UNIVERSIDAD NACIONAL DEL SUR

TESIS DE DOCTOR EN CIENCIAS DE LA COMPUTACIÓN

Servicios de Mensajería Programable para Gobierno Electrónico

Elsa Estevez

BAHÍA BLANCA ARGENTINA

2009

UNIVERSIDAD NACIONAL DEL SUR

TESIS DE DOCTOR EN CIENCIAS DE LA COMPUTACIÓN

Servicios de Mensajería Programable para Gobierno Electrónico

Elsa Estevez

BAHÍA BLANCA ARGENTINA

2009

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Doctor en Ciencias de la Computación de la Universidad Nacional del Sur, Bahía Blanca, Argentina 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 de la Universidad Nacional del Sur (DCIC-UNS) y en el Centro de Gobernabilidad Electrónica (UNU-IIST-EGOV) de la Universidad de Naciones Unidas - Instituto Internacional de Tecnología de Software (UNU-IIST) en Macao SAR, China. El trabajo se realizó durante el período comprendido entre el 28 de febrero de 2006 y el 11 de agosto de 2009, bajo la dirección del Dr. Pablo Fillottrani, Profesor Adjunto del DCIC-UNS y el Dr. Tomasz Janowski, Senior Research Fellow en UNU-IIST y Director del UNU-IIST Centro de Gobernabilidad Electrónica.

Mg. Elsa Estevez

Bahía Blanca, 14 de agosto de 2009

Departamento de Ciencias e Ingeniería de la Computación

Universidad Nacional del Sur

Agradecimientos

Esta tesis está dedicada a mis padres y a mis hijos, Ignacio y Agustín. A ellos, les agradezco eternamente. Especialmente a mi padre, por su aliento permanente, su optimismo contagioso y sus silenciosas enseñanzas de cómo enfrentar las dificultades de la vida. A mi madre, por su ejemplo de trabajo y constancia, de sostén de familia y de servicio a los demás. A Ignacio, por enseñarme a aceptar las diferencias y por sus hermosas rebeldías que nos ayudaron a crecer. A Agustín, por guiarme en una vida más saludable y por su infinita comprensión.

Mi profundo agradecimiento a las personas que me ayudaron y apoyaron a lo largo de estos años para que pueda alcanzar este objetivo, especialmente a mis directores. Al Dr. Tomasz Janowski, por confiar en mí y darme muchas oportunidades, por todas sus enseñanzas, por sus ejemplos de rectitud como persona y de perfección en el trabajo, por sus charlas motivadoras en los momentos en que mis fuerzas se extinguían, por su dedicación y paciencia. Al Dr. Pablo Fillottrani, por alentarme continuamente, por estar siempre presente y dispuesto a ayudarme, por enseñarme y aconsejarme. Agradezco también a mis compañeros de trabajo. A todos los integrantes del UNU-IIST Center for Electronic Governance, por las fructíferas discusiones compartidas en un ambiente de camaradería, y por todo lo que aprendí junto a ellos. A todos los integrantes del Departamento de Ciencias e Ingeniería de la Computación de la Universidad Nacional del Sur por brindarme su apoyo y el espacio donde fundamentalmente me nutrí de la mayoría de mi experiencia académica. Deseo también mencionar con gratitud a las instituciones que me permitieron desarrollar este trabajo, al Departamento de Ciencias e Ingeniería de la Computación de la Universidad Nacional del Sur, Argentina, y al Center for Electronic Governance del International Institute for Software Technology, United Nations University, Macao SAR, China.

Por último, mi profundo reconocimiento a personas muy cercanas y queridas: a Alberto, por su amor, comprensión, compañerismo y tolerancia; a María Inés, por ayudar a cubrir mis ausencias; a Susy, por su amistad incondicional; a Mónica, por todo lo compartido; a Isa, por su apoyo.

Resumen

Gobierno Integrado es un nuevo paradigma para las administraciones públicas que promueve la colaboración y el trabajo en red entre las agencias de gobierno y entre los sectores público, privado y del voluntariado como una forma de responder mejor a las necesidades de los ciudadanos, empresas y otros órganos de gobierno. Existen muchos beneficios por colaboraciones en gobierno. Uno de ellos es la entrega de Servicios Públicos Integrados, disponibles a través de puntos de un acceso, tanto electrónico como tradicional, de acuerdo a las necesidades de los clientes y no de la estructura interna del gobierno. Otro, es la mejora en la eficiencia del gobierno, eliminando esfuerzos duplicados y haciendo un mejor uso de los escasos recursos públicos. No obstante, el paradigma presenta una completa variedad de desafíos para su implementación: (1) Legales – reconocimiento de los procesos electrónicos como legalmente equivalentes a los procesos basados en papel; (2) Financieros – financiamiento de proyectos multi-anuales y ejecutados por varias agencias; (3) Sociales – construcción de capacidades humanas para poder utilizar las nuevas tecnologías; (4) Organizacionales – creación de procesos que crucen fácilmente las fronteras organizacionales; y (5) Tecnológicos – construcción de soluciones que integren software y procesos a través de fronteras organizacionales y administrativas, capaces de adaptarse a los cambiantes requerimientos legales y organizacionales.

Esta tesis presenta el concepto y los fundamentos de Mensajería Programable – un paradigma para el intercambio automático de mensajes entre entidades colaboradoras, que responde a diversas necesidades de comunicación de entornos colaborativos complejos y dinámicos, como los que caracterizan a Gobierno Integrado. Asimismo, presenta una realización concreta de Mensajería Programable - Government-Enterprise Ecosystem Gateway (G-EEG). G-EEG es una plataforma de comunicación y coordinación de alto nivel que soporta colaboraciones a través de procesos y aplicaciones multi-organizacionales. Los requerimientos de G-EEG fueron identificados en base a dos estudios: la naturaleza de las colaboraciones entre agencias de gobierno, siguiendo un estudio del sistema de la administración pública en Macao, y en soluciones tecnológicas disponibles que permiten tales colaboraciones. A fin de implementar estos requerimientos, G-EEG consta de tres componentes:

- G-EEG-CORE – un framework de mensajería de tiempo de ejecución que permite el intercambio asincrónico de mensajes entre miembros registrados a través de canales creados y suscriptos dinámicamente;

- G-EEG-EXTEND – un repositorio de extensiones horizontales (independientes del proceso) y verticales (dependientes del proceso) y un mecanismo para la habilitación dinámica de tales extensiones por sobre G-EEG-CORE;

- G-EEG-DEVELOP – un framework de desarrollo para especificar rigurosamente, diseñar y verificar extensiones de mensajería, ya sean totalmente nuevas o creadas a partir de extensiones existentes.

G-EEG fue diseñado para ser: minimalista – basado en el menor número posible de conceptos; extensible – la funcionalidad requerida por las aplicaciones está disponible a través de extensiones en el repositorio de G-EEG-EXTEND o a través del framework de desarrollo G-EEG-DEVELOP; dinámico – las estructuras de comunicación pueden ser creadas y modificadas dinámicamente en tiempo de ejecución; y confiable – todas las partes del framework están basadas en un modelo formal, permitiendo el desarrollo riguroso de nuevas extensiones. G-EEG puede ser usado por las agencias de gobierno para intercambiar mensajes a través de canales de comunicación lógicos cuidadosamente administrados, con las extensiones necesarias habilitadas sobre ellos. Basado en los conceptos de G-EEG, un prototipo de investigación fue construido a través de un riguroso uso de técnicas de modelado en las etapas de desarrollo.

Las principales contribuciones de la tesis son: (1) el concepto de Mensajería Programable, (2) la definición, formalización e implementación de G-EEG como una concreta realización de Mensajería Programable, (3) un modelo formal para XML y la familia de tecnologías XML, resultante del esfuerzo de formalización, (4) un exhaustivo estudio de conceptos y desafíos de Gobierno Integrado, (5) evaluación rigurosa de G-EEG con respecto a Gobierno Integrado, y (6) un marco para evaluar soluciones organizacionales, tecnológicas y fundacionales para Gobierno Integrado.

Índice General

Índice de Figuras

Índice de Tablas

Índice de Definiciones

Índice de Ejemplos

Lista de Abreviaturas

AGIMO	Australian Government Information Management Office
API	Application Programming Interface
BO	Back-Office
BPEL	Business Process Execution Language
CB	Fire Services Bureau – Government of Macao, SAR, China
CIO	Chief Information Officer
CRM	Customer Relationship Management
DSAL	Labour Affairs Bureau – Government of Macao, SAR, China
DSSOPT	Land, Public Works and Transport Bureau – Government of Macao, SAR, China
DTV	Digital TV
eGIF	Electronic Government Interoperability Framework
EPOS	Electronic Procurement Optimized System
FO	Front-Office
GBP	Great British Pound (£)
GDP	Gross Domestic Product
G-EEG	Government-Enterprise Ecosystem Gateway
G2B	Government to Business
G2C	Government to Citizens
G2E	Government to Employees
G2G	Government to Government
G2V	Government to Visitors
IC	Cultural Affairs Bureau – Government of Macao, SAR, China
NGP	Nueva Gestión Pública
IACM	Civic and Municipal Affairs Bureau – Government of Macao, SAR, China
IDABC	Interoperable Delivery of European e-Governm. Services to Public Administrations, Businesses and Citizens
IM	Instant Messaging
iTV	Interactive TV
IVR	Interactive Voice Response
OSS	Open Source Software
OWL	Web Ontology Language
OWL-S	OWL-based Services Ontology
PFI	Private Finance Initiative
QoS	Quality of Service
RDF	Resource Description Framework
RPN	Reingeniería de Procesos de Negocio
SAGA	Standards and Architectures for e-Government Applications
SMS	Short Message Service
SOA	Service-Oriented Architecture
SPE	Servicio Público Electrónico
SPP	Sociedad Publico-Privada
SS	Health Bureau – Government of Macao, SAR, China
TI	Tecnologías de la Información
TIC	Tecnologías de la Información y la Comunicación
UE	Unión Europea
UML	Unified Modeling Language
UNDESA	United Nations – Department of Economics and Social Affairs
UNPAN	United Nations Public Administration Network
URL	Uniform Resource Locator
USD	United States Dollar
XML	Extended Markup Language
XSL	Extensible $_{\text{Stylesheet}}$ Language

XSLT	XSL Transformations
W3C	World Wide Web Consortium
WS	Web Service
WSMO	Web Service Modeling Ontology
WS-CDL	Web Services – Choreography Description Language

Capítulo 1

Introducción

El Gobierno Electrónico se trata de la transformación de las agencias de gobierno, soportada por tecnología, para dar un mejor servicio a sus clientes – ciudadanos, empresas, sociedad civil, y otras ramas de gobierno. Un elemento de esta transformación es el aumento en la colaboración y en el intercambio de datos entre agencias gubernamentales como una manera de: (1) optimizar el uso de recursos públicos, (2) aumentar la calidad de los servicios públicos, (3) crear una interfaz única que refleje la necesidad de los ciudadanos y de las empresas, en lugar de la estructura del gobierno, y (4) contribuir al desarrollo económico utilizando organizaciones no gubernamentales para la entrega de bienes y servicios públicos. La colaboración soportada por tecnología y el intercambio de datos en gobierno define el enfoque principal de esta tesis – Gobierno Integrado.

Este capítulo esboza el área principal de la tesis – Gobierno Electrónico, formula y justifica el principal problema tratado – la implementación de Gobierno Integrado, y desglosa este problema en un número de desafíos tecnológicos y organizacionales que deben ser tratados como parte de cualquier implementación. En el centro de tales desafíos se encuentra la colaboración y el intercambio de datos dentro del gobierno y entre entidades públicas y no públicas. Este capítulo esboza una solución, que cubre dimensiones tecnológicas, organizativas y fundacionales, propuesta en la tesis para permitir tal colaboración y argumenta, haciendo referencia a los desafíos mencionados anteriormente, cómo la solución puede ser usada en la práctica.

El capítulo está organizado de la siguiente manera. La Sección 1.1 presenta algunos antecedentes, explicando las presiones que enfrentan los gobiernos y dos tipos de iniciativas – Reforma del Sector Público y Gobierno Electrónico – emprendidas por ellos en respuesta a tales presiones. A continuación, se explica el tema principal de la tesis – Gobierno Integrado, resaltando la importancia de colaboración en gobierno, particularmente para la entrega de servicios públicos realizada por varias agencias. La Sección 1.2 presenta cinco grupos de desafíos – legales, financieros, sociales, organizacionales y tecnológicos, que enfrentan las implementaciones de Gobierno Integrado, con énfasis en los desafíos organizacionales y tecnológicos. La Sección 1.3 presenta un caso de estudio de un servicio integrado de la vida real (Licencias de Negocios) para ilustrar los conceptos introducidos y a ser usado como un ejemplo recurrente a lo largo de la tesis. La Sección 1.4 formula el problema principal tratado en esta tesis – el desarrollo de infraestructura de software para permitir la entrega colaborativa de servicios integrados, haciendo referencia a los desafíos presentados en la Sección 1.2 y al caso de estudio en la Sección 1.3. La Sección 1.5 bosqueja la solución propuesta - Government-Enterprise Ecosystem Gateway (G-EEG), explicando los principios subyacentes además de los detalles acerca de la estructura y el comportamiento de G-EEG. La Sección 1.6 justifica las razones por las cuales G-EEG puede ser considerado una solución válida con respecto a los desafíos tecnológicos identificados en la Sección 1.2, el caso de estudio descripto en la Sección 1.3, y el problema formulado en la Sección 1.4. Por último, la Sección 1.7 presenta la contribución de la tesis y la Sección 1.8 esboza su estructura.

