal, 21(4):130–137, 2003, ISSN 1358-3948.

[280] Martin, David, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry R. Payne, Evren Sirin, Naveen Srinivasan y Katia Sycara: *OWL-S: Semantic Markup for Web Services.* Informe técnico, November 2004. *http://www.w3.org/Submission/OWL-S/*.

[281] Maximilien, E. Michael y Munindar P. Singh: *Reputation and endorsement for web services.* SIGecom Exch., 3(1):24–31, 2002, ISSN 1551-9031.

[282] McCahill, M. P. y F. X. Anklesaria: *Evolution of Internet Gopher.* J.UCS: Journal of Universal Computer Science, 1(4):235–??, Abril 1995, ISSN 0948-6968. *http://www.jucs.org/evolution_of_internet_gopher*.

[283] McClure, Carma: *Software reuse techniques: adding reuse to the system development process.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997, ISBN 0-13-661000-5.

[284] McCool, Rob: *The Common Gateway Interface.* National Center of Supercomputing Applications, spanish1.1 edición, 1994. *http://hoohoo.ncsa.uiuc.edu/cgi/overview.html*.

[285] McDermott, Drew: *DRS: A set of conventions for representing logical languages in RDF.* *http://www.daml.org/services/owl-s/1.1B/DRSguide.pdf*, [Consulta: marzo 2009].

[286] McFall, Cynthia: *An Object Infrastructure for Internet Middleware: IBM on Component Broker.* IEEE Internet Computing, 2(2):46–51, 1998.

[287] McGuinness, Deborah L.: *Ontologies Come of Age.* En Fensel, Dieter, Jim Hendler, Henry Lieberman y Wolfgang Wahlster (editores): *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential.* MIT Press, Cambridge, MA, 2002.

[288] McGuinness, Deborah L., Richard Fikes, Lynn Andrea Stein y James A. Hendler: *DAML-ONT: An Ontology Language for the Semantic Web.* En Fensel, Dieter, James A. Hendler, Henry Lieberman y Wolfgang Wahlster (editores): *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, páginas 65–94. The MIT Press, Cambridge, Massachusetts, Febrero 2003, ISBN 0262062321.

[289] Mcilraith, S. A., T. C. Son y Honglei Zeng: *Semantic Web services.* Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications], 16(2):46–53, 2001. *http://dx.doi.org/10.1109/5254.920599*.

[290] Mcilroy, D.: *Mass-Produced Software Components.* En *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, páginas 88–98, 1968.

[291] McManis, Chuck y Vipin Samar: *Solaris ONC: Design and Implementation of Transport-Independent RPC*, 1991.

[292] Medjahed, B., B. Benatallah, A. Bouguettaya, A. H. H. Ngu y A. K. Elmagarmid: *Business-to-business interactions: issues and enabling technologies.* VLDB Journal: Very Large Data Bases, 12(1):59–85, Mayo 2003, ISSN 1066-8888 (PRINT), 0949-877X (ELECTRONIC). *http://link.springer.de/link/service/journals/00778/bibs/3012001/30120059.htm*.

[293] Meersman, Robert: *Semantic Ontology Tools in IS Design.* En *ISMIS '99: Proceedings of the 11th International Symposium on Foundations of Intelligent Systems*, páginas 30–45, London, UK, 1999. Springer-Verlag, ISBN 3-540-65965-X.

[294] Meyer, Bertrand: *Applying "Design by Contract"*. Computer, 25(10):40–51, 1992, ISSN 0018-9162.

[295] Meyer, Bertrand: *Object-Oriented Software Construction*. Prentice Hall PTR, March 1997, ISBN 0136291554. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0136291554.

[296] Microsoft Corporation: *COM: Component Object Model Technologies*. [En Línea]. http://www.microsoft.com/com/default.mspx, [Consulta: noviembre de 2007].

[297] Microsoft Corporation: *Introduction to ActiveX Controls*. [En Línea]. http://msdn.microsoft.com/en-us/library/aa751972(VS.85).aspx, [Consulta: marzo de 2008].

[298] Microsoft Corporation: *Microsoft Developer Network. Microsoft patterns and practices. Integration Topologies*. [En Línea]. http://msdn.microsoft.com/es-ar/library/ms978579(en-us).aspx, [Consulta: febrero de 2008].

[299] Microsoft Corporation: *Microsoft Message Queuing*. [En Línea]. http://www.microsoft.com/spain/windowsserver2003/technologies/msmq/default.aspx, [Consulta: marzo de 2008].

[300] Microsoft Corporation: *Microsoft Open Database Connectivity (ODBC)*. [En Línea]. http://msdn2.microsoft.com/en-us/library/ms710252.aspx, [Consulta: Agosto-2007].

[301] Microsoft Corporation: *Microsoft TechNet. Quick Tour of MS Transaction Server*. [En Línea]. http://www.microsoft.com/technet/archive/transsrv/quicktr.mspx?mfr=true, [Consulta: febrero de 2008].

[302] Microsoft Corporation: *RPC Technical Reference*. [En Línea]. http://technet2.microsoft.com/WindowsServer/en/library/e5677c57-3182-497a-b53b-a536580b542b1033.mspx?mfr=true, [Consulta: febrero de 2008].

[303] Microsoft Corporation: *The Official Microsoft ASP.NET Site*. [En Línea]. http://www.asp.net/, [Consulta: marzo de 2008].

[304] Microsoft Corporation: *UBR Shutdown FAQ*. [En Línea]. http://uddi.microsoft.com/about/FAQshutdown.htm, [Consulta: mayo 2008].

[305] Microsoft Corporation: *Application Architecture for .NET: Designing Applications and Services*. Patterns & Practices. Microsoft Corporation, 2002.

[306] Microsoft Corporation: *Microsoft's Vision for an Identity Metasystem*. Informe técnico, Mayo 2005. http://msdn.microsoft.com/en-us/library/ms996497.aspx, [Consulta: junio de 2008].

[307] Mili, Hafedh, Fatma Mili y Ali Mili: *Reusing Software: Issues and Research Directions*. Software Engineering, 21(6):528–562, 1995. http://citeseer.ist.psu.edu/mili95reusing.html.

[308] Miller, Eric y Frank Manola: *RDF Primer*. W3C Recommendation, W3C, Febrero 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[309] Miller, Howard Wilbert: *Reengineering legacy software systems*. Digital Press, Newton, MA, USA, 1998, ISBN 1-55558-195-1.

[310] Minsky, Steven: *Business Process Management (BPM) - The Challenge of BPM Adoption*. [En Línea], 2005. http://www.ebizq.net/topics/bpm/features/5757.html, [Consulta: 19-Agosto-2007].

[311] MIT: *Kerberos: The Network Authentication Protocol*. http://web.mit.edu/kerberos/what_is.