1.1 Gobierno Integrado

Los gobiernos alrededor del mundo se encuentran bajo presión de los ciudadanos y las empresas para que brinden un mejor gobierno. Particularmente, se les requiere que: (i) sean más abiertos y transparentes en la administración de fondos públicos y en la toma de decisiones en general; (ii) entreguen servicios públicos de alta calidad según las necesidades de los clientes y no la estructura interna del gobierno; (iii) hagan participar a los ciudadanos en el desarrollo de políticas e iniciativas que reflejen sus necesidades y expectativas; (iv) lleven a cabo la implementación de tales políticas de manera eficaz y eficiente; (v) faciliten el desarrollo económico a través de la simplificación administrativa y el apoyo a pequeñas y medianas empresas; y (vi) promuevan el estado de derecho, desarrollo social y

acceso equitativo a oportunidades [FML03]. Al mismo tiempo, se espera que los gobiernos respondan a las fuerzas de la globalización creando condiciones para fortalecer la competitividad nacional y organizativa a través del: (vii) desarrollo de la economía digital y de la sociedad de la información en su totalidad.

De manera creciente, los gobiernos usan las Tecnologías de la Información y la Comunicación (TIC) para responder a tales presiones. Aquí hay algunos ejemplos, respondiendo a diferentes tipos de presiones:

i) Proveyendo acceso en línea a información oficial, políticas y procedimientos para poder garantizar mayor transparencia y apertura en las operaciones y en los procesos de toma de decisiones del gobierno.
ii) Promocionando servicios civiles y un acercamiento a los ciudadanos, estableciendo portales de gobierno de acceso único que organicen la información y los servicios de acuerdo a eventos de vida (para ciudadanos) y episodios de negocios (para empresas), entregando servicios públicos a través de canales tradicionales (ventanillas, teléfono) y electrónicos (Internet, celulares), etc.
iii) Permitiendo retro-alimentación por parte de los usuarios de los servicios públicos a través de teléfonos de línea directa abiertos para quejas y cuestionarios online, y apoyando la participación en línea en debates de políticas a través de foros de discusión con moderadores.
iv) Estableciendo repositorios de datos centrales, aumentando el intercambio de información entre agencias en diferentes áreas y niveles funcionales de gobierno, y mejorando el apareamiento entre las necesidades de los clientes y los limitados recursos públicos ofrecidos para cumplir con ellos (por ejemplo, desempleados y vacantes), usando la colaboración a través de Internet y plataformas de intercambio de datos.
v) Promoviendo canales electrónicos para reducir cargas administrativas y requerimientos para la interacción directa entre empresas y el gobierno, y permitiendo la entrega de servicios y bienes públicos por parte del sector privado, especialmente de manera electrónica, usando sociedades público-privadas.
vi) Promoviendo el uso de aplicaciones electrónicas (e-Educación, e-Salud, e-Cultura, e-Participación, etc.) para proveer acceso a oportunidades educacionales y económicas, para aumentar la participación en actividades de la sociedad civil y en la creación de políticas públicas, y por consiguiente, aumentar la confianza en el gobierno.
vii) Desregulando el mercado de las telecomunicaciones para mejorar el acceso a bienes y servicios digitales, y aumentando el alfabetismo en TIC de la sociedad en su totalidad para promover el consumo y uso de tales bienes y servicios.

Tales respuestas crean una demanda para la mejora continua del gobierno y el uso de TIC en todo el sector público a través de un cambio organizacional subyacente. En general, la respuesta involucra una combinación de cambios tecnológicos y organizacionales. Nos referimos a la primera como Gobierno Electrónico y a la segunda como Reforma del Sector Público.

Gobierno Electrónico se refiere al uso de TIC, particularmente Internet, como una herramienta para un mejor gobierno [FML03]. Dependiendo de la definición de “mejor gobierno”, el objetivo de Gobierno Electrónico es: proveer acceso a servicios públicos eficiente, confiable y orientado al cliente; comprometer a los ciudadanos en interacciones bilaterales con el gobierno; apoyar las operaciones internas de gobierno; permitir la entrega de servicios entre agencias; y establecer portales de gobierno de acceso único. Utilizando el poder de trabajo en red de Internet, el Gobierno Electrónico conecta agencias, ciudadanos y empresas en un ambiente integrado digital, donde las políticas públicas son creadas y discutidas por diferentes interesados y donde los bienes y servicios públicos son producidos y entregados.

Definición 1: Gobierno Electrónico
Gobierno Electrónico se refiere al uso de Tecnologías de la Información y la Comunicación, en particular Internet, como una herramienta para conseguir un mejor gobierno. [FML03].

La Reforma del Sector Público se trata de la introducción deliberada de cambios a estructuras organizativas y procesos en las organizaciones del sector público para mejorar la eficiencia [UNE06]. Ejemplos de cambios a las estructuras incluyen: transformar jerarquías de agencias autónomas en redes de agencias colaborativas; permitir procesos entre agencias y su coordinación y control en niveles bajos de decisión; permitir procesamiento colaborativo de información; descentralizar y delegar la toma de decisiones en el gobierno; hacer disponibles a todo el gobierno bases de datos y de conocimiento; la subcontratación de servicios públicos al sector privado, y otros cambios más [Kot96]. Ejemplos de cambios a proceso incluyen: conectar e integrar procesos de negocios y sistemas de información; adoptar estándares

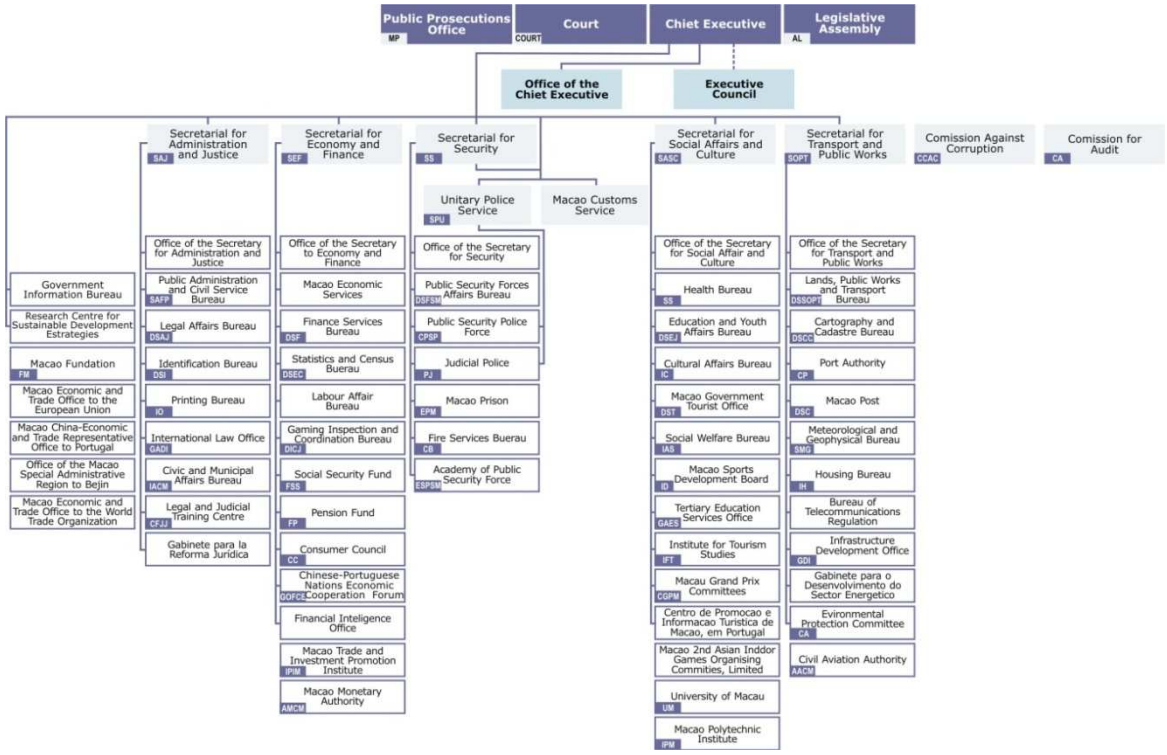
de calidad y medidas de desempeño en todo el gobierno; y construir capacidades para liderar y administrar los cambios resultantes.

Una lección aprendida, varios años después de que muchos gobiernos aceptaran al Gobierno Electrónico como una parte importante de sus iniciativas de reforma, es que la dependencia excesiva en la tecnología limita severamente los beneficios de tales iniciativas [FML03]. Mientras que publicar servicios e información de gobierno en línea crea un interés inicial en el público y un impulso para la inversión en tecnología por parte del gobierno, es la mejora lograda en base a tecnología, del gobierno mismo – su estructura, proceso, trabajo, y aún su cultura – que puede mantener este impulso. Asimismo, se ha reconocido que las mejoras en base a la tecnología, que están restringidas a agencias individuales, son de un valor limitado. Como resultado, el énfasis en la práctica de Gobierno Electrónico se enfoca en permitir la colaboración y el trabajo en red entre agencias gubernamentales, resultando en un Gobierno Integrado.

Como los gobiernos dependen tradicionalmente en autoridad y control para realizar sus operaciones, están estructurados como jerarquías. En una jerarquía, diferentes niveles y áreas del gobierno operan, por lo general, de manera independiente. Si bien existen fuertes dependencias a lo largo de líneas verticales de la jerarquía - hacia arriba (autoridad) y hacia abajo (control), mucha menos comunicación se establece horizontalmente; para que dos agencias puedan interactuar, la oficina ubicada jerárquicamente por encima de ambas debe estar involucrada.

Por ejemplo, La Figura 1 muestra la estructura vertical del Gobierno de Macao, según http://www.gov.mo. La figura representa una estructura de tres capas con el Jefe Ejecutivo, La Oficina de Prosecución Pública, La Corte y la Asamblea Legislativa en la primera capa, cinco Secretarías y doce agencias en la segunda capa, y cincuenta y seis agencias en la tercera capa.

Figura 1: Estructura Vertical de Gobierno



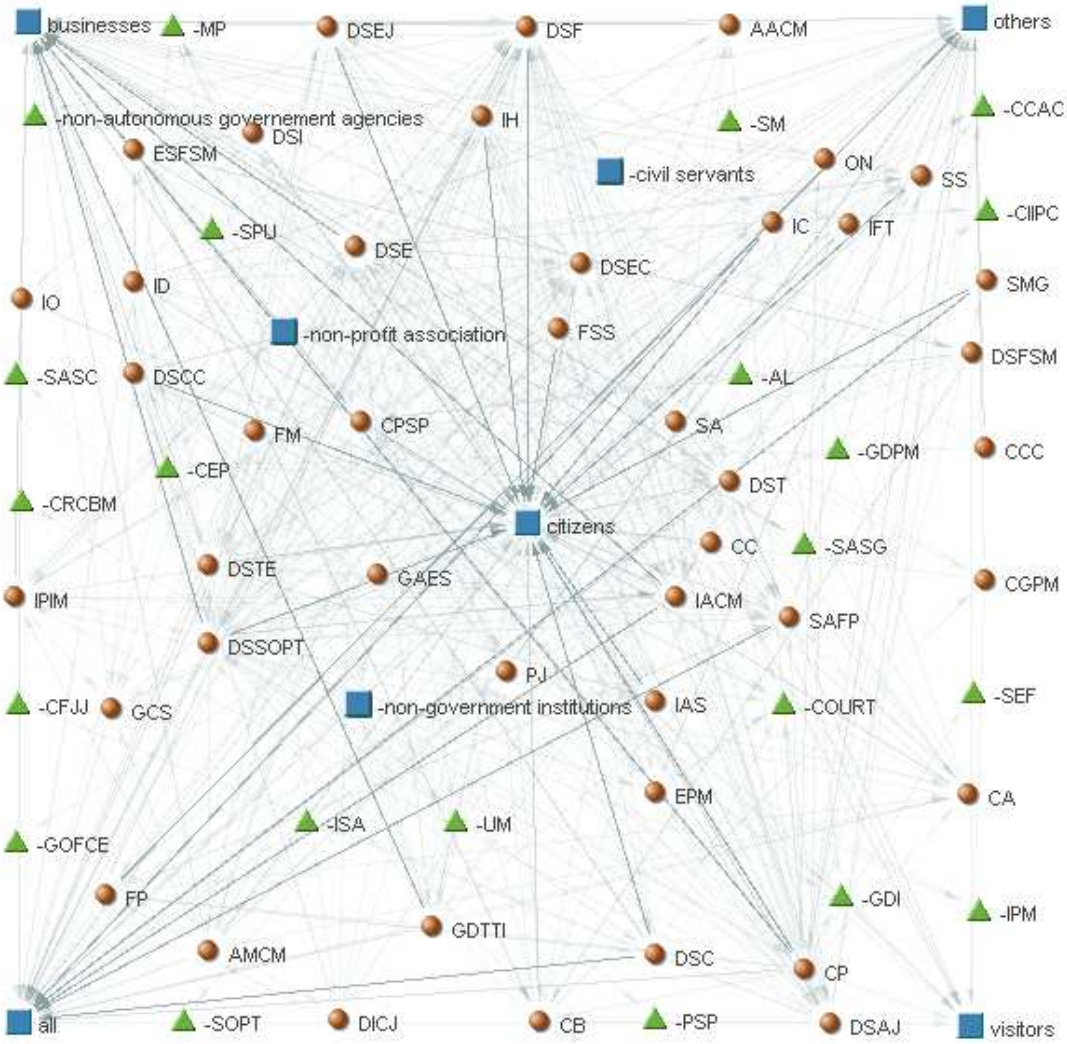
La Figura 2 muestra la estructura horizontal del mismo gobierno, representando las dependencias entre agencias para la provisión y recepción de servicios [JOE05]. La estructura está representada como un grafo, donde los nodos representan las agencias y los arcos representan la provisión de servicios. Mientras más cantidad de servicios se produzcan y reciban entre agencias, más grueso es el arco que las conecta. El gráfico también muestra receptores muy importantes, como ciudadanos y empresas.

En contraste a la estructura vertical, que es esencialmente estática – cualquier cambio en la jerarquía puede requerir un cambio en la ley, la estructura horizontal es dinámica – se pueden agregar y remover nodos, así como también se pueden agregar nuevos servicios, y los existentes pueden ser modificados. Esto refleja una naturaleza dinámica y colaborativa de la entrega de bienes y servicios públicos. El concepto resultante de Gobierno Integrado promueve la colaboración como piedra angular para mejoras en base a la tecnología en el gobierno – entre agencias de diferentes niveles y áreas de gobierno, entre organizaciones públicas y privadas, entre diferentes administraciones públicas. A continuación proponemos nuestra definición.