[312] Mitchell, Richard, Jim McKim y Bertrand Meyer: *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002, ISBN 0-201-63460-0.

[313] Moats, R.: *URN Syntax*. RFC 2141 (Proposed Standard), Mayo 1997. *http://www.ietf.org/rfc/rfc2141.txt*.

[314] Mockler, Robert J., Dorothy G. Dologite y Marc E. Gartenfeld: *B2B E-Business*, volumen Encyclopedia of E-commerce, E-government and Mobile Commerce, páginas 26–30. IGI Publishing, Hershey, PA, USA, 2006, ISBN 1591407990.

[315] Monson-Haefel, Richard: *J2EE Web Services*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003, ISBN 0321146182.

[316] Moore, K.: *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*. RFC 2047 (Draft Standard), Noviembre 1996. *http://www.ietf.org/rfc/rfc2047.txt*, Updated by RFCs 2184, 2231.

[317] Moro, Gianluca, Aris M. Ouksel y Claudio Sartori: *Agents and Peer-to-Peer Computing: A Promising Combination of Paradigms*. páginas 15–28. 2003. *http://www.springerlink.com/content/jcj1kyhylb8b5xqv*.

[318] Moss, J. Eliot B.: *Nested Transactions: An Approach to Reliable Distributed Computing*. Informe técnico MIT/LCS/TR-260, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981. *http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai%3Ancstrlh%3Amitai%3AMIT-LCS%2F%2FMIT%2FLCS%2FTR-260*.

[319] MSDNAA Editor: *.NET Framework Essentials*. Microsoft Research University Relations, Agosto 03 2001. *http://www.msdnaa.net/curriculum/?382*.

[320] Mühl, Gero: *Generic Constraints for Content-Based Publish/Subscribe*. En *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, páginas 211–225, London, UK, 2001. Springer-Verlag, ISBN 3-540-42524-1.

[321] Mühl, Gero y Ludger Fiege: *Supporting Covering and Merging in Content-Based Publish/-Subscribe Systems: Beyond Name/Value Pairs*. IEEE Distributed Systems Online (DSOnline), 2(7), 2001.

[322] Mühl, Gero, Ludger Fiege y Peter Pietzuch: *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006, ISBN 3540326510.

[323] Narayanan, Srini y Sheila Mcilraith: *Simulation, Verification and Automated Composition of Web Services*. En *Proceedings of the 11th International Conference on World Wide Web*, páginas 77–88, New York, NY, USA, 2002. ACM Press.

[324] Nehmer, Jürgen y Peter Sturm: *Systemsoftware - Grundlagen moderner Betriebssysteme*. dpunkt.verlag, 2001.

[325] Nelte, Michael y Elton Saul: *Cookies: weaving the Web into a state*. 2000. *http://portal.acm.org/citation.cfm?id=351097\&coll=portal\&dl=ACM*.

[326] Netscape: *DMOZ - ODP Open Directory Project*. [En Línea]. *http://www.dmoz.org/*, [Consulta: enero 2009].

[327] Newcomer, Eric: *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002, ISBN 0201750813.

[328] Newcomer, Eric y Greg Lomow: *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, 2004, ISBN 0321180860.

[329] Nowostawski, Mariusz, Martin K. Purvis, Marcos De Oliveira y Stephen Cranefield: *Institutions in the OPAL multi-agent system.* Journal of Intelligent and Fuzzy Systems, 17(3):191–207, 2006. http://iospress.metapress.com/openurl.asp?genre=article\&issn=1064-1246\&volume=17\&issue=3\&spage=191.

[330] OASIS eXtensible Access Control Markup Language (XACML) TC: *eXtensible Access Control Markup Language (XACML) Version 2.0.* OASIS Standard, OASIS, Febrero 2005. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[331] OASIS Organization for the Advancement of Structured Information Standards: *OASIS: Advancing open standards for the global information society.* [En Línea]. http://www.oasis-open.org/, [Consulta: enero 2009].

[332] OASIS SOA Reference Model TC: *Reference Model for Service Oriented Architecture 1.0.* OASIS Standard, OASIS, Octubre 2006. http://docs.oasis-open.org/soa-rm/v1.0/.

[333] OASIS UDDI Specification TC: *UDDI Core v2 and v2/v3 Utility Classification Schemes, Taxonomies, Identifier Systems, and Relationships Version 3.0.2.* UDDI Spec Technical Committee, OASIS, Octubre 2004. http://www.uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm.

[334] OASIS UDDI Specification TC: *UDDI Executive Overview: Enabling Service-Oriented Architecture.* Organization for the Advancement of Structured Information Standards – www.oasis-open.org., 2004.

[335] OASIS UDDI Specification TC: *UDDI Version 3.0.2.* UDDI Spec Technical Committee Draft, OASIS, Octubre 2004. http://uddi.org/pubs/uddi-v3.0.2-20041019.htm.

[336] OASIS Web Services Business Process Execution Language (WSBPEL) TC: *Web Services Business Process Execution Language Version 2.0.* OASIS Standard, OASIS, Abril 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[337] OASIS Web Services Reliable Exchange (WS-RX) TC: *Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.1.* OASIS Standard, OASIS, Enero 2007. http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01-e1.html.

[338] OASIS Web Services Security (WSS) TC: *SAML Token Profile 1.1.* OASIS Standard, OASIS, Noviembre 2006. http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-errata-os-SAMLTokenProfile.pdf.

[339] OASIS Web Services Security (WSS) TC: *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004).* OASIS Standard, OASIS, Noviembre 2006. http://www.oasis-open.org/committees/download.php/21257/wss-v1.1-spec-errata-os-SOAPMessageSecurity.htm.

[340] OASIS WS-TX Technical Committee: *OASIS Web Services Transaction (WS-TX) TC Home Page.* [En Línea]. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx, [Consulta: junio 2008].

[341] OASIS WS-TX Technical Committee: *Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1.* WS-TX Committee Specification, OASIS, Julio 2007. http://docs.oasis-open.org/ws-tx/wsat/2006/06.

[342] OASIS WS-TX Technical Committee: *Web Services Business Activity (WS-BusinessActivity) Version 1.1.* WS-TX Committee Specification, OASIS, Julio 2007. http://docs.oasis-open.org/ws-tx/wsba/2006/06.

[343] OASIS WS-TX Technical Committee: *Web Services Coordination (WS-Coordination) Version 1.1*. WS-TX Committee Specification, OASIS, Abril 2007. *http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-cs-03/*.

[344] ObjectWeb Consortium: *JORAM: Java (TM) Open Reliable Asynchronous Messaging*. [En Línea]. *http://joram.objectweb.org/*, [Consulta: marzo de 2008].

[345] Odette España.: *Curso EDI Online*. [En Línea]. *http://www.odette.es/CMS/index.php?option=com_content\&task=view\&id=80\&Itemid=142*, [Consulta: abril de 2008].