Definición 2: Gobierno Integrado

Gobierno Integrado es un conjunto de agencias que trabajan conjuntamente como una entidad única, generando respuestas integradas a las necesidades de la comunidad.

Figura 2: Estructura Horizontal de Gobierno



La colaboración en el gobierno ofrece claros beneficios: mejor intercambio de información entre agencias, mejor utilización de recursos, mayor participación en la redacción de políticas, mayor innovación en el trabajo público, etc. Sin embargo, el caso más prominente de colaboración en el gobierno es la entrega de servicios integrados [OPS04]. Tales servicios trascienden los límites entre las áreas y niveles del gobierno. Accedidos a través de un portal de ingreso único, y organizados en grupos, permiten a los clientes especificar un pedido – un evento de vida para los ciudadanos

y un episodio de negocios para las empresas – y obtener un servicio que satisfaga sus necesidades sin saber qué agencia o nivel de gobierno es contactado. En muchos casos, varias agencias a distintos niveles pueden estar involucradas sin que los clientes estén al tanto de esto. Por ejemplo, servicios relacionados con el nacimiento de un hijo, pueden incluir la emisión de un certificado de nacimiento, emisión de seguro social y la licencia por maternidad y protección al empleado. Aunque involucran diferentes agencias, el postulante puede acceder a todos estos servicios, como un servicio único a través del portal de gobierno de acceso único, ignorando que agencias están involucradas. La definición de Servicio Integrado se presenta a continuación.

Definición 3: Servicio Integrado
Un Servicio Integrado es un servicio público accedido a través de un contacto único, y entregado colaborativamente por varias organizaciones gubernamentales y no gubernamentales, que presenta un interfaz de organización única para los clientes.

Una presentación detallada del concepto de Gobierno Integrado y de Servicio Integrado es el objetivo del Capítulo 2.

Cumplimentar los requerimientos de Gobierno Integrado y Servicios Integrados presenta varios tipos de desafíos para las agencias. La Sección 1.2 esboza tales desafíos y la Sección 1.3 presenta un ejemplo real de un Servicio Integrado.

1.2 Desafíos para el Gobierno Integrado

Esencialmente, el Gobierno Integrado promueve la colaboración como piedra angular de las mejoras en base a la tecnología en el gobierno. Permite a las agencias interactuar según la estructura horizontal del gobierno, mientras dependen de la estructura vertical para la autoridad y el control. La naturaleza y extensión de interacciones entre agencias de gobierno pueden diferir sustancialmente, desde la provisión de servicios, a través del intercambio de datos, hasta la transferencia de mejores prácticas entre agencias.

Sin embargo, el cambio de contexto de las operaciones llevadas a cabo por una sola agencia a las operaciones realizadas en conjunto por varias agencias colaboradoras puede presentar desafíos. Puede resultar difícil extender lo que una agencia pueda lograr por sí sola, considerando el marco legal, acuerdos financieros, cultura de trabajo, estructura organizativa y preparación tecnológica, a un contexto de inter-agencias. Por ejemplo, sin un marco legal apropiado que reconozca los documentos electrónicos como equivalentes a los documentos en papel, las agencias no pueden depender del intercambio electrónico de información con otras agencias como base para emitir permisos y licencias. Careciendo de acuerdos financieros apropiados, las agencias dudarían en participar en proyectos inter-agencias, por la incertidumbre en cómo compartir gastos, riesgos y beneficios. En la cultura laboral que solamente recompensa desempeños individuales, las agencias no están incentivadas para colaborar y compartir recompensas con otras agencias. Cuando dos agencias no pueden acordar en la coordinación estructural y la división de responsabilidades para actividades conjuntas, les resulta difícil ejecutar procesos inter-agencias y entregar servicios integrados. Del mismo modo, la falta de infraestructura tecnológica compatible puede afectar la colaboración entre agencias que no pueden hacer que sus aplicaciones de misión crítica se comuniquen entre ellas.

En general, la implementación de Gobierno Integrado tiene que enfocarse en varios desafíos legales, financieros, sociales, organizacionales y tecnológicos, como se describe a continuación:

1) *Desafíos Legales* – Gobierno Integrado requiere que documentos y procesos electrónicos sean legalmente equivalentes a procesos y documentos basados en papel. Esto incluye la protección legal de registros electrónicos y actividades gubernamentales basadas en ellos. Asimismo, es un gran desafío la falta de reglas, regulaciones y procedimientos establecidos para reglamentar la colaboración entre agencias y entre organizaciones públicas y no públicas. Esto incluye incertidumbre acerca de adoptar estándares técnicos para el intercambio de información y la interoperabilidad.

2) *Desafíos Financieros* – Gobierno Electrónico desafía los principios tradicionales de la financiación en la administración pública, donde la financiación está destinada a agencias individuales, registrada a través del desempeño individual, y basada en ciclos presupuestarios anuales y bianuales. En contraste, Gobierno Electrónico depende principalmente de proyectos inter-agencias y financiaciones multi-anuales, sus resultados y

beneficios son generalmente expresados en términos no financieros, y su éxito depende en gran parte del desempeño colectivo.

3) *Desafíos Sociales* – La brecha digital es el principal desafío social para implementar Gobierno Electrónico. La pobreza, el acceso limitado a Internet y a otros canales electrónicos, y la baja alfabetización en TIC significan que grandes segmentos de la sociedad no pueden beneficiarse de servicios en línea. La lenta adopción de servicios en línea, la falta de demanda social para la innovación y mejoramiento del gobierno, y la falta de confianza en procesos gubernamentales pueden limitar el apoyo público, y por consiguiente, restringir las inversiones en Gobierno Electrónico. Finalmente, un número de desafíos sociales se originan dentro del gobierno, particularmente, la resistencia al cambio y la falta de alfabetismo en TIC de los funcionarios.

4) *Desafíos Organizacionales* – El desafío organizacional principal es permitir la colaboración entre agencias, entre sectores públicos y privados, y entre diferentes sistemas de administración pública, como es requerido por un Gobierno Integrado. Tales colaboraciones traen un cambio cualitativo al trabajo de las organizaciones gubernamentales, en contra de las estructuras jerárquicas y el modo de operaciones de comando y control. En general, los funcionarios situados en niveles bajos de las jerarquías de gobierno no tienen acceso a información y no pueden tomar decisiones, limitando severamente la colaboración requerida para un Gobierno Integrado. Específicamente, identificamos cinco tipos de desafíos organizacionales para implementar Gobierno Integrado. El primero (01) trata de transformar el gobierno de jerarquías organizativas a redes organizativas, a través del empoderamiento del personal y el acceso a información. Los cuatros desafíos restantes se refieren a varios escenarios de integración en donde diferentes organizaciones entregan de manera colaborativa un servicio a ciudadanos y empresas: entre agencias de diferentes áreas funcionales de gobierno (02), entre agencias ubicadas en diferentes niveles de gobierno - nacional, provincial, municipal, etc. (03), entre sectores públicos y privados (04), y entre diferentes sistemas de administración pública (05). Cada escenario de integración da origen a un conjunto específico de desafíos organizacionales.

5) *Desafíos Tecnológicos* – Los desafíos tecnológicos están ubicuos en los proyectos de Gobierno Electrónico. Por el caso de la tesis, identificamos un conjunto de diez desafíos generales: (T1) construcción de portales de gobierno de acceso único para consolidar la información y los servicios provistos por diferentes agencias; (T2) coordinación de la ejecución de procesos inter-agencias; (T3) entrega de servicios en línea en conformidad con las políticas y los estándares predefinidos; (T4) integración de aplicaciones utilizadas en diferentes agencias para garantizar que tales aplicaciones pueden inter-operar tanto en niveles sintácticos (T5) como semánticos (T6); (T7) subcontratación flexible de servicios públicos a organizaciones del tercer sector; (T8) mantenimiento de un ecosistema dinámico de organizaciones públicas y no públicas para la entrega de servicios públicos; (T9) entrega de servicios públicos a través de múltiples canales; y (T10) fiabilidad del software de gobierno.

Una presentación detallada de los desafíos enfrentados para la implementación de Gobierno Integrado es el objetivo del Capítulo 3.

1.3 Ejemplo de Gobierno Integrado

El ejemplo está basado en un caso real de un servicio público entregado por Civic and Municipal Affairs Bureau (IACM), una agencia del Gobierno de Macao. IACM es responsable de emitir varios tipos de licencias empresariales: para importar y vender productos, para establecer actividades relacionadas con alimentos y bebidas, para hacer publicidades en lugares públicos, etc. Para que IACM pueda emitir licencias, debe depender de otras agencias para llevar a cabo inspecciones, proveer opiniones técnicas acerca de una aplicación, controlar conformidad con leyes y regulaciones relevantes, etc. En general, un servicio de licencias acepta aplicaciones y las procesa a través de la colaboración entre varias agencias gubernamentales antes de tomar una decisión sobre si la aplicación debe ser aceptada o rechazada, y en base a esto, responder al postulante.

El ejemplo siguiente describe el procedimiento para solicitar a IACM una licencia para establecer un negocio de comidas y bebidas [IACM04].

Ejemplo 1: Licencias para Negocios de Comidas y Bebidas – Procedimiento

El proceso de aplicación para la emisión de licencias para negocios de comidas y bebidas involucra seis etapas:

1) *Pre-Solicitud* – Durante esta etapa, el postulante reúne información acerca de procedimientos de postulación, y solicita asesoramiento sobre cómo los procedimientos se aplican en su caso. El postulante también puede pedir hasta tres reuniones técnicas, dependiendo de la complejidad del caso. Luego de reunir todos los documentos necesarios y de llenar los formularios de solicitud, el postulante está listo para enviar su solicitud.

2) *Solicitud* – Esta fase involucra la presentación de un formulario de postulación y los documentos necesarios por parte del postulante. Los documentos necesarios son: una foto del establecimiento; reporte escrito del registro de propiedad; obras de construcción de interiores; planos de ubicación; planos de construcción; planos del sistema de suministro de agua; planos de desagüe y cloacas; proyecto de modificación; memorándum de proyecto y el diagrama de cableado de las principales centrales eléctricas; planos de control de incendio y sistemas de prevención de incendios; declaración de responsabilidad del Planificador del Proyecto, del Director del Proyecto y del Ingeniero del Proyecto; y póliza de seguro para obras de construcción. La presentación puede involucrar varias etapas.

3) *Evaluación de Completitud* – Esta fase comprende la comprobación de que esté completo el formulario de postulación y todos los documentos necesarios, y la notificación al postulante acerca de documentos que falten, si es el caso.

4) *Evaluación* – Esta fase involucra el pedido de opiniones y asistencia de otras agencias, por ejemplo:

   a) De Labour and Employment Bureau (DSAL) acerca de la seguridad del ambiente laboral y las condiciones sanitarias del establecimiento, cuando se emplea a más de 30 personas;
   b) De Lands, Public Works and Transport Bureau (DSSOPT) acerca de infraestructura y planos de construcción;
   c) De Cultural Affairs Bureau (IC) acerca de ubicación, paisajismo de exteriores, aspecto externo, reconstrucción y renovación interna, cuando se trata de un establecimiento ubicado dentro de una zona de preservación de patrimonio cultural;
   d) De Fire Services Bureau (CB) solicitando la inspección edilicia para comprobar las medidas de prevención de incendios; y
   e) De Health Bureau (SS) solicitando una inspección para comprobar las condiciones sanitarias.

   Una vez que los pedidos son enviados a estas agencias, el proceso espera hasta recibir todas las respuestas, las cuales pueden incluir solicitudes para coordinar inspecciones in situ con el postulante, en particular por CB y SS. IACM hace un seguimiento para coordinar la fecha de las inspecciones y notificar a CB y SS acerca de las mismas. Una vez realizadas las inspecciones, IACM recibe las opiniones restantes de CB y SS.

5) *Toma de Decisión* – Esta fase involucra la toma de decisión por las autoridades de la agencia para decidir si la postulación debe ser aceptada o rechazada, en base a las opiniones recibidas anteriormente.

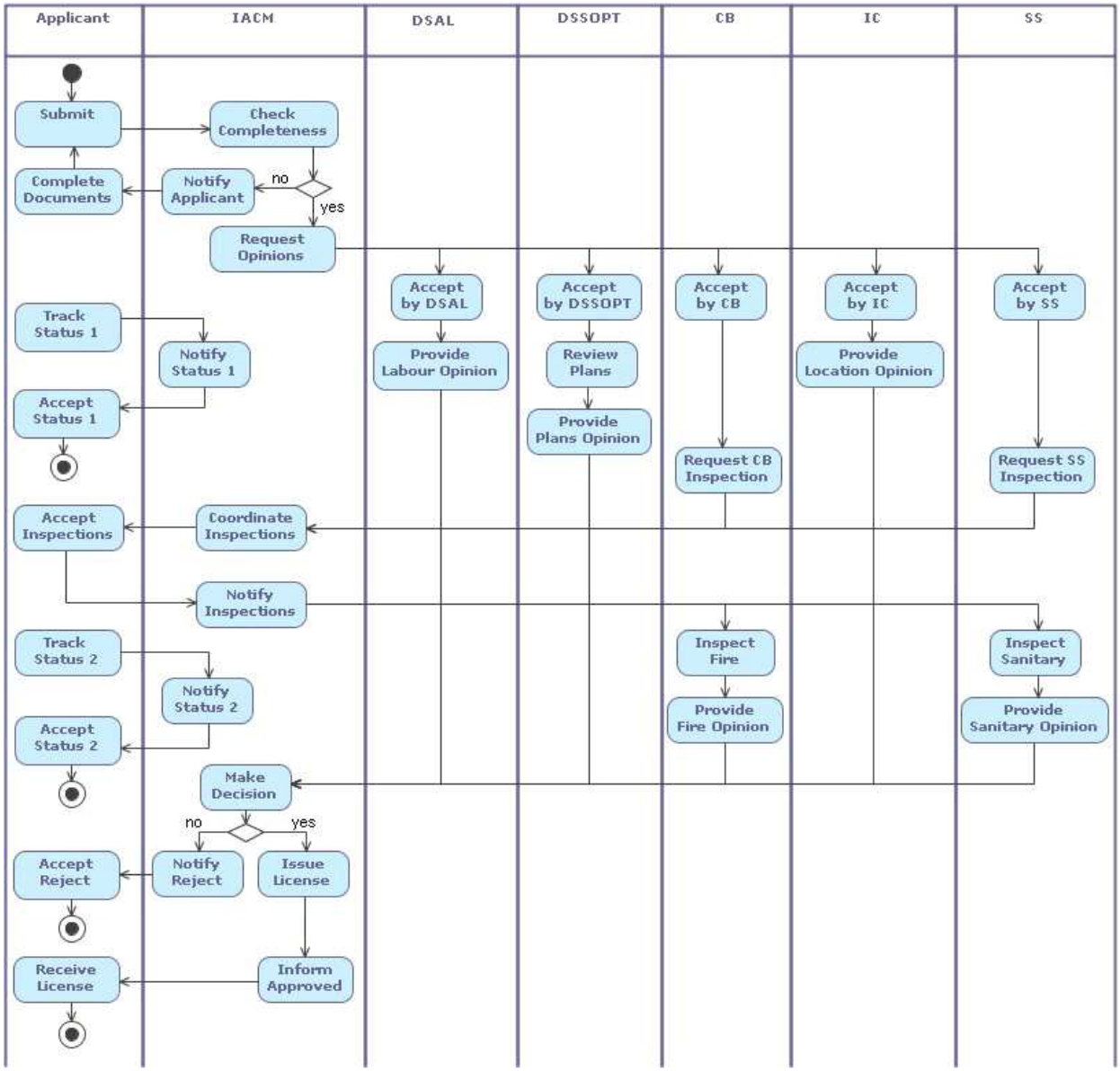
6) *Seguimiento* – Esta fase involucra la notificación al postulante acerca de la decisión. Tras una decisión positiva, IACM emite la licencia, le informa al postulante de su disponibilidad, y le entrega la licencia al postulante.

La Figura 3 describe el proceso de negocios que corresponde al procedimiento descripto. La figura utiliza la notación gráfica de los diagramas de actividad de UML (Unified Modelling Language). Como parte del proceso, el postulante solicita rastrear la postulación 2 veces y en ambos casos, IACM notifica al postulante acerca de su estado.

Para asegurar que el Servicio de Licencias para Negocios de Comidas y Bebidas sea implementado como un servicio integrado, se deben cumplir con los siguientes requerimientos: (1) Solicitud y Seguimiento - un postulante debería poder presentar la solicitud de licencia, presentar los documentos necesarios, así como rastrear el estado de su solicitud a través de un portal de gobierno de acceso único; (2) Entrega Integrada - el postulante no debería estar al tanto de, o al menos afectado por, la participación en el proceso de seis agencias diferentes – IACM, DSAL, DSSOPT, CB, IC y SS; (3) Integración - las agencias participantes deberían estar listas para colaborar a fin de entregar el servicio, en conjunto; (4) Interoperabilidad - el software usado por estas agencias debería permitirles intercambiar e interpretar

correctamente la información; (5) Coordinación - el proceso de negocio subyacente debería ser acordado y coordinado entre todas las agencias involucradas; y (6) Flexibilidad - los cambios en el proceso de negocios debido a, por ejemplo la subcontratación o la actualización de las políticas, deberían ser transparentes para el postulante.

Figura 3: Servicio de Licencias – Flujo de Trabajo



1.4 Enunciado del Problema

Siguiendo los antecedentes (Sección 1.1), los desafíos (Sección 1.2), y el caso de estudio (Sección 1.3), esta sección formula el principal problema tratado por esta tesis – implementación de Gobierno Integrado.

El enfoque principal de Gobierno Integrado y el desafío principal para su implementación es permitir le entrega colaborativa de Servicios Integrados a ciudadanos, empresas, sociedad civil y otras ramas de gobierno a través de redes organizativas compuestas por organizaciones gubernamentales y no gubernamentales. Si bien toda solución exhaustiva debe enfocarse en los desafíos legales, financieros, sociales, organizacionales y tecnológicos, como se explica en la Sección 1.2, esta tesis se enfoca principalmente en los desafíos tecnológicos. El problema se formula de la siguiente manera:

Problema
Construir una plataforma de comunicación y coordinación, con el modelo y la teoría subyacentes, para facilitar el establecimiento, operación y evolución de redes organizativas, capaces de entregar servicios públicos integrados. Esta plataforma debería cumplir con los siguientes requerimientos: R1) Conectar software y personas trabajando dentro de las organizaciones miembros a través de un ambiente de trabajo en red, para permitir el intercambio, interpretación y procesamiento de información. R2) Apoyar procesos de negocios inter-organizacionales a través de los cuales las organizaciones miembros puedan de manera conjunta, entregar servicios públicos a clientes. R3) Permitir el monitoreo de la entrega de servicios públicos, los procesos subyacentes y el comportamiento de organizaciones que participan en ellos, en base a políticas y regulaciones prescriptas por el gobierno. R4) Permitir la evolución de redes organizacionales a través de varios cambios en su entorno, afectando la membrecía, contratos, regulaciones y servicios provistos a través de tales redes. Adicionalmente, la plataforma debería ser validada demostrando su capacidad para: o enfrentar los desafíos técnicos T1 a T10 (Sección 1.2) y o apoyar la implementación del Servicio de Licencias (Sección 1.3).

1.5 Solución Propuesta

La tesis propone G-EEG (Government-Enterprise Ecosystem Gateway) como solución para el problema formulado en la Sección 1.4. G-EEG es una plataforma de comunicación de alto nivel que soporta la ejecución de procesos de negocios inter-organizacionales. Permite la construcción, aplicación, y evolución de estructuras de comunicación complejas para el intercambio asincrónico de mensajes en varios contextos de aplicación, como por ejemplo, la entrega colaborativa de servicios públicos. Se diferencia de otras soluciones por su habilidad para crear y configurar dinámicamente las estructuras de comunicación, y proveer un mecanismo de extensión, que permite agregar la funcionalidad requerida por aplicaciones que usan la mensajería, integrando ésta funcionalidad con los servicios básicos, todo bajo la misma solución.

Esta sección introduce brevemente G-EEG. Presenta sus atributos (Sección 1.5.1), estructura (Sección 1.5.2) y comportamiento (Sección 1.5.3). La presentación en detalle de G-EEG es el objetivo de los Capítulos 4 (Fundamentos) y 5 (Implementación).

1.5.1 Atributos de G-EEG

G-EEG se compone de tres conceptos principales: núcleo del framework de mensajería (G-EEG-CORE), un repositorio de extensiones para proveer funcionalidad enriquecida por encima del núcleo (G-EEG-EXTEND), y un ambiente de desarrollo para construir nuevas extensiones (G-EEG-DEVELOP). El objetivo de esta estructura es garantizar que G-EEG sea minimalista, extensible, dinámico y confiable:

1) *Minimalista* – G-EEG-CORE se construye con la menor cantidad posible de conceptos – mensajes, miembros, canales, propietarios y suscriptores, y su objetivo es permitir el servicio de mensajería básico – intercambio asincrónico de mensajes entre miembros registrados a través de canales creados y suscriptos dinámicamente.
2) *Extensible* – G-EEG-EXTEND provee a las aplicaciones funcionalidad de mensajería adicional, usualmente requerida por ellas, a través de un repositorio de extensiones independientes (horizontales) y dependientes de los procesos (verticales).

3) *Dinámico* – G-EEG-CORE y G-EEG-EXTEND contienen provisiones para que G-EEG pueda responder a los cambios en el ambiente de operación: las estructuras de comunicación pueden evolucionar con el paso del tiempo y las extensiones pueden ser habilitadas, configuradas, y deshabilitadas según las necesidades de las aplicaciones.
4) *Confiable* – G-EEG está basado en modelo formal que permite el análisis de propiedades importantes, permitiendo el desarrollo riguroso de nuevas extensiones y guiando el desarrollo de software de G-EEG.

1.5.2 Estructura de G-EEG

G-EEG-CORE – Un framework de mensajería de tiempo de ejecución que permite el intercambio asincrónico de mensajes entre miembros registrados a lo largo de canales creados y suscriptos dinámicamente. Los miembros pueden ser personas, software u organizaciones que lleven a cabo la mensajería como parte de un proceso inter-organizacional más extenso, como por ejemplo, la entrega de servicios integrados. G-EEG-CORE ofrece lo mínimo indispensable en cuanto a funcionalidad para sus usuarios – habilidad para registrarse y des-registrarse como miembros, crear y destruir canales, suscribirse y des-suscribirse a canales, y enviar y recibir mensajes.

G-EEG-EXTEND – Un repositorio de extensiones horizontales y verticales y un mecanismo para habilitarlas dinámicamente sobre G-EEG-CORE. Las extensiones horizontales son independientes de cualquier proceso que requiera el intercambio de mensajes. Por ejemplo: (H1) generación de la historia de mensajes que circulan a través de un canal, (H2) validación de mensajes enviados a través de un canal con respecto a un formato dado; (H3) transformación de mensajes enviados a través de un canal de un formato a otro; (H4) cifrado y des-cifrado de mensajes a través de un canal; (H5) control de que sólo miembros autorizados puedan usar servicios de mensajería; (H6) información provista acerca de miembros o canales; y (H7) uniendo miembros en alianzas para llevar a cabo el envío de mensajes en conjunto. Las extensiones verticales son específicas para procesos, por ejemplo: (V1) garantizando que el flujo de los mensajes a través de algunos canales esté en conformidad con los procesos de negocios predefinidos; (V2) verificando si el intercambio de mensajes está en conformidad con ciertas políticas de comunicación; (V3) permitiendo la composición de canales existentes en canales más complejos; y (V4) determinando la ubicación de un mensaje en tránsito a través de un canal complejo. La Tabla 1 presenta la lista de estas extensiones.

Tabla 1: Ejemplo de Extensiones Horizontales y Verticales de G-EEG

H1	Auditoría	V1	Orden
H2	Validación	V2	Puntualidad
H3	Transformación	V3	Composición
H4	Cifrado	V4	Seguimiento
H5	Autenticación		
H6	Localización		
H7	Alianza		

G-EEG DEVELOP – Un marco de desarrollo para especificar, diseñar y verificar rigurosamente las extensiones de mensajería, construido sobre la funcionalidad central ofrecida por G-EEG CORE y contribuyendo a G-EEG-EXTEND con nuevas extensiones. Para poder facilitar el desarrollo riguroso de extensiones, G-EEG-DEVELOP permite su descripción en diferentes niveles de abstracción:

- *Nivel Abstracto* – En este nivel, el comportamiento de los servicios de mensajería está descripto en términos de los cambios que se espera que causen en el estado global del Gateway (compuesto por los estados de los miembros y canales), sin determinar cómo se realizan tales cambios. Este nivel permite especificar el comportamiento esperado.
- *Nivel Concreto* – En este nivel, los servicios de mensajería involucran un intercambio concreto de mensajes entre miembros a lo largo de canales creados y suscriptos dinámicamente, produciendo cambios en los estados locales de miembros y canales, y, por consiguiente, cambios al estado global. Este nivel describe el comportamiento de la mensajería a medida que va sucediendo.

Cambios de estado a nivel global o local, no solo involucran el envío y recepción de mensajes por parte de los miembros, sino también la evolución del Gateway mismo: (i) los miembros pueden suscribirse o des-suscribirse al Gateway en cualquier momento; (ii) los canales pueden ser creados o destruidos dinámicamente por sus miembros,

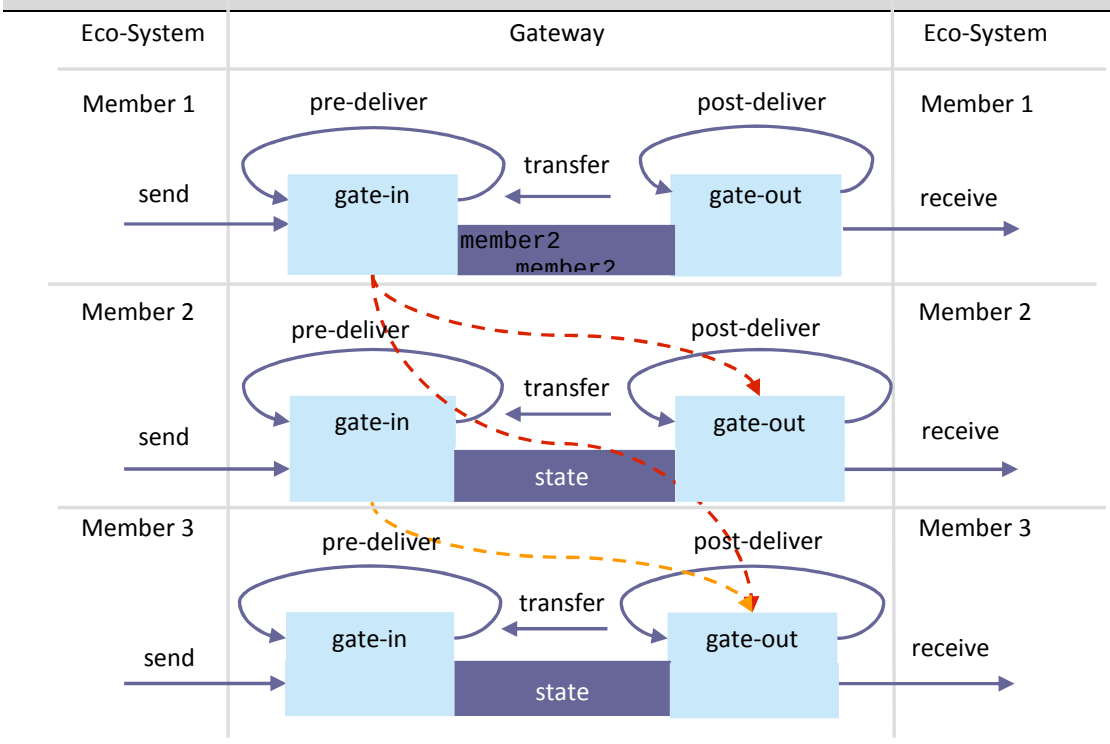
(iii) los canales pueden ser suscriptos o des-suscriptos dinámicamente por los miembros, y (iv) funcionalidad de mensajería adicional puede ser habilitada o deshabilitada en canales individuales o a través de miembros.