[346] Oki, Brian, Manfred Pfluegl, Alex Siegel y Dale Skeen: *The Information Bus: an architecture for extensible distributed systems*. En *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, páginas 58–68, New York, NY, USA, 1993. ACM, ISBN 0-89791-632-8.

[347] OMG Object Management Group: *Catalog of OMG CORBAfacilities Specifications*. [En Línea]. *http://www.omg.org/technology/documents/corbafacilities_spec_catalog.htm*, [Consulta: noviembre de 2007].

[348] OMG Object Management Group: *Catalog of OMG CORBA/IIOP Specifications*. [En Línea]. *http://www.omg.org/technology/documents/corba_spec_catalog.htm*, [Consulta: noviembre de 2007].

[349] OMG Object Management Group: *Catalog of OMG CORBAservices Specifications*. [En Línea]. *http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm*, [Consulta: noviembre de 2007].

[350] OMG Object Management Group: *Catalog of OMG IDL Language Mappings Specifications*. [En Línea]. *http://www.omg.org/technology/documents/idl2x_spec_catalog.htm*, [Consulta: marzo de 2008].

[351] OMG Object Management Group: *CORBA BASICS: How do remote invocations work?* [En Línea]. *http://www.omg.org/gettingstarted/corbafaq.htm{\#}RemoteInvoke*, [Consulta: marzo de 2008].

[352] OMG Object Management Group: *History of CORBA*. [En Línea]. *http://www.omg.org/gettingstarted/history_of_corba.htm*, [Consulta: noviembre de 2007].

[353] OMG Object Management Group: *Trading Object Service, version 1.0*. [Pdf], 2000. *http://www.omg.org/technology/documents/formal/trading_object_service.htm*, [Consulta: marzo de 2008].

[354] OMG Object Management Group: *Life Cycle Service, version 1.2*. [En Línea], 2002. *http://www.omg.org/technology/documents/formal/life_cycle_service.htm*, [Consulta: marzo de 2008].

[355] OMG Object Management Group: *Meta-Object Facility (MOF) 1.4 Specification.*, April 2002. *http://www.omg.org/docs/formal/02-04-03.pdf*.

[356] OMG Object Management Group: *Transaction Service, version 1.4*. [En Línea], 2003. *http://www.omg.org/technology/documents/formal/transaction_service.htm*, [Consulta: marzo de 2008].

[357] OMG Object Management Group: *CORBA Event Service, version 1.2*. [Pdf], 2004. *http://www.omg.org/technology/documents/formal/event_service.htm*, [Consulta: marzo de 2008].

[358] OMG Object Management Group: *Naming Service, version 1.3*. [Pdf], 2004. *http://www.omg.org/technology/documents/formal/naming_service.htm*, [Consulta: marzo de 2008].

[359] OMG Object Management Group: *Notification Service, version 1.1.* [Pdf], 2004. http://www.omg.org/technology/documents/formal/notification_service.htm, [Consulta: marzo de 2008].

[360] OMG Object Management Group: *CORBA Component Model Specification, version 4.0.* [Pdf], 2006. http://www.omg.org/cgi-bin/doc?formal/06-04-01, [Consulta: marzo de 2008].

[361] OMG Object Management Group: *Common Object Request Broker Architecture (CORBA) Specification, version 3.1. Part 1: CORBA Interfaces - Chap. 11 Dynamic Invocation Interface.*, 2008. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF.

[362] OMG Object Management Group: *Common Object Request Broker Architecture (CORBA) Specification, version 3.1. Part 1: CORBA Interfaces - Chap. 14 The Interface Repository.*, 2008. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF.

[363] OMG Object Management Group: *Common Object Request Broker Architecture (CORBA) Specification, version 3.1. Part 1: CORBA Interfaces - Chap. 6 CORBA Overview.*, 2008. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF.

[364] OMG Object Management Group: *Common Object Request Broker Architecture (CORBA) Specification, version 3.1. Part 1: CORBA Interfaces - Chap. 7 OMG IDL Syntax and Semantics.*, 2008. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF.

[365] OMG Object Management Group: *Common Object Request Broker Architecture (CORBA) Specification, version 3.1. Part 2: CORBA Interoperability - Chap. 9 General Inter-ORB Protocol.*, 2008. http://www.omg.org/spec/CORBA/3.1/Interoperability/PDF.

[366] OMG Object Management Group: *Java to IDL Language Mapping, version 1.4.* [Pdf], 2008. http://www.omg.org/spec/JAV2I/1.4/, [Consulta: marzo de 2008].

[367] Open Mobile Alliance: *Wireless Application Protocol*, 2001. http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html\#wap20.

[368] Orchard, David, Frederick Hirsch, Asir S Vedamuthu, Prasad Yendluri, Toufic Boubez, Ümit Yalçinalp y Maryann Hondo: *Web Services Policy 1.5 - Framework.* W3C Recommendation, W3C, Septiembre 2007. http://www.w3.org/TR/2007/REC-ws-policy-20070904.

[369] Orfali, Robert, Dan Harkey y Jeri Edwards: *Client/server survival guide (3rd ed.).* John Wiley & Sons, Inc., New York, NY, USA, 1999, ISBN 0-471-31615-6.

[370] Orton, Douglas J. y Karl E. Weick: *Loosely Coupled Systems: A Reconceptualization.* The Academy of Management Review, 15(2):203–223, 1990. http://dx.doi.org/10.2307/258154.

[371] Osterman, Michael: *What do you really know about MOM?* [En Línea], Mayo 2000. http://www.networkworld.com/newsletters/gwm/0529gw1.html, [Consulta: marzo de 2008].

[372] OTK-Project: *Welcome to OIL.* [En Línea]. http://www.ontoknowledge.org/oil/, [Consulta: enero 2009].

[373] OTK-Project: *Welcome to Ontoknowledge.* [En Línea]. http://www.ontoknowledge.org/, [Consulta: enero 2009].

[374] Otte, Randy, Paul Patrick y Mark Roy: *Understanding CORBA (Common Object Request Broker Architecture).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996, ISBN 0-13-459884-9.

[375] Paolucci, Massimo, Naveen Srinivasan y Katia Sycara: *Expressing WSMO Mediators in OWLS*. En *In Proceedings of the workshop on Semantic Web Services: Preparing to Meet the World of Business Applications held at the 3rd International Semantic Web Conference (ISWC 2004*, 2004.

[376] Paradi, Joseph C. y Gloria Yan: *Business to Consumer Electronic Commerce - Introduction*. En *HICSS*, 1999. http://computer.org/proceedings/hicss/0001/00015/00015002abs.htm.

[377] Parfett, M. (editor): *What is EDI? A Guide to Electronic Data Interchange*. NCC Blackwel, Oxford, 1992.