Vinculando los niveles abstractos y concretos se crea una oportunidad para la verificación – probar que una extensión de mensajería implementa correctamente su especificación. Sin embargo, un requerimiento necesario para esto es la existencia de una semántica formal que sustente las descripciones en ambos niveles, haciendo posible que se relacionen. G-EEG-DEVELOP provee semánticas formales que no sólo cubren estos dos niveles, sino también a G-EEG-CORE. Efectivamente, los servicios de mensajería básicos de G-EEG-CORE son especificados, implementados y verificados en la misma manera que los servicios de mensajería extendidos de G-EEG-EXTEND.

1.5.3 Comportamiento de G-EEG

Durante el desarrollo de G-EEG, abstracción fue el principio conductor usado para: resolver los desafíos tecnológicos de Gobierno Integrado (Sección 1.2), proveer una solución al problema principal (Sección 1.4), realizar los atributos que apuntalan a G-EEG (Sección 1.51.), permitir el desarrollo riguroso (Sección 1.5.2), y simultáneamente ocultar la complejidad de G-EEG a desarrolladores y aplicaciones que lo usan. Esta sección presenta el comportamiento de G-EEG en tres niveles relacionados de abstracción, pero a su vez incrementalmente más concretos: abstracto (Figura 4) intermedio (Figura 5) y concreto (Figura 6).

Figura 4: Comportamiento de G-EEG – Abstracto



En el nivel abstracto, G-EEG comprende el Gateway y el ecosistema de miembros que interactúan al enviar y recibir mensajes a través de él. A cada miembro se le asigna: gate-in – un contenedor para los mensajes enviados por el miembro; gate-out – un contenedor para los mensajes recibidos por un miembro; y el estado local.

Los siguientes pasos se llevan a cabo con el envío de un mensaje por parte de un miembro, hasta la recepción del mismo por parte de otro.

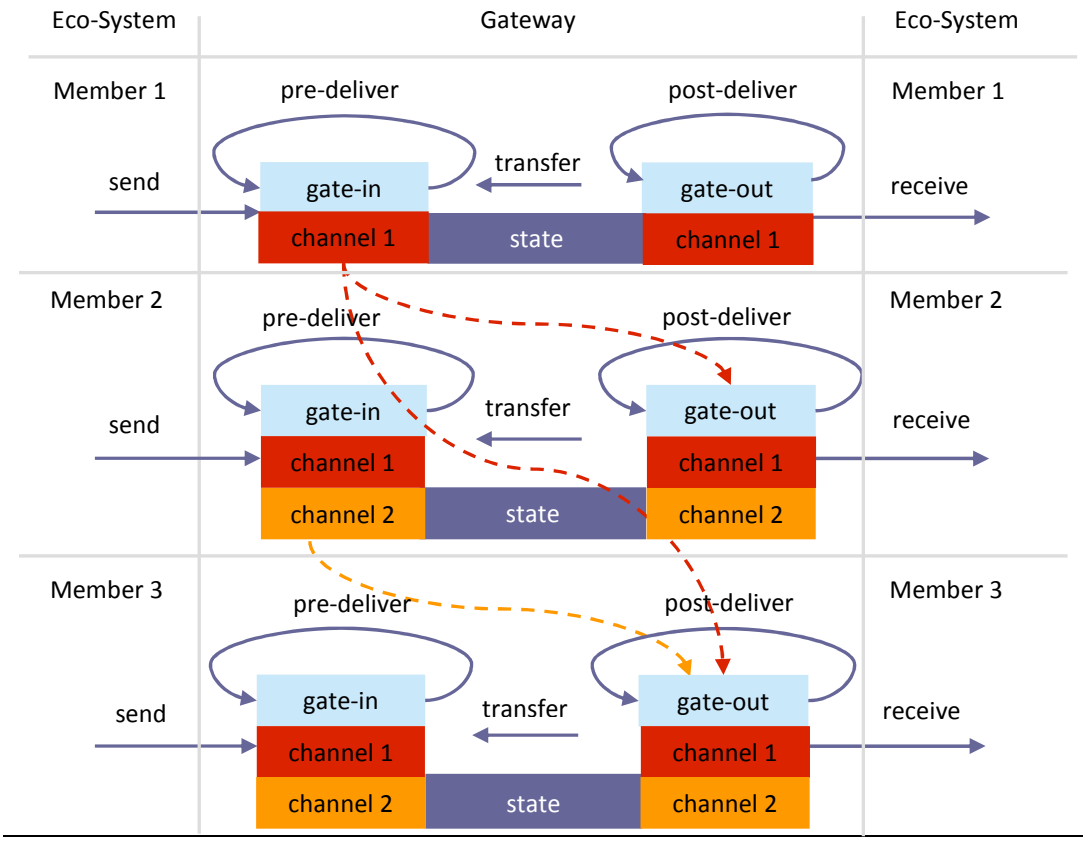
1) El mensaje es insertado en la estructura gate-in del emisor y procesado por la función de pre-entrega, posiblemente cambiando el contenido del mensaje, los receptores deseados y el estado local del emisor.
2) El mensaje es distribuido desde la estructura gate-in del emisor a la estructura gate-out de todos los receptores.

3) Para cada miembro receptor, el mensaje recibido es procesado por la función post-entrega, posiblemente afectando su contenido y el estado local del receptor, y posteriormente es recibido por el miembro.
4) En vez de recibir un mensaje como en (3), el mensaje es transferido de la estructura gate-out a la estructura gate-in del receptor, como preparación para posteriores entregas, en cuyo caso, se vuelve a ejecutar el paso (1).

La Figura 4 presenta un escenario particular a través del cual el Miembro 1 entrega un mensaje a los Miembros 2 y 3 – el mensaje es insertado en el gate-in del Miembro 1 y posteriormente trasladado a los gate-out de los Miembros 2 y 3. Luego de recibir este mensaje, el Miembro 2 se lo transfiere al Miembro 3 – el mensaje es trasladado desde el gate-out al gate-in del Miembro 2, y posteriormente al gate-out del Miembro 3. Cuando el Miembro 3 recibe ambos mensajes, los mismos son efectivamente removidos del gate-out. Mientras el mensaje se traslada entre los gate-in y gate-out, su contenido y los estados locales de los miembros pueden cambiar a través de funciones de pre-entrega, post-entrega y transferencia.

En el nivel actual, G-EEG omite donde deberían entregarse los mensajes. La conexión entre Miembro 1, y Miembro 2 y Miembro 3 (primer mensaje) y entre Miembro 2 y Miembro 3 (segundo mensaje) se decide implícitamente en base al contenido del mensaje, y no por el mecanismo de mensajería del Gateway. La decisión principal cuando se va bajando en el nivel de abstracción es hacer explícito este mecanismo de entrega. Ver Figura 5.

Figura 5: Comportamiento de G-EEG – Intermedio

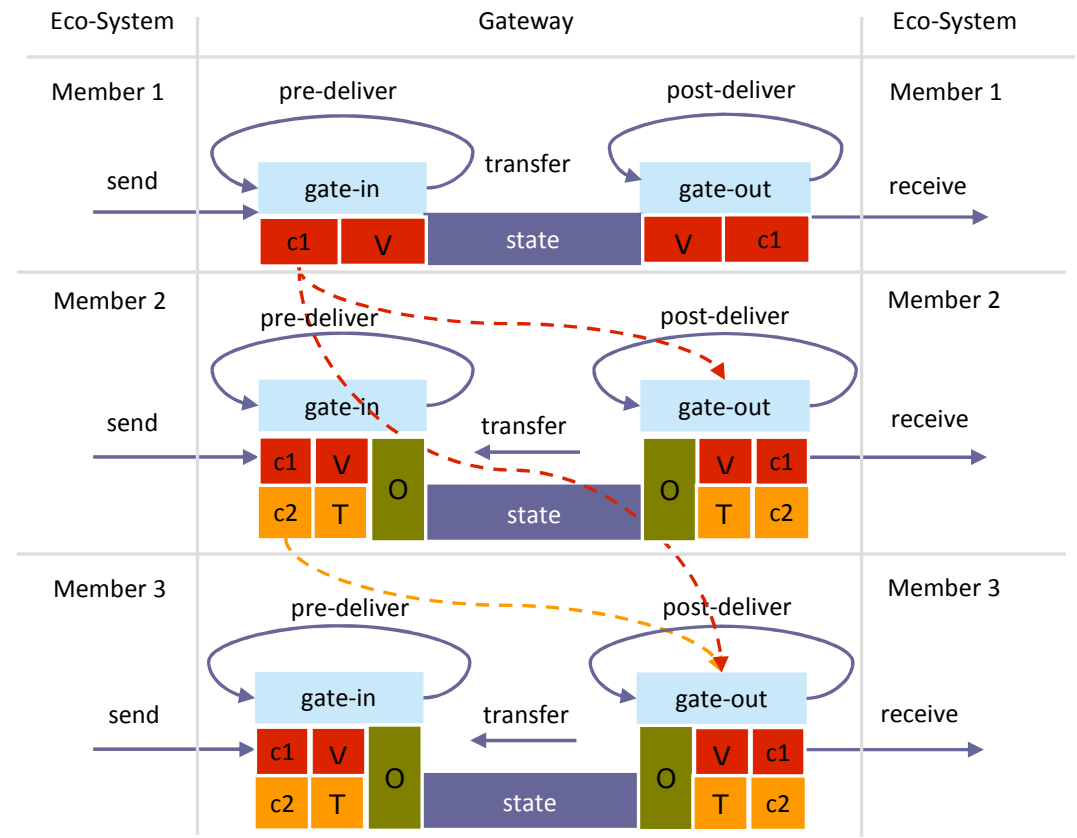


A fin de explicitar el mecanismo de entrega, se introduce el concepto de canal – una conexión lógica entre miembros. Ciudadano de primera clase para el Gateway, un canal puede ser creado y destruido por los miembros, y suscriptos y des-suscriptos por ellos. El miembro que crea un canal se convierte en su propietario. Cuando un suscriptor envía un mensaje a través de un canal, el mismo es entregado al resto de los suscriptores. La existencia de un canal, al igual que su conjunto de suscriptores, puede cambiar con el paso del tiempo.

La Figura 5 muestra la estructura de canal para el escenario en la Figura 4, donde el primer mensaje es enviado por el Miembro 1 al Canal 1, suscripto por los Miembros 1, 2 y 3, mientras que el segundo es enviado por el Miembro 2 al Canal 2, suscripto por los Miembros 2 y 3. Luego de recibir el mensaje a través del Canal 1, el Miembro 2 lo envía al Canal 2.

En el nivel actual se mantiene oculto cómo los mensajes son procesados y cómo la funcionalidad de mensajería, más allá del simple envío y recepción de mensajes, puede ser integrada en el Gateway. En el nivel concreto, tal funcionalidad es implementada e integrada al Gateway a través de extensiones. Ver Figura 6.

Figura 6: Comportamiento de G-EEG – Concreto



Las extensiones son principalmente habilitadas en canales para implementar funcionalidad sobre la transferencia de mensajes a través de un canal, por ejemplo validando un mensaje con respeto a un formato dado, cifrando y descifrando un mensaje, etc. Lo que permite tal proceso es el hecho de que un mensaje es enviado a un canal enviándolo primero al propietario de ese canal, quien posteriormente puede invocar funcionalidad adicional. Varias extensiones pueden ser habilitadas a un canal, y el propietario puede invocarlas a todas y coordinar una respuesta conjunta. Las extensiones también pueden ser habilitadas a miembros, por ejemplo, una alianza de miembros (H7). Un miembro especial, representando a todos los miembros de la alianza, es creado para que implemente la extensión.

Las extensiones pueden ser horizontales (independientes de los procesos) o verticales (dependientes de los procesos). La distinción, en el lenguaje de canales, es que las extensiones horizontales se refieren a un solo canal – afectando la transferencia de mensajes a través de ese canal, mientras que las extensiones verticales se pueden referir a varios canales – coordinando la transferencia de mensajes a través esos canales.

La Figura 6 representa el escenario de la Figura 5 involucrando las extensiones. Muestra que el Canal 1 (c1) está configurado con la extensión de Validación (V), mientras que el Canal 2 (c2) está configurado con la extensión de Transformación (T). Adicionalmente, muestra que la extensión vertical Orden (O) controla el flujo de los mensajes que circulan por los Canales 1 y 2.

1.6 Evaluación de la Solución

Basado en el resumen de la solución propuesta en la Sección 1.5, esta sección evalúa G-EEG como solución al problema formulado en la Sección 1.4. Con este fin, explicamos:

1) Como G-EEG cumple con los requerimientos R1 a R4 – Sección 1.6.1;
2) Como G-EEG resuelve los desafíos tecnológicos T1 a T10 – Sección 1.6.2, y
3) Como G-EEG soporta la implementación del servicio de licencias – Sección 1.6.3

Posteriormente, la Sección 1.7 explica la contribución de la tesis mostrando como G-EEG mejora las soluciones existentes y avanza el estado del arte en el área. La evaluación detallada de G-EEG es el objetivo del Capítulo 6.