[378] Parnas, D. L.: *On the Criteria to Be Used in Decomposing Systems into Modules*. Communications of the ACM, 15(12):1053–1058, December 1972.

[379] Patil, Abhijit A., Swapna A. Oundhakar, Amit P. Sheth y Kunal Verma: *Meteor-s web service annotation framework*. En *WWW '04: Proceedings of the 13th international conference on World Wide Web*, páginas 553–562, New York, NY, USA, 2004. ACM, ISBN 1-58113-844-X.

[380] Pedersen, Torben P.: *HTTPS, Secure HTTPS*. En van Tilborg, Henk C. A. (editor): *Encyclopedia of Cryptography and Security*. Springer, 2005, ISBN 978-0-387-23473-1. http://dx.doi.org/10.1007/0-387-23483-7_189.

[381] Penserini, Loris, Lin Liu, John Mylopoulos y Luca Spalazzi: *Modeling and Evaluating Cooperation Strategies in P2P Agent Systems*. En Moro, Gianluca y Manolis Koubarakis (editores): *AP2PC*, volumen 2530 de *Lecture Notes in Computer Science*, páginas 87–99. Springer, 2002, ISBN 3-540-40538-0. http://springerlink.metapress.com/openurl.asp?genre=article\&issn=0302-9743\&volume=2530\&spage=87.

[382] Pérez, Asunción G., Mariano Fernández y Antonio J. De Vicente: *Towards a Method to Conceptualize Domain Ontologies*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.167.

[383] Perkins, C.: *IP Encapsulation within IP*. RFC 2003 (Proposed Standard), Octubre 1996. http://www.ietf.org/rfc/rfc2003.txt.

[384] Phillips, A. y M. Davis: *Matching of Language Tags*. RFC 4647 (Best Current Practice), Septiembre 2006. http://www.ietf.org/rfc/rfc4647.txt.

[385] Phillips, A. y M. Davis: *Tags for Identifying Languages*. RFC 4646 (Best Current Practice), Septiembre 2006. http://www.ietf.org/rfc/rfc4646.txt.

[386] Plakosh, Daniel, Scott Hissam y Kurt Wallnau: *Into the Black Box: A Case Study in Obtaining Visibility into Commercial Software*, 1999.

[387] Platt, David S.: *Understanding COM+*. Microsoft Press, Redmond, WA, USA, 1999, ISBN 0735606668.

[388] Pope, Alan LaMont: *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998, ISBN 0-201-63386-8.

[389] Postel, J.: *Telnet Protocols*. RFC 318, Abril 1972. http://www.ietf.org/rfc/rfc318.txt, Updated by RFC 435.

[390] Postel, J.: *User Datagram Protocol*. Informe técnico 768, IETF - The Internet Engineering Task Force, Agosto 1980. http://www.ietf.org/rfc/rfc768.txt.

[391] Postel, J. y J. Reynolds: *File Transfer Protocol.* RFC 959 (Standard), Octubre 1985. *http://www.ietf.org/rfc/rfc959.txt*, Updated by RFCs 2228, 2640, 2773, 3659.

[392] Postel, J. y J.K. Reynolds: *Telnet Protocol Specification.* RFC 854 (Standard), Mayo 1983. *http://www.ietf.org/rfc/rfc854.txt*.

[393] Poulin, Jeffrey S.: *Measuring software reuse: principles, practices, and economic models.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996, ISBN 0-201-63413-9.

[394] Predonzani, P., A. Sillitti y T. Vernazza: *Components and data-flow applied to the integration of Web services.* Industrial Electronics Society, 2001. IECON '01. The 27th Annual Conference of the IEEE, 3(5):2204–2207, 2001.

[395] Preist, Chris: *A Conceptual Model and Technical Architecture for Semantic Web Services.* Informe técnico HPL-2005-220, Hewlett Packard Laboratories, Diciembre 21 2005. *http://www.hpl.hp.com/techreports/2005/HPL-2005-220.html;http://www.hpl.hp.com/techreports/2005/HPL-2005-220.pdf*.

[396] Pressman, Roger S.: *Software Engineering: A Practitioner's Approach.* McGraw-Hill Higher Education, 2001, ISBN 0072496681.

[397] Price, Eric V.: *Organizational Culture and Behavioral Issues Affecting Software Reuse*, mar 1997. *http://citeseer.ist.psu.edu/317940.html;ftp://gandalf.umcs.maine.edu/pub/WISR/wisr8/proceedings/ps/price.ps*.

[398] Prieto-Diaz, Rubén: *Status Report: Software Reusability.* IEEE Software, 10(3):61–66, Mayo 1993.

[399] Princeton University Cognitive Science Laboratory: *WordNet - A lexical database for the English language.* [En Línea]. *http://wordnet.princeton.edu/*, [Consulta: enero 2009].

[400] Puder, Arno, Kay Romer y Frank Pilhofer: *Distributed Systems Architecture: A Middleware Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005, ISBN 1558606483.

[401] Purvis, Martin, Stephen Cranefield, Mariusz Nowostawski y Dan Carter: *Opal: A Multi-Level Infrastructure for Agent-Oriented Software Development.* Information Science Discussion Paper Series, (Number 2002/01), 2002, ISSN 1172-6024. *citeseer.ist.psu.edu/507528.html*.

[402] Rambhia, Ajay M.: *XML Distributed Systems Design.* Sams Publishing, 2002, ISBN 0-672-32328-1.

[403] Ratnasamy, Sylvia Paul: *A scalable content-addressable network.* Tesis de Doctorado, 2002. Chair-Scott Shenker and Chair-Ion Stoica.

[404] Real Academia Española: *DICCIONARIO DE LA LENGUA ESPAÑOLA - Vigésima segunda edición. Def:Ontología.* [En Línea]. *http://buscon.rae.es/draeI/SrvltConsulta?TIPO_BUS=3\&LEMA=ontolog{\'i}a*, [Consulta: enero 2009].

[405] Reddy, M: *ORBs and ODBMSs: two complementary ways to distribute objects.* Object Magazine., 10(13):23–31, 1995.

[406] Reilly, Michael y David Reilly: *Java Network Programming & Distributed Computing.* Addison-Wesley Professional, 2002, ISBN 0201710374.

[407] Reiter, Raymond: *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* The MIT Press, Cambridge, Massachusetts, 2001.

[408] Renzel, K. y W. Keller: *Three Layer Architecture*, 1997. http://citeseer.ist.psu.edu/renzel97three.html.

[409] Rescorla, E.: *HTTP Over TLS*. RFC 2818 (Informational), Mayo 2000. http://www.ietf.org/rfc/rfc2818.txt.

[410] Robinson, D. y K. Coar: *The Common Gateway Interface (CGI) Version 1.1*. RFC 3875 (Informational), Octubre 2004. http://www.ietf.org/rfc/rfc3875.txt.

[411] Robinson, Ian: *The roots of WS-Coordination*. [En Línea]. http://ianrobinson.blogspot.com/2007/01/roots-of-ws-coordination.html, [Consulta: junio 2008].

[412] Roessler, Thomas, David Solo, Frederick Hirsch, Donald Eastlake y Joseph Reagle: *XML Signature Syntax and Processing (Second Edition)*. W3C Recommendation, W3C, Junio 2008. http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/.

[413] Rogers, Rich: *Reuse engineering for SOA*. [En Línea], Septiembre 2005. http://www.ibm.com/developerworks/webservices/library/ws-reuse-soa.html.

[414] Roman, Dumitru, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler y Dieter Fensel: *Web Service Modeling Ontology*. Applied Ontology, 1(1), 2005.

[415] Rosen, Michael, Boris Lublinsky, Kevin T. Smith y Marc J. Balcer: *Applied SOA: Service-Oriented Architecture and Design Strategies*. Wiley Publishing, 2008, ISBN 0470223650, 9780470223659.

[416] RosettaNet Home: *RosettaNet Home*. [En Línea]. http://www.rosettanet.org/cms/sites/RosettaNet/, [Consulta: abril de 2008].

[417] Ross-Talbot, Steve y Tony Fletcher: *Web Services Choreography Description Language: Primer*. W3C Working Draft, W3C, Junio 2006. http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/.

[418] Roy, Jaideep y Anupama Ramanujan: *Understanding Web Services*. IT Professional, 3(6):69–73, 2001, ISSN 1520-9202.

[419] Russell, Stuart J. y Peter Norvig: *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003, ISBN 0137903952.

[420] Santifaller, Michael: *TCP/IP and NFS: internetworking in a UNIX environment*. Addison-Wesley, Reading, MA, USA, 1991, ISBN 0-201-54432-6.

[421] Schematron: *Schematron - A language for making assertions about patterns found in XML documents*. [En Línea]. http://www.schematron.com/overview.html, [Consulta: abril de 2008].

[422] Schmidt, Douglas: *Why Software Reuse has Failed and How to Make It Work for You*. C++ Report, January 1999. http://www.dre.vanderbilt.edu/\~{}schmidt/reuse-lessons.html.

[423] Schmidt, Douglas, Michael Stal, Hans Rohnert y Frank Buschmann: *Pattern-Oriented Software Architecture*, volumen 2, Patterns for Concurrent and Networked Objects. John Wiley and Sons, New York, 2000.

[424] Schollmeier, R. y G. SchollTelecom Italia laboratorymeier: *Why peer-to-peer (P2P) does scale: an analysis of P2P traffic patterns*. páginas 112–119, September 2002. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1046320.

[425] Shaft, Teresa M. y Iris Vessey: *The relevance of application domain knowledge: characterizing the computer program comprehension process.* J. Manage. Inf. Syst., 15(1):51–78, 1998, ISSN 0742-1222.

[426] Shaw, Mary: *Architectural issues in software reuse: it's not just the functionality, it's the packaging.* SIGSOFT Softw. Eng. Notes, 20(SI):3–6, 1995, ISSN 0163-5948.

[427] Shepler, S., B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler y D. Noveck: *Network File System (NFS) version 4 Protocol.* RFC 3530 (Proposed Standard), Abril 2003. http://www.ietf.org/rfc/rfc3530.txt.

[428] Sheu, Ray Yuan, Michael Czajkowski y Martin Hofmann: *Adaptive Peer-to-Peer Agent Sensor Networks.* En *First International Workshop on Agent Technology for Sensor Networks (ATSN-07)*, Honolulu, Hawaii, USA, 2007. http://users.ecs.soton.ac.uk/acr/atsn/Paper16.pdf.

[429] SHOE Team: *SHOE (Simple HTML Ontology Extensions).* [En Línea]. http://www.cs.umd.edu/projects/plus/SHOE/, [Consulta: enero 2009].

[430] Shohoud, Yasser: *Real World XML Web Services: For VB and VB .NET Developers.* Pearson Education, Inc., 2002, ISBN 0201774259. http://yassers.com/content/book/, [Consulta: abril de 2008].

[431] Singh, Munindar P.: *Peer-to-Peer Computing for Information Systems.* páginas 167–177. 2003. http://www.springerlink.com/content/2pq221enqlvfnp5j.

[432] Singhal, Mukesh y Niranjan G. Shivaratri: *Advanced Concepts in Operating Systems.* McGraw-Hill, Inc., New York, NY, USA, 1994, ISBN 007057572X.

[433] Smith, David Mitchell: *Microsoft Continues Web Service Leadership With New XML Specs.* Informe técnico FT-13-8004, Gartner, Mayo 2001. http://www.gartner.com/DisplayDocument?doc_cd=98366.

[434] Smith, Dennis: *Migration of Legacy Assets to Service-Oriented Architecture Environments.* En *Proceedings of 29th International Conference on Software Engineering (ICSE'07 Companion)*, volumen 00, páginas 174–175, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[435] Sneed, Harry M.: *Encapsulating legacy software for use in client/server systems.* En *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, página 104, Washington, DC, USA, 1996. IEEE Computer Society, ISBN 0-8186-7674-4.

[436] Sneed, Harry M.: *Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment.* En *IWPC*, páginas 167–175. IEEE Computer Society, 2001, ISBN 0-7695-1131-7. http://computer.org/proceedings/iwpc/1131/11310167abs.htm.

[437] Sneed, Harry M.: *Wrapping Legacy COBOL Programs behind an XML-Interface.* En *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, página 189, Washington, DC, USA, 2001. IEEE Computer Society, ISBN 0-7695-1303-4.

[438] Sneed, Harry M.: *Wrapping Legacy Software for Reuse in a SOA.* En Lehner, F., H. Nösekabel y P. Kleinschmidt (editores): *Multikonferenz Wirtschaftsinformatik 2006*, volumen 2 de *XML4BPM Track*, páginas 345–360. GITO-Verlag Berlin, 2006.

[439] Sneed, Harry M. y Katalin Erdos: *Extracting Business Rules from Source Code.* En *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, página 240, Washington, DC, USA, 1996. IEEE Computer Society, ISBN 0-8186-7283-8.

[440] Snell, James: *Automating business processes and transactions in Web services. An introduction to BPELWS, WS-Coordination, and WS-Transaction.* Informe técnico, IBM Corporation, Agosto 2002. http://www.ibm.com/developerworks/library/ws-autobp/, [Consulta: abril de 2008].

[441] Snell, James, Doug Tidwell y Pavel Kulchenko: *Programming Web services with SOAP.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002, ISBN 0-596-00095-2.

[442] SOAP Lite: *SOAP::Lite for Perl.* [En Línea]. http://www.soaplite.com/, [Consulta: abril de 2008].