1.6.1 Evaluación de G-EEG – Problema

Refiriéndose al problema formulado en la Sección 1.4, es posible justificar que G-EEG efectivamente cumple con las necesidades para una "plataforma de comunicación y coordinación de alto nivel, con el modelo y la teoría subyacente, para facilitar el establecimiento, operación y evolución de redes organizativas, capaces de entregar servicios públicos integrados":

Primero, G-EEG soporta todo el ciclo de vida de las redes organizativas:

1) *Establecimiento* – G-EEG debe contener dos miembros predeterminados – administrador (para administrar la membrecía) y un visitante (para registrar a nuevos miembros) conectados por un solo canal. Desde este estado inicial, G-EGG soporta el establecimiento de redes organizativas completas con unos pocos servicios básicos.
2) *Operación* – Un miembro de G-EEG puede ser una persona, un software o una organización entera. Una vez registrado, el miembro obtiene acceso a todos los servicios de mensajería provistos por G-EEG, además del acceso a otros miembros a través de canales.
3) *Evolución* – No solo los miembros pueden comunicarse entre ellos a través de G-EEG, sino que también pueden reorganizar la red misma, con nuevos socios, socios existentes desvinculándose, nuevos canales creados, canales existentes destruidos, y miembros suscribiéndose a, o des-suscribiéndose de canales existentes, todos disponibles como servicios de mensajería.

Segundo, G-EEG-DEVELOP cumple con la necesidad del modelo y la teoría subyacente – formula servicios de mensajería centrales y permite la especificación, implementación y verificación de extensiones de mensajería basadas en esta formalización.

Tercero, la habilidad de redes organizativas facilitada por G-EEG para la "entrega de servicios públicos integrados" es demostrada en la Sección 1.6.3, enfocándose en la entrega integrada de servicios de licencias (como en la Sección 1.3), a través de G-EEG.

Cuarto, G-EEG cumple los requerimientos R1 a R4. Con respecto a R1, basado en el esquema de membrecía flexible, G-EEG "conecta software y personas trabajando dentro de las organizaciones miembros asociadas" y, dependiendo de varias extensiones habilitadas a canales y del procesamiento independiente de mensajes por parte de los miembros, permite el "intercambio, interpretación y procesamiento de información". Con respecto a R2, G-EEG "apoya a procesos de negocios inter-organizacionales" proveyendo un ambiente para el intercambio regulado de mensajes entre las organizaciones a través de canales definidos por el usuario, y permitiendo la habilitación de extensiones verticales para coordinar la mensajería a través de canales. Con respecto a R3, G-EEG permite monitorear si la comunicación entre los miembros procede de acuerdo a procesos y políticas predefinidas. Esto se puede hacer, por ejemplo, a través de extensiones verticales como V1 – Orden. Con respecto a R4, por diseño, G-EEG ayuda a "la evolución de redes organizativas" a través de facilitar la introducción de varios tipos de cambios en su entorno.

1.6.2 Evaluación de G-EEG – Desafíos

Todos los desafíos tecnológicos T1 a T9 requieren comunicación entre agencias y entre sectores públicos y privados. Los servicios de mensajería básicos ofrecidos por G-EEG-CORE proveen el marco subyacente para unir organizaciones, por lo tanto soportando la resolución de tales desafíos. Al mismo tiempo, desafíos individuales pueden ser resueltos a través de una combinación particular de extensiones de G-EEG-EXTEND. Por ejemplo:

- *Acceso Único (T1)* – Supongamos que un miembro es registrado para representar al portal de acceso único y un miembro es registrado para cada organización que provee servicios a través del portal, con canales dedicados conectando el portal con los proveedores de servicios. Canales adicionales unen a los proveedores de servicios con otras organizaciones. Tres extensiones horizontales son usadas: Auditoría (H1) para evaluar la cantidad de servicios requeridos a través del portal; Validación (H2) para garantizar que las aplicaciones reciban solicitudes válidas; y Cifrado (H4) para proteger a la información intercambiada de ser interceptada a escondidas.

- *Procesos Inter-Organizacionales (T2)* – La ejecución de los pasos de procesos de negocios por los miembros puede ser coordinada a través de la extensión de Orden (V1), mientras que la extensión de Composición (V3) puede unir a las organizaciones involucradas en pasos posteriores y el Seguimiento (V4) puede determinar que paso se está ejecutando actualmente. Además, la Autenticación (H5) puede garantizar el origen de los mensajes, y Localización (H6) puede ser usadas para ubicar a los proveedores de servicios.

- G-EEG pueden ayudar a enfrentar otros desafíos como se explica a continuación. Por ejemplo: (T3) *Monitoreo de Políticas* – La extensión de Puntualidad (V2) puede ser usada para garantizar el cumplimiento de propiedades relacionadas a los mensajes enviados y recibidos a través de un canal, como lo requiera una política de comunicación; (T4) *Integración de Aplicaciones* – las extensiones de Validación (H2) y Transformación (H3) pueden garantizar que los sistemas legados reciban y provean información en formatos apropiados, mientras que Auditoría (H1) permite que las bases de datos de mensajes estén disponibles a los sistemas legados cuando éstos no pueden comunicarse directamente con otras aplicaciones; (T5) *Interoperabilidad Sintáctica* – la extensión de Validación (H2) garantiza que los mensajes sean sintácticamente válidos; (T6) *Interoperabilidad Semántica* – la extensión de Transformación (H3) puede convertir el formato de mensajes usado por una organización a un formato usado por otra a través de una ontología común para garantizar consistencia semántica; (T7) *Subcontratación Flexible* – la subcontratación flexible de servicios públicos puede ser apoyada por Puntualidad (V2) y la extensión de Orden (V1) para controlar la calidad y el proceso de subcontratación, al igual que por Composición (V3) y Seguimiento (V4) para monitorear el proceso; (T8) *Ecosistema Dinámico* – la extensión de Localización (H6) puede ayudar a descubrir proveedores de servicios; y (T9) *Entrega por Múltiples Canales* – la extensión de Transformación (H3) ayuda a adaptar mensajes al formato requerido por cada canal, mientras que la extensión de la Composiciones (V3) conecta el portal de gobierno de acceso único con cada canal usado.

La Tabla 2 representa la relación resultante entre desafíos tecnológicos y las extensiones de mensajería.

Tabla 2: Desafíos Tecnológicos .vs. Extensiones de G-EEG

		H1	H2	H3	H4	H5	H6	H7	V1	V2	V3	V4
		Auditoría	Validación	Transformación	Criptografía	Autenticación	Localización	Alianza	Orden	Puntualidad	Composición	Seguimiento
T1	Acceso Unico	■	■		■							
T2	Procesos Inter-Organizacionales					■		■	■		■	■
T3	Monitoreo de Políticas									■		
T4	Integración de Aplicaciones	■	■	■								
T5	Interoperabilidad Sintáctica		■									
T6	Interoperabilidad Semántica			■								
T7	Subcontratación Flexible	■			■	■			■	■	■	■
T8	Ecosistema Dinámico						■					
T9	Entrega por Múltiples Canales			■							■	

1.6.3 Evaluación de G-EEG – Caso de Estudio

En esta sección, usamos el caso de estudio de servicios de licencias para ilustrar como G-EEG puede apoyar la entrega integrada de servicios públicos. En particular, G-EEG satisface los requerimientos de comunicación y colaboración del Servicio de Licencia para Negocios de Comida y Bebida (Sección 1.3) como se explica a continuación:

1) *Solicitud y Seguimiento* – El uso de un canal de comunicación por parte del portal de gobierno y la aplicación de front-office responsable por la emisión de las licencias en IACM, permite que todas las solicitudes y documentos necesarios enviados por los postulantes a través del portal pueden ser entregados a la aplicación de front-office de IACM. De manera similar, el uso de un canal de comunicación entre el portal de gobierno y la aplicación de back-office responsable por la emisión de las licencias en IACM, permite que la aplicación de back-office informe el estado de las solicitudes para que el portal pueda responder a los requerimientos de seguimiento.

2) *Entrega Integrada* – El uso de canales de comunicación entre IACM y cada una de las agencias involucradas en la producción del servicio – DSAL, DSSOPT, CB, IC y SS, permite que IACM pueda solicitar colaboración directamente a cada una de estas agencias, manteniendo al postulante al margen de estas interacciones.

3) *Integración* – Independientemente de las aplicaciones de software usadas por cada una de las agencias involucradas en la prestación del servicio, y aún en el caso que esas aplicaciones no existan, las agencias pueden usar los servicios de mensajería de G-EEG, ya que G-EEG permite la integración de sistemas legados (Sección 6.2.4) y también permite que la mensajería sea realizada por personas.

4) *Interoperabilidad* – Como se explicó en el punto anterior (Integración), G-EEG facilita el intercambio de información entre aplicaciones de software heterogéneas. Adicionalmente, G-EEG provee la extensión de Transformación que permite mediar entre dos vocabularios distintos usados por las agencias que colaboran en la provisión del servicio.

5) *Coordinación* – Como se explicó en el punto 2 (Entrega Integrada), G-EEG facilita la ejecución de un proceso de negocios entre varias agencias. Adicionalmente, la extensión V1 – Orden, permite monitorear la correcta ejecución del mismo.

6) *Flexibilidad* – De acuerdo a las especificaciones, G-EEG facilita la creación, configuración y modificación dinámica de las estructuras de comunicaciones (Sección 6.2.8). Adicionalmente, existen extensiones como V1 – Orden, V2 – Puntualidad, o V3 – Composición, que permiten fácilmente introducir cambios en los pasos de los procesos de negocios, o las políticas que regulan la ejecución del proceso, o los canales usados para las comunicaciones, sin que esto represente un trastorno importante en la prestación de los servicios de mensajería.

Una presentación detallada de la resolución del caso de estudio en base a una solución creada en base a las estructuras y servicios de G-EEG es el objetivo de la Sección 6.3.

1.7 Contribuciones de esta Tesis

Después de formular el problema principal tratado por esta tesis en la Sección 1.4, proponer a G-EEG como la solución a dicho problema en la Sección 1.5, y evaluar a G-EEG en función del problema original en la Sección 1.6, esta sección tiene por objeto describir cómo G-EEG se relaciona con otras soluciones y resultados en el área y en qué medida puede mejorarlas, considerando particularmente el dominio del Gobierno Integrado.

Las soluciones existentes se dividen en tres categorías - Organizacionales, Tecnológicas y Fundacionales – considerando las limitaciones de aplicarlas directamente al tratamiento de los desafíos de Gobierno Integrado, y se compara G-EEG a través de las tres categorías para identificar argumentos para su aplicación y mejoras potenciales:

- *Soluciones Organizacionales* – Tales soluciones comprenden guías y estándares usados por organizaciones para clasificar y organizar representaciones descriptivas de la información de la empresa como base para la interoperabilidad – como por ejemplo, información mantenida por las administraciones, servicios ofrecidos al cliente, etc. Para asegurar el cumplimiento de estándares en todo el gobierno, independientemente del software

concreto que se utiliza, los gobiernos aplican dos formas de soluciones organizacionales – Marcos de Interoperabilidad [IDA04] y Arquitecturas Empresariales [GAO06]. Ejemplos concretos incluyen: Marco de Interoperabilidad de Gobierno Electrónico (e-GIF) del Reino Unido [UK07], Marco de Interoperabilidad de Gobierno Electrónico de Nueva Zelanda (NZe-GIF) [NZ08a], Marco Europeo de Interoperabilidad de la Unión Europea [IDA04], la arquitectura de Zachman [Zac87], Arquitectura Empresarial Federal de Estados Unidos [FEA07], y Estándares y Arquitecturas para Aplicaciones de Gobierno Electrónico (SAGA) de Alemania [KBST06].

- *Soluciones Tecnológicas* – Comprenden productos de software específicos que permiten a las aplicaciones de software el intercambio de información y la ejecución de procesos de negocio. Comprenden: Soluciones de Propósito General, Soluciones de Dominio Específico, Arquitecturas de Software y Frameworks de Desarrollo. Las Soluciones de Propósito General incluyen software que puede ser aplicado en cualquier dominio, por ejemplo basados en Middleware Orientado a Mensajes (MOM) [SEI07], para proveer servicios de comunicación a aplicaciones de software. Algunos ejemplos son: BeaMessageQ de BEA Systems [BEA00], MSMQ de Microsoft [Dic98], y WebSphere de IBM [IBM08a]. Las Soluciones de Dominio Específico refieren a software confeccionado a medida del cliente para tratar necesidades particulares de gobierno, por ejemplo para intercambiar mensajes a través de plataformas de TI. Ejemplos incluyen: IAMS Messaging System de Irlanda [Col03] y Hermes Messaging Gateway de Hong Kong [CEC07]. Las Arquitecturas de Software son patrones de diseño de alto nivel que ayudan a la implementación y distribución de software dentro de las organizaciones, como Service-Oriented Architecture (SOA) [Erl05]. Los Frameworks de Desarrollo comprenden lenguajes, herramientas y métodos integrados para asistir el desarrollo de software, como ser Java Agent Development Framework (JADE) [TI08a].

- *Soluciones Fundacionales* – Tales soluciones incluyen modelos, lenguajes, teorías y frameworks que proveen semánticas formales y rigor metodológico para soportar soluciones organizacionales y tecnológicas. La Soluciones Fundacionales difieren en términos de: (1) estilo de especificación, por ejemplo especificaciones basadas en estado o en acciones; (2) tipo de semántica formal aplicada, como ser operacional, denotacional o axiomática; (3) la forma de relacionar los artefactos de desarrollo, tales como refinamiento o pre orden observacional; y (4) el soporte provisto por herramientas. Ejemplos son: RSL - RAISE Specification Language [RLG92], CSP - Communicating Sequential Processes [Ros05], Reo [Arb02] y Pi-Calculus [MPW89].

Aunque las soluciones mencionadas arriba son relevantes hasta cierto punto para el Gobierno Integrado, su aplicación directa a este dominio presenta ciertas limitaciones. Por ejemplo, las soluciones organizacionales proveen orientación a las agencias que implementan Gobierno Electrónico en términos de aplicación de estándares y mejores prácticas para alcanzar interoperabilidad. Sin embargo, para obtener impacto real, tales soluciones deben ser complementadas por software listo para usar capaz de ejecutar los procesos asociados. En el caso de las soluciones tecnológicas, la limitación más importante de las Soluciones de Propósito General y de Dominio Específico es que ofrecen funcionalidad fija, descansan sobre estructuras de comunicación definidas en forma estática, y carecen de soporte para manejar la evolución de dichas estructuras en el tiempo. Aunque son ampliamente adoptados para integración de aplicaciones, los enfoques basados en SOA sólo proveen arquitectura de referencia para el software a construir. Los servicios concretos como validación o transformación de mensajes, deben ser provistos e incluidos dentro de la arquitectura para solucionar desafíos tecnológicos particulares. Adicionalmente, SOA carece de soporte nativo para acuerdos de niveles de servicio y coordinación de proceso inter-agencias, si bien este tema es usualmente tratado combinando soluciones basadas en SOA con workflows. En cuanto a los Frameworks de Desarrollo, típicamente carecen de fundamentos precisos y de soporte para procesos organizacionales. Finalmente, las soluciones fundacionales presentan a los usuarios un desafío en términos de la habilidad requerida para su aplicación. La falta de conocimiento del dominio incorporado dentro de las mismas, incrementa también la distancia conceptual entre las descripciones formales y las aplicaciones reales.

En vista de las limitaciones de las soluciones existentes para soportar directamente el desarrollo de Gobierno Integrado, se enuncia que G-EEG puede soportar tal desarrollo a través de las dimensiones organizacionales, tecnológicas y fundacionales como se muestra a continuación:

- *G-EEG como Solución Organizacional* – G-EEG puede complementar los Marcos de Interoperabilidad mediante el soporte para interoperabilidad técnica, semántica y organizacional. Escrito en Java y utilizando XML para escribir mensajes, G-EEG es independiente de la plataforma y permite interoperabilidad técnica. Usando la extensión de transformación y un conjunto de extensiones semánticas para validar mensajes, mediar sintaxis y descubrir recursos, G-EEG soporta interoperabilidad semántica. G-EEG también permite la implementación de nuevas

extensiones para proveer funcionalidades requeridas por procesos de negocio, soportando así la interoperabilidad organizacional.

- *G-EEG como Solución Técnica* – G-EEG puede ser considerado como un MOM mejorado. Permite la comunicación entre aplicaciones de software mediante el intercambio de mensajes asincrónicos a través de canales creados y suscriptos en forma dinámica. También ofrece una serie de extensiones que proveen funcionalidad enriquecida, y un mecanismo para construir nuevas extensiones.

- *G-EEG como Solución Fundacional* – Un lenguaje formal y una teoría existen como fundamento de G-EEG y sus extensiones. Adicionalmente, el lenguaje formal puede ser usado para especificar nuevas extensiones.

A continuación, exponemos tres argumentos para considerar G-EEG como una mejora sobre las soluciones existentes: (1) Fundamento Formal – existe un modelo formal para definir G-EEG, mientras que las soluciones existentes carecen de fundamentos formales y por lo tanto de la capacidad para definir y verificar propiedades de comportamiento; (2) Mecanismos de Extensibilidad – mientras que las soluciones existentes ofrecen funcionalidad fija y carecen de extensibilidad incorporada, G-EEG permite por diseño la definición, implementación e integración de nuevas funcionalidades por medio de G-EEG-DEVELOP; y (3) soporte nativo para Gobierno Electrónico – mientras que la mayoría de las soluciones técnicas carecen de la perspectiva de aplicación de dominio específico, G-EEG fue diseñado particularmente para resolver los requerimientos planteados por la implementación de Gobierno Integrado.

1.8 Organización de la Tesis

El resto de esta tesis está organizada como se describe a continuación.

El Capítulo 2 explica sistemáticamente el dominio de Gobierno Integrado, focalizando la Agencia como el elemento principal de las estructuras de gobierno. La Agencia (Sección 2.1) produce Resultados utilizando Capacidades existentes o desarrollando nuevas, en base a Recursos, y enfrentando varios tipos de Desafíos. Resultados (Sección 2.2) captura los efectos visibles al público que produce el trabajo de una Agencia – las necesidades de los Clientes de gobierno, los Servicios ofrecidos para satisfacer dichas necesidades y los Canales usados para entregar tales Servicios. Capacidades (Sección 2.3) captura cómo las Agencias cumplen sus misiones, incluyendo pero no limitado a la prestación de Servicios: Colaboración a través de límites organizacionales, Asociación con el sector privado, Integración de varios recursos, y Coordinación de recursos para la entrega de resultados. Recursos (Sección 2.4) captura cómo una Agencia entrega Servicios: conducida por Procesos y soportada por Tecnología. Desafíos (Sección 2.5) captura algunas barreras que las Agencias deben sortear para la entrega de Servicios: desafíos Organizacionales para diferentes escenarios de integración, y desafíos Tecnológicos para ayudar a superar los desafíos Organizacionales.

El Capítulo 3 describe trabajos relacionados – soluciones organizacionales, tecnológicas y fundacionales que pueden soportar la implementación de Gobierno Integrado. Primero, se introduce un marco de evaluación para las diferentes categorías de soluciones (Sección 3.1). Segundo, se presentan seis soluciones organizacionales (Sección 3.2) – tres marcos de interoperabilidad y tres arquitecturas empresariales. Tercero, se explican siete soluciones tecnológicas (Sección 3.3) – tres productos middleware de propósito general, dos aplicaciones de software específicas para Gobierno, una arquitectura de software y un framework de desarrollo. Cuarto, cuatro lenguajes de especificación se introducen como soluciones organizacionales (Sección 3.4). Finalmente, las soluciones presentadas son evaluadas y comparadas (Sección 3.5).

El Capítulo 4 presenta los fundamentos de Mensajería Programable y de G-EEG. Se presenta el concepto de Mensajería Programable y los conceptos relacionados con su implementación G-EEG (Sección 4.1). Luego, se introduce la notación gráfica definida para representar los conceptos de G-EEG (Sección 4.2). En base a los conceptos, se explica la estructura de G-EEG, en término de sus tres componentes: G-EEG-CORE, G-EEG-EXTEND y G-EEG-DEVELOP (Sección 4.3). A continuación, se introduce el comportamiento de G-EEG (Sección 4.4), seguido de una descripción de un conjunto de extensiones propuestas por G-EEG en apoyo de la entrega integrada de servicios públicos (Sección 4.5). Finalmente, se explica el enfoque de formalización de G-EEG, desde las estructuras básicas de XML usadas para representar las estructuras de datos de mensajes y variables usados por G-EEG, a los servicios de mensajería ofrecidos por G-EEG, a través de una familia de lenguajes XML usados por G-EEG para definir la sintaxis y la semántica de los servicios de mensajería (Sección 4.6). Por último, se discuten los resultados obtenidos (Sección 4.7).

El Capítulo 5 presenta el desarrollo del prototipo G-EEG. El prototipo incluye los servicios de mensajería básicos y tres extensiones – Auditoría, Validación y Transformación. El proceso de desarrollo está documentado en cinco etapas: requerimientos (Sección 5.1), modelado (Sección 5.2), diseño (Sección 5.3), implementación (Sección 5.4) y entrega (Sección 5.5). En diferentes etapas, varios tipos de diagramas UML fueron producidos: diagrama de clases conceptual para capturar los conceptos del dominio; diagramas de casos de uso para modelar requerimientos; diagramas de clases de diseño para capturar la estructura de la arquitectura; diagramas de secuencia y colaboración para capturar el comportamiento de la arquitectura; y diagramas de componentes y entrega para describir la implementación.

El Capítulo 6 valida G-EEG. G-EEG es validado con respecto a: el problema tratado por la tesis presentado en la Sección 1.4 (Sección 6.1), los desafíos tecnológicos identificados en la Sección 2.5 (Sección 6.2), el caso de estudio que ilustra la entrega integrada de servicios de licencias presentado en la Sección 1.3 (Sección 6.3), y los trabajos relacionados presentados en las Secciones 3.2 a 3.4 (Sección 6.4).

El Capítulo 7 presenta conclusiones, incluyendo el resumen de la tesis (Sección 7.1), su contribución (Sección 7.2), y los planes para trabajos futuros de investigación y desarrollo (Sección 7.3).

La tesis incluye cuatro apéndices: Apéndice A contiene especificaciones en RSL; Apéndice B contiene requerimientos; Apéndice C incluye diagramas de desarrollo y Apéndice D describre artefactos de implementación.

Capítulo 2

Conceptos de Gobierno Integrado

Este capítulo muestra los resultados del análisis de dominio de Gobierno Integrado, enfocándose en el trabajo en red y la colaboración entre agencias, necesarios para la entrega de Servicios Integrados. El resultado principal de este análisis de dominio es un conjunto de conceptos (antecedentes) que ayuda a comprender el resto de la tesis.

Siguiendo con la Definición 2, un Gobierno Integrado es un conjunto de agencias que trabajan conjuntamente como una entidad única, generando respuestas integradas a las necesidades de la comunidad. Esta definición menciona explícitamente un concepto – Agencia, e insinúa cuatro más – Resultados, Capacidades, Recursos y Desafíos. Una "respuesta a las necesidades de la comunidad" es un Resultado de acciones llevadas a cabo por agencias. En la tesis, nos enfocamos en un tipo de Resultado – Servicio. Un Servicio es ofrecido por las agencias a sus clientes para satisfacer sus necesidades (Cliente) y es entregado a través de diferentes canales (Canal). Cuando una "respuesta" denota un Servicio, una "respuesta integrada" indica un Servicio Integrado – el resultado que se obtiene por el trabajo conjunto de las agencias. Con este fin, las agencias deberían ser capaces de (Capacidades): asociarse (Asociación), de colaborar a lo largo de los límites organizacionales (Colaboración), de integrar procesos y aplicaciones (Integración), y de coordinar la entrega de servicios (Coordinación). Para poder entregar estos Servicios Integrados, las agencias dependen de recursos humanos, financieros, organizacionales y técnicos (Recursos). En la tesis, nos enfocamos en recursos organizacionales (Proceso) y técnicos (Tecnología). Por último, las agencias que entregan Servicios Integrados deben superar desafíos legales, financieros, sociales, organizacionales y tecnológicos. En esta tesis, nos enfocamos en los desafíos organizacionales (Organizacionales) y tecnológicos (Tecnológicos).

Este conjunto de conceptos se presenta en la Figura 7. Cada concepto está elaborado en secciones posteriores: Agencia (Sección 2.1), Resultados (Sección 2.2) Capacidades (Sección 2.3) Recursos (Sección 2.4) y Desafíos (Sección 2.5).

2.1 Agencia

La capa administrativa de gobierno está dividida generalmente en unidades organizativas responsables por la ejecución de funciones específicas – ministerios, agencias, departamentos y departamentos de línea [Sch00]. Mientras que las tres primeras están organizadas alrededor de funciones, la última se enfoca en programas o políticas específicas de un sector. Sin embargo, mientras se desarrolla Gobierno Integrado, estas diferencias estructurales pierden relevancia, y la Agencia Gubernamental se mantiene como la principal unidad organizativa de interés. A continuación se presenta la definición de Agencia Gubernamental provista por Schacter [Sch00].

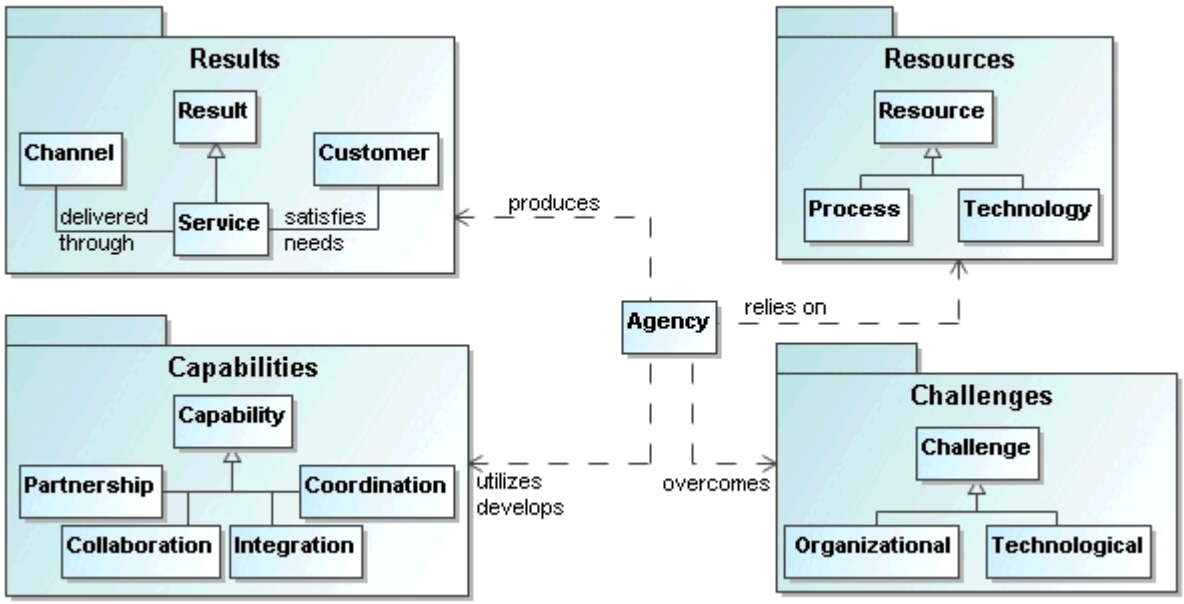
Definición 4: Agencia de Gobierno
Una Agencia Gubernamental es una unidad organizativa del sector público que se enfoca en funciones específicas y que tiene una serie de responsabilidades y autoridades.

Tradicionalmente, en gobierno, las agencias solamente se preocupan por cumplir sus misiones – funciones específicas asignadas a ellas dentro de su área de responsabilidad, y están poco incentivadas para colaborar con otras agencias. En contraste, Gobierno Integrado apunta a transformar a las agencias gubernamentales para permitirles trabajar como una entidad única, generando respuestas integradas a las necesidades de la comunidad (Definición 2).