[443] Software AG: *NATURAL.* [En Línea]. http://www.softwareag.com/Corporate/products/natural/default.asp, [Consulta: enero 2009].

[444] Software AG: *webMethods - SOA to Accelerate Service Re-use and Savings.* [En Línea]. http://www.softwareag.com/Corporate/products/wm/default.asp, [Consulta: marzo de 2008].

[445] Software AG: *webMethods Broker.* [En Línea]. http://www.softwareag.com/es/products/wm/integration/esb/messaging/default.asp, [Consulta: marzo de 2008].

[446] SoftWrap: *SoftWrap - The Complete E-Commerce & Digital Right Solution.* [En Línea]. http://www.softwrap.com/, [Consulta: enero 2009].

[447] Sommerville, Ian: *Software engineering (5th ed.).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995, ISBN 0-201-42765-6.

[448] Spinnler, Heinrich: *EDI: Electronic Data Interchange.* Computer-Forum, páginas 30–34, Junio 1993.

[449] Srinivasan, R.: *RPC: Remote Procedure Call Protocol Specification Version 2.* RFC 1831 (Proposed Standard), Agosto 1995. http://www.ietf.org/rfc/rfc1831.txt.

[450] Staab, Steffen, Rudi Studer, Hans Peter Schnurr y York Sure: *Knowledge Processes and Ontologies.* IEEE Intelligent Systems, 16(1):26–34, 2001, ISSN 1541-1672.

[451] Stehle, Edward, Brian Piles, Jonathan Max-Sohmer y Kevin Lynch: *Migration of Legacy Software to Service Oriented Architecture.* [En Línea]. https://www.cs.drexel.edu/~bmitchel/course/cs575/WorkshopF0708/1T9Paper.pdf, [Consulta: enero 2009].

[452] Steiner, Jennifer G., Clifford Neuman y Jeffrey I. Schiller: *Kerberos: An Authentication Service for Open Network Systems.* Informe técnico, MIT Athena, March 30, 1988.

[453] Steinke, Steve: *Middleware Meets the Network.* LAN: The Network Solutions Magazine., 10(13):56, 1995.

[454] Stonebraker, Michael: *Legacy Systems - the Achilles Heel of Downsizing (Panel).* En *PDIS '94: Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, página 108, Washington, DC, USA, 1994. IEEE Computer Society, ISBN 0-8186-6400-2.

[455] Stonebraker, Michael: *Too much middleware.* SIGMOD Rec., 31(1):97–106, 2002, ISSN 0163-5808.

[456] Studer, Rudi, V. Richard Benjamins y Dieter Fensel: *Knowledge engineering: principles and methods.* Data Knowl. Eng., 25(1-2):161–197, 1998, ISSN 0169-023X.

[457] Sun Microsistems Inc.: *Java Servlet Technology.* [En Línea], 2000. http://java.sun.com/products/servlet/index.jsp, [Consulta: 21-septiembre-2007].

[458] Sun Microsystems Inc.: *Enterprise JavaBeans Technology.* [En Línea]. http://java.sun.com/products/ejb/, [Consulta: marzo de 2008].

[459] Sun Microsystems Inc.: *J2EE Connector Architecture.* [En Línea]. http://java.sun.com/j2ee/connector/index.jsp, [Consulta: marzo de 2008].

[460] Sun Microsystems Inc.: *Java Card Technology.* [En Línea]. http://java.sun.com/javacard/, [Consulta: marzo de 2008].

[461] Sun Microsystems Inc.: *Java EE at a Glance.* [En Línea]. http://java.sun.com/javaee/, [Consulta: marzo de 2008].

[462] Sun Microsystems Inc.: *Java ME at a Glance.* [En Línea]. http://java.sun.com/javame/index.jsp, [Consulta: marzo de 2008].

[463] Sun Microsystems Inc.: *Java Message Service (JMS).* [En Línea]. http://java.sun.com/products/jms/, [Consulta: marzo de 2008].

[464] Sun Microsystems Inc.: *Java Naming and Directory Interface (JNDI).* [En Línea]. http://java.sun.com/products/jndi/, [Consulta: marzo de 2008].

[465] Sun Microsystems Inc.: *Java Remote Method Invocation - Distributed Computing for Java.* [En Línea]. http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp, [Consulta: Agosto-2007].

[466] Sun Microsystems Inc.: *Java SE at a Glance.* [En Línea]. http://java.sun.com/javase/, [Consulta: marzo de 2008].

[467] Sun Microsystems Inc.: *Java Transaction Service (JTS).* [En Línea]. http://java.sun.com/javaee/technologies/jts/index.jsp, [Consulta: marzo de 2008].

[468] Sun Microsystems Inc.: *JavaServer Pages Technology.* [En Línea]. http://java.sun.com/products/jsp/, [Consulta: marzo de 2008].

[469] Sun Microsystems Inc.: *JDBC Overview.* [En Línea]. http://java.sun.com/products/jdbc/overview.html, [Consulta: Agosto-2007].

[470] Sun Microsystems Inc: *ONC+ Developer's Guide.* [En Línea]. http://docs.sun.com/app/docs/doc/816-1435, [Consulta: febrero de 2008].

[471] Sun Microsystems Inc: *Solaris Operating System.* [En Línea]. http://www.sun.com/software/solaris/index.jsp, [Consulta: febrero de 2008].

[472] Sun Microsystems Inc.: *The Java Community Process(SM) Program - Java Specification Requests 907: JavaTM Transaction API (JTA).* [En Línea]. http://jcp.org/en/jsr/detail?id=907, [Consulta: marzo de 2008].

[473] Sun Microsystems Inc.: *RPC: Remote Procedure Call Protocol specification.* RFC 1050 (Historic), Abril 1988. http://www.ietf.org/rfc/rfc1050.txt, Obsoleted by RFC 1057.

[474] Sun Microsystems Inc.: *NFS: Network File System Protocol specification.* RFC 1094 (Informational), Marzo 1989. http://www.ietf.org/rfc/rfc1094.txt.

[475] Sun Microsystems Inc: *Applets. Code Sample and Apps.* [En Línea], 2000. http://java.sun.com/applets, [Consulta: 21-septiembre-2007].

[476] Sun Microsystems Inc: *Getting Started With Forte 4GL. Release 3.5 of ForteTM 4GL.* Sun Microsystems, Inc., 2000.

[477] Sun Microsystems Inc.: *Building Web Services*. Forte for Java Programming Series. Sun Microsystems, Inc., 2002.

[478] Sybase Inc.: *SyBooks Online*. http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.ase_15.0.verity/html/verity/verity214.htm.

[479] Tanenbaum, Andrew S.: *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1984, ISBN 0138544239.

[480] Tanenbaum, Andrew S.: *Distributed Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995, ISBN 0-13-219908-4.

[481] Tanenbaum, Andrew S. y Maarten Van Steen: *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001, ISBN 0130888931.

[482] Tari, Zahir y Omran Bukhres: *Fundamentals of distributed object systems: the CORBA perspective*. John Wiley & Sons, Inc., New York, NY, USA, 2001, ISBN 0-471-35198-9.

[483] Team, IBM Web Services Architecture: *Web Services architecture overview*. [En Línea], Septiembre 2000. http://www.ibm.com/developerworks/web/library/w-ovr/, [Consulta: abril de 2008].

[484] Team, IBM Web Services Architecture: *Web services architect, Part 3: Is Web services the reincarnation of CORBA?* [En Línea], Julio 2001. http://www.ibm.com/developerworks/webservices/library/ws-arc3/, [Consulta: abril de 2008].

[485] Telecom Italia Lab (TILAB): *Jade - Java Agent DEvelopment Framework*. [En Línea]. http://jade.tilab.com/, [Consulta: marzo de 2008].

[486] Thai, Thuan L.: *Learning DCOM*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999, ISBN 1565925815. Designed By-Nancy Priest.

[487] Thatte, Satish: *XLANG: Web Services for Business Process Design*. Informe técnico, Microsoft Corporation, Junio 2001.

[488] The Apache Software Foundation: *Web Services Project @ Apache*. [En Línea]. http://ws.apache.org/soap/index.html, [Consulta: abril de 2008].

[489] The Open Group: *Trasaction Processing Titles*. [En Línea]. http://www.opengroup.org/products/publications/catalog/tp.htm, [Consulta: febrero de 2008].

[490] The Open Group: *DCE 1.2.2: Introduction to OSF DCE.*, 1997. http://www.opengroup.org/pubs/catalog/f201.htm.

[491] The Open Group: *Transfer Syntax NDR*, volumen DCE 1.1: Remote Procedure Call: CAE Specification de *Technical Standard*, capítulo Chap. 14 Transfer Syntax NDR, páginas 625–651. The Open Group, Octubre 1997. http://www.opengroup.org/onlinepubs/9629399/chap14.htm\#tagcjh_19, [Consulta: febrero de 2008].

[492] The Open Group: *Universal Unique Identifier*, volumen DCE 1.1: Remote Procedure Call: CAE Specification, capítulo Appendix A Universal Unique Identifier. The Open Group, 1997. http://www.opengroup.org/onlinepubs/9629399/apdxa.htm\#tagcjh_20, [Consulta: febrero de 2008].

[493] The Open Group: *What is Distributed Computing and DCE?* [En Línea], 2005. http://www.opengroup.org/dce/, [Consulta: febrero de 2008].

[494] The World Wide Web Consortium (W3C): *The World Wide Web Consortium (W3C). Leading the Web to Its Full Potential...* [En Línea]. http://www.w3c.org/, [Consulta: enero 2009].

[495] Thomas Freund, Tony Storey: *Transactions in the world of Web services, Part 1. An overview of WS-Transaction WS-Coordination.* Informe técnico, IBM Corporation, Agosto 2002. http://www.ibm.com/developerworks/library/ws-wstx1/, [Consulta: abril de 2008].

[496] Thomas Freund, Tony Storey: *Transactions in the world of Web services, Part 2. An overview of WS-Transaction WS-Coordination.* Informe técnico, IBM Corporation, Agosto 2002. http://www.ibm.com/developerworks/library/ws-wstx2/, [Consulta: abril de 2008].

[497] Thomas Publishing Company.: *Thomas Register.* [En Línea]. http://www.thomasnet.com/companyhistory/ThomasRegister_1800s.html, [Consulta: mayo de 2008].

[498] Thommandra, Rama Krishna, Hari Prasad Chinta, Nakul Sharma y Sateesh Ranjan: *An Approach to Identify and Extract SOA Services from Legacy iSeries Applications.* Informe técnico, 2008. http://www.scribd.com/doc/3448388/Migrating-from-Legacy-Application-to-SOA.

[499] TIBCO Software Inc.: *TIBCO ActiveEnterprise Receives Network Computing Editor's Choice Award.* [En Línea], 2002. http://www.tibco.com/company/news/releases/2002/press471.jsp, [Consulta: marzo de 2008].

[500] Tilley, Scott R., John Gerdes Jr., Terrance Hamilton, Shihong Huang, Hausi A. Müller, Dennis B. Smith y Kenny Wong: *On the business value and technical challenges of adopting Web services.* Journal of Software Maintenance, 16(1-2):31–50, 2004. http://dx.doi.org/10.1002/smr.283.

[501] Tilley, Scott R. y Dennis B. Smith: *Perspectives on Legacy System Reengineering.* http://www.sei.cmu.edu/reengineering/lsysree.html.

[502] Transarc Corporation: *Distributed Transaction Processing with Encina and the OSF DCE*, 1992. http://citeseer.ist.psu.edu/transarc92distributed.html.

[503] umts-forum.org: *UMTS Forum.* [En Línea]. http://www.umts-forum.org/, [Consulta: marzo de 2008].

[504] Unicode, Inc.: *What is Unicode? in Spanish.* [En Línea]. http://unicode.org/standard/translations/spanish.html, [Consulta: enero 2009].

[505] United Nations Economic Commission for Europe.: *United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport.* [En Línea]. http://www.unece.org/trade/untdid/welcome.htm, [Consulta: marzo de 2008].

[506] UNSPSC: *UNSPSC Homepage.* [En Línea]. http://www.unspsc.org/, [Consulta: mayo de 2008].

[507] U.S. Census Bureau: *North American Industry Classification System (NAICS).* [En Línea]. http://www.census.gov/epcd/www/naics.html, [Consulta: mayo de 2008].

[508] Uschold, Mike y Michael Grüninger: *Ontologies: principles, methods, and applications.* Knowledge Engineering Review, 11(2):93–155, 1996. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.5917.

[509] Uschold, Mike y Martin King: *Towards a methodology for building ontologies.* En *In Workshop on Basic Ontological Issues in Knowledge Sharing, held in conjunction with IJCAI-95*, 1995.

[510] Vachharajani, Manish, Neil Vachharajani y David I. August: *A Comparison of Reuse in Object-oriented Programming and Structural Modeling Systems*, Enero 28 2004. *http://citeseer.ist.psu.edu/691989.html;http://liberty.cs.princeton.edu/Publications/tech03-01_oop.ps*.

[511] van der Vlist, Eric: *RELAX NG*. O'Reilly Media, Inc., 2003, ISBN 0596004214.

[512] van Gelder, Allen, Kenneth A. Ross y John S. Schilpf: *The Well-Founded Semantics for General Logic Programs*. Journal of the ACM, 38(3):620–662, 1991.