La Figura 7 representa y relaciona los conceptos que definen el dominio de Gobierno Integrado, usando la Agencia como concepto central de este modelo. Según este cuadro, las Agencias producen Resultados – entregan Servicios a

Clientes para satisfacer sus necesidades a través de Canales, utilizando Recursos disponibles, usando o desarrollando nuevas Capacidades y superando varios tipos de desafíos. El enfoque de la tesis en el soporte tecnológico para el Gobierno Integrado determina la elección de Capacidades – Asociación, Colaboración, Integración y Coordinación; Recursos – Procesos y Tecnología; y Desafíos – Organizacional y Tecnológico.

Figura 7: Conceptos de Dominio – Gobierno Integrado



Los conceptos de Resultados, Capacidades, Recursos y Desafíos están explicados a continuación:

- *Resultados* – Componen los efectos beneficiosos o tangibles de las acciones gubernamentales [Mer08], llevadas a cabo principalmente por agencias de gobierno u organizaciones asociadas. La entrega de Resultados significa proveer facilidades (Servicios) exigidas por interesados en el gobierno (Clientes) a través de los medios más apropiados para ellos (Canales).

- *Capacidades* – Se refiere a habilidades de las agencias gubernamentales para llevar a cabo acciones específicas. En el contexto de Gobierno Integrado, las agencias deberían ser capaces de: colaborar más allá de los límites organizacionales (Colaboración); de llegar a acuerdos con organizaciones públicas, privadas o no gubernamentales (Asociación); utilizar diversos recursos efectivamente (Integración); y de coordinar el uso de Recursos y la entrega de Resultados con otras agencias (Coordinación). Mientras que algunas capacidades están presentes dentro de una Agencia, algunas otras deben ser desarrolladas.

- *Recursos* – Son la fuente de suministro o soporte [Mer08]. Para poder entregar los Resultados esperados, las agencias generalmente requieren recursos humanos, financieros, organizacionales y técnicos. En el contexto de Gobierno Integrado, nos enfocamos en los recursos organizacionales – la entrega de servicios a través de la ejecución de procesos eficaces (Procesos) y en los recursos técnicos – sistemas que permitan a las agencias trabajar conjuntamente, para automatizar los pasos de los procesos y para abrir nuevos canales de entrega de servicios entre las agencias y sus clientes (Tecnología).

- *Desafíos* – Consiste de las barreras que se deben superar para poder producir resultados. En el contexto de Gobierno Integrado y particularmente para colaboración, las agencias deben superar desafíos legales, financieros, sociales, organizacionales y tecnológicos. En la tesis, nos enfocamos en desafíos organizacionales y tecnológicos: transformar a las agencias de organizaciones jerárquicas a organizaciones en red (Organizacional) y en proveer soluciones tecnológicas para permitir trabajos colaborativo entre las agencias (Tecnológico).

Las siguientes secciones explican detalladamente los conceptos de Resultados, Capacidades, Recursos y Desafíos.

2.2 Resultados

Esta sección explica tres conceptos de dominio relacionados a Resultados: (1) Cliente – el receptor final de los resultados producidos por las agencias; (2) Servicio – resultado entregado por las agencias a sus clientes, y (3) Canal – el medio usado para entregar los Servicios a los Clientes. Estos conceptos están elaborados en las siguientes secciones.

2.2.1 Cliente

El concepto del cliente se entiende perfectamente en las actividades diarias del sector privado. En el sector público, el concepto comenzó a utilizarse con la adopción por parte de los gobiernos de iniciativas de la Nueva Gestión Pública (NGP). Mientras que en ambos contextos un cliente se refiere a cualquier entidad que reciba un producto o un servicio por parte de un proveedor, el contexto influencia de gran manera el comportamiento del cliente y la relación entre clientes y proveedores. Por ejemplo, mientras que en el sector privado los clientes tienen libertad para elegir productos o servicios de varios proveedores, en el sector público el gobierno es el único proveedor de bienes y servicios públicos. A continuación se presenta la definición de Cliente de Gobierno.

Definición 5: Cliente de Gobierno
El Cliente de Gobierno es una persona, un grupo de personas o una organización que hace uso o recibe los productos o servicios provistos por Agencias de Gobierno o por organizaciones intermediarias en representación de Agencias de Gobierno.

Ejemplos específicos de Clientes de Gobierno son: ciudadanos, empresas, visitantes, funcionarios, etc. Ejemplos de servicios que reciben son: certificados de nacimiento (ciudadanos), licencias para empresas (empresas) información turística (visitantes) y cursos de capacitación (funcionarios).

El Gobierno entrega productos y servicios a sus clientes para satisfacer sus necesidades y para cumplir con sus expectativas. Sin embargo, tales necesidades y expectativas varían entre diferentes grupos de personas, dependiendo de los valores culturales, nivel de educación, experiencias vividas y otros factores. Por ejemplo, mientras que los desempleados exigen información acerca de subsidios, oportunidades laborales y programas de capacitación para mejorar sus habilidades, las personas de la tercera edad esperan información acerca de sus jubilaciones y el acceso a servicios de obras sociales. Al mismo tiempo, estos últimos pueden preferir recibir servicios a través de una ventanilla de la agencia, mientras que los profesionales prefieren el uso de canales electrónicos. Además, las necesidades y expectativas de diferentes clientes pueden cambiar con el paso del tiempo. Por ejemplo, los desempleados, luego de recibir servicios de subsidio, pueden pedir servicios de capacitación, tales como inscribirse en un curso de conducción, antes de solicitar un empleo como chofer.

Aprendiendo de experiencias sobre cómo mejorar las necesidades de los clientes, los gobiernos están adoptando una orientación de enfoque a sus clientes como parte de estrategias e iniciativas de Gobierno Electrónico. A continuación podemos encontrar la definición de enfoque al cliente, específica para el dominio de gobierno [FML03].

Definición 6: Enfoque al Cliente
El enfoque al cliente se trata de proveer a clientes y empresas con una interfaz coherente del gobierno, que refleje las necesidades del cliente en vez de la estructura del gobierno.

Introducir el enfoque al cliente en el sector público significa poner a los clientes en el medio de las actividades gubernamentales, considerando sus necesidades y diseñando servicios y operaciones basadas en estas necesidades, y no en lo que el gobierno pueda producir. Por ejemplo, la Oficina del Gabinete del Reino Unido [Cab08b] se refiere a las organizaciones transformadas, luego de adoptar iniciativas de Enfoque al Cliente, como organizaciones de servicio público que tratan a sus clientes con dignidad y respeto, aseguran que los servicios provistos representen valor por dinero, optimizan las opciones para usuarios, y toman en cuenta las expectativas del cliente cuando crean servicios. Esta interpretación trata tres razones principales para introducir el enfoque al cliente en gobierno: (1) mejorar las

interacciones entre el gobierno y los clientes, (2) tomar en cuentas las expectativas del cliente mientras se crean servicios y (3) producir valor para los clientes mientras se entregan los servicios.

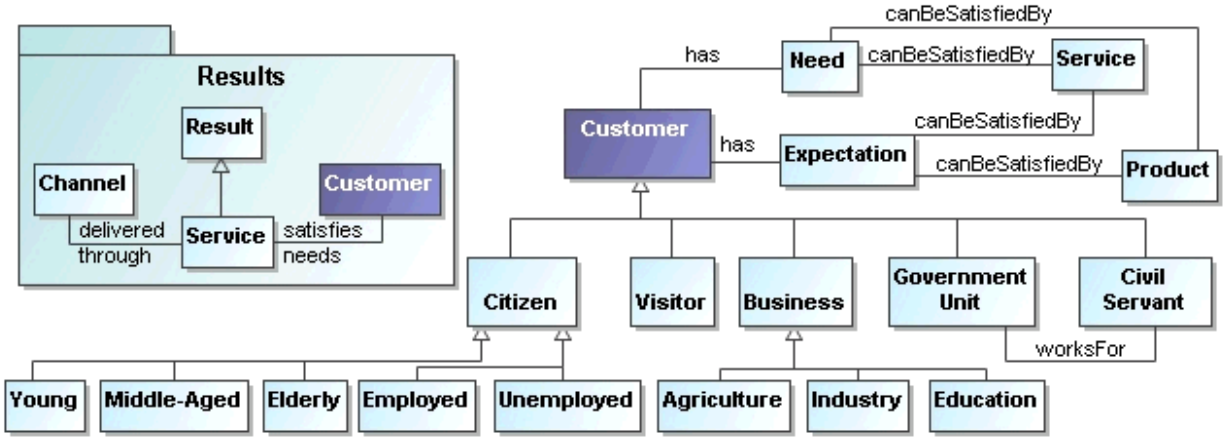
En la adopción de enfoque al cliente, una herramienta común para poder entender mejor las necesidades del cliente es la Segmentación de Clientes. La segmentación de clientes trata acerca de dividir una población usuaria en grupos homogéneos que tienen atributos en común o que comparten ciertos intereses. El enfoque permite diseñar servicios a medida acorde a las necesidades y expectativas de cada segmento. Segmentos típicos son: ciudadanos, empresas, visitantes, empleados y el gobierno mismo. Dentro de cada segmento, segmentos más refinados pueden ser definidos, basándose en datos de los usuarios disponibles por las administraciones y que pueden identificar características más específicas. Otras herramientas y técnicas usadas para adquirir conocimiento acerca de clientes de gobierno son: censos, encuestas, cuestionarios, grupos de enfoque, entrevistas personales, observaciones, etc. Las bases para la segmentación de cliente, las características de los segmentos considerados y posible información usada se pueden ver en la Tabla 3.

Tabla 3: Bases, Características e Información para la Segmentación de Clientes

Bases	Características	Información
¿Quiénes son?	Sociodemográficas	Edad, sexo, lugar de vivienda (urbano o rural), región
¿A qué se dedican?	Económicas	Ingresos, sector, número de empleados, volumen de trabajo
¿Cómo piensan?	Psicográficas	Estilo de vida, valores, reacción a nuevas tendencias
¿Cómo sienten?	Físicas/Psicológicas	Incapacidades, actitud, lealtad

A principios de la década del ’90, el enfoque al cliente y la segmentación de clientes fueron dos iniciativas introducidas por los gobiernos como parte de sus agenda de la NGP, ya que uno de los principios de la NGP es transformar a las agencias en organizaciones conducidas por los clientes, donde principalmente se satisfacen las necesidades de los clientes y no los de la burocracia. Si bien se logran claros beneficios por introducir orientación al cliente en gobierno, también existen algunas preocupaciones [DD02]. Por ejemplo, la naturaleza emprendedora de la NGP es criticada basándose en el argumento de que “los funcionarios no entregan atención al cliente, sino que entregan democracia”. Mientras que la ciudadanía no es considerada como un factor crítico por la NGP, los ciudadanos deben ser diferenciados de los clientes ya que “los ciudadanos son portadores de derechos y obligaciones dentro del contexto de una comunidad”; por el contrario, los clientes no comparten intereses, sino que están más interesados en sus propios beneficios. Asimismo, preocupa la transformación de ciudadanos en clientes ya que esto puede dañar la democracia y la administración pública debido a la desaparición del interés público. Sin embargo, hoy en día la NGP es una práctica no discutida y adoptada por líderes en Gobierno Electrónico. Además, algunas iniciativas de la NGP, como el enfoque al cliente, son adoptadas por el gobierno como reacción a las presiones de los ciudadanos para que se produzcan innovaciones en el gobierno. Sin embargo, los gobiernos siempre deberían trabajar para lograr un balance entre cumplir tanto con las necesidades de los clientes (individual) como con las del público (colectivo).

Figura 8: Conceptos de Dominio Relacionados con Cliente



La Figura 8 muestra los principales conceptos del dominio relacionados al concepto Cliente. El paquete nombrado Resultados ubica al concepto Cliente en el dominio general. Como se explicó anteriormente, los Clientes tienen Necesidades y Expectativas que pueden ser satisfechas por Servicios públicos y por Productos provistos por el gobierno. Generalmente, los Clientes están segmentados en: Ciudadanos, Visitantes, Empresas, Unidades de Gobierno, Funcionarios, etc. Segmentos más refinados pueden ser definidos basándose en información detallada del cliente, como por ejemplo segmentar a los Ciudadanos en base a edad: Jóvenes, de Mediana Edad y Adultos; o por su estado laboral: Empleado o Desempleado; o segmentar Empresas en base a un sector económico: Agricultura, Industria o Educación, entro otros.

El Ejemplo 2 muestra una iniciativa de Enfoque al Cliente por parte del Gobierno del Reino Unido [Cab08a].

Ejemplo 2: Una Iniciativa de Enfoque al Cliente
El Gobierno del Reino Unido apunta a entregar servicios públicos eficientes, efectivos, excelentes, equitativos y que den poder para todos, con el cliente siempre y en todas partes en el corazón de la producción de servicios públicos. Con este fin, y para impulsar el cambio de enfoque al cliente dentro las agencias se desarrolló la herramienta de Excelencia en Atención al Cliente. Esta herramienta define el Standard de Excelencia en Atención al Cliente que ofrece guías a las agencias en áreas identificadas como prioridad para los clientes, como la entrega, puntualidad, información, profesionalismo y buena actitud por parte de los empleados. Los estándares también tratan el desarrollo del conocimiento del cliente, el entendimiento de la experiencia del usuario, y la medición de la satisfacción del Servicio. Se puede encontrar más información en http://www.cse.cabinetoffice.gov.uk/homeCSE.do?

2.2.2 Servicios

Los gobiernos entregan servicios públicos a los clientes para cumplir con sus necesidades y expectativas. La definición de Servicio Público, según el diccionario American Heritage [Pic00], se presenta a continuación.

Definición 7: Servicio Público
Un Servicio Público es un servicio provisto para el beneficio del público, especialmente producido por organizaciones sin fines de lucro.

Como se explicó en la Sección 2.2.1, se proveen diferentes tipos de servicios públicos a diferentes segmentos de clientes. Por ejemplo: (1) un servicio de gobierno a cliente (G2C) que emite certificados de nacimiento, (2) un servicio de gobierno a empresas (G2B) que emite licencias de negocios para abrir y operar un restaurante, (3) un servicio de gobierno