[513] van Harmelen, Frank y Deborah L. McGuinness: *OWL Web Ontology Language Overview*. W3C Recommendation, W3C, Febrero 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[514] Vázquez, María Celeste Campo: *Tecnologías middleware para el desarrollo de servicios en entornos de computación ubicua.* Tesis de Doctorado, Departamento de Ingeniería y Telemática. Escuela Politécnica Superior. Universidad Carlos III de Madrid, 2004. *http://www.it.uc3m.es/celeste/tesis/*.

[515] Vedamuthu, Asir S. y Daniel Roth: *Understanding Web Services Policy*. Informe técnico, Julio 2004. *http://msdn.microsoft.com/en-us/library/ms996497.aspx*, [Consulta: junio de 2008].

[516] Verma, Kunal, Kaarthik Sivashanmugam, Amit Sheth, Abhijit Patil, Swapna Oundhakar y John Miller: *METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web services*. Journal of Information Technology and Management, 6:2005, 2005.

[517] Vogel, Andreas, Brett Gray y Keith Duddy: *Understanding any IDL - Lesson one: DCE and CORBA*. En *SDNE '96: Proceedings of the 3rd Workshop on Services in Distributed and Networked Environments (SDNE '96)*, página 114, Washington, DC, USA, 1996. IEEE Computer Society, ISBN 0-8186-7499-7.

[518] von Mayrhauser, A. y A. M. Vans: *Comprehension processes during large scale maintenance*. páginas 39–48, 1994. *http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=296764*.

[519] W3C World Wide Web Consortium: *W3C Semantic Web Activity*. [En Línea]. *http://www.w3.org/2001/sw/*, [Consulta: enero 2009].

[520] W3Schools.: *DTD Tutorial*. [En Línea]. *http://www.w3schools.com/dtd/default.asp*, [Consulta: abril de 2008].

[521] Wade, Andrew E.: *Distributed Client-Server databases*. Object Magazine., 4(1):47–52, 1994, ISSN 1055-3641.

[522] Wall, David A. y Arthur Griffith: *101 Instant Java Applets*. IDG Books, pub-IDG:adr, Mayo 1997, ISBN 0-7645-3082-8 (SOFTCOVER).

[523] Wallnau, Kurt C., Edwin J. Morris, Peter H. Feiler, Anthony N. Earl y Emile Litvak: *Engineering Component-Based Systems with Distributed Object Technology*. En *WWCA '97: Proceedings of the International Conference on Worldwide Computing and Its Applications*, páginas 58–73, London, UK, 1997. Springer-Verlag, ISBN 3-540-63343-X.

[524] Walmsley, Priscilla y David C. Fallside: *XML Schema Part 0: Primer Second Edition*. W3C Recommendation, W3C, Octubre 2004. *http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/*.

[525] WAP Forum: *Wireless Application Protocol: Wireless Markup Language Specification*, 1998. *citeseer.ist.psu.edu/forum00wireless.html*.

[526] WAP Forum: *Wireless Application Protocol - White Paper*, Junio 2000. http://www.wapforum.org/what/WAP_white_pages.pdf.

[527] Warren, Ian: *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999, ISBN 1852330600.

[528] Weaver, Alfred C.: *Secure Sockets Layer*. Computer, 39(4):88–90, 2006, ISSN 0018-9162.

[529] Web Services Interoperability Organization (WS-I): *Welcome to the WS-I Organization's Web site*. [En Línea]. http://www.ws-i.org/, [Consulta: enero 2009].

[530] WebOnt Working Group: *Web-Ontology (WebOnt) Working Group Home Page*. [En Línea]. http://www.w3.org/2001/sw/WebOnt/, [Consulta: enero 2009].

[531] Weibel, S., J. Kunze, C. Lagoze y M. Wolf: *Dublin Core Metadata for Resource Discovery*, 1998.

[532] Weiderman, Nelson, Linda Northrop, Dennis Smith, Scott Tilley y Kurt Wallnau: *Implications of Distributed Object Technology for Reengineering*. Technical Report CMU/SEI-97-TR-005, Software Engineering Institute of Carnegie Mellon University, June 1997.

[533] Weikum, Gerhard y Gottfried Vossen: *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann, May 2001, ISBN 1558605088. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/1558605088.

[534] Weiser, Mark: *The computer for the 21st century*. SIGMOBILE Mob. Comput. Commun. Rev., 3(3):3–11, 1999, ISSN 1559-1662.

[535] WhatIs.com: *What is stateless? - a definition from WhatIs.com*. [En Línea]. http://whatis.techtarget.com/definition/0,,sid9_gci213051,00.html, [Consulta: septiembre 2008].

[536] White, J.E.: *Proposed Mail Protocol*. RFC 524, Junio 1973. http://www.ietf.org/rfc/rfc524.txt.

[537] Wiederhold, Gio: *Mediators in the architecture of future information systems*. páginas 185–196, 1998.

[538] Wirth, Niklaus: *Program development by stepwise refinement*. Commun. ACM, 14(4):221–227, 1971, ISSN 0001-0782.

[539] Wooldridge, M.: *Agent-based software engineering*. Software Engineering. IEE Proceedings, 144(1):26–37, 1997. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=580350.

[540] Wooldridge, Michael: *Introduction to MultiAgent Systems*. John Wiley & Sons, June 2002, ISBN 047149691X. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/047149691X.

[541] WSML working group: *D16.1v1.0 WSML Language Reference WSML Final Draft*. Informe técnico, DERI, aug 2008. http://www.wsmo.org/TR/d16/d16.1/v1.0/20080808/.

[542] WSMX working group.: *WSMX: What is it?* [En Línea]. http://www.wsmx.org, [Consulta: marzo 2009].

[543] Xerox: *"Courier: The remote procedure call protocol", Xerox System Integration Standard XSIS 038112*. Informe técnico, Xerox Corporation, Stamford, CT, 1981.

[544] Yahoo! Inc.: *Yahoo! Directory*. [En Línea]. http://dir.yahoo.com/, [Consulta: enero 2009].

[545] Yalçinalp, Ümit, David Orchard, Maryann Hondo, Toufic Boubez, Prasad Yendluri, Asir S Vedamuthu y Frederick Hirsch: *Web Services Policy 1.5 - Attachment*. W3C Recommendation, W3C, Septiembre 2007. *http://www.w3.org/TR/2007/REC-ws-policy-attach-20070904*.

[546] Yu, Liyang: *Introduction to the Semantic Web and Semantic Web Services*. Chapman & Hall/CRC, 2007, ISBN 1584889330.

[547] Zeiger, Stefan: *Servlet Essentials - Version 1.3.6*. [En Línea], 1999. *http://www.novocode.com/doc/servlet-essentials/*, [Consulta: marzo de 2008].

[548] Zhang, Zhuopeng y Hongji Yang: *Incubating Services in Legacy Systems for Architectural Migration*. En *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, páginas 196–203, Washington, DC, USA, 2004. IEEE Computer Society, ISBN 0-7695-2245-9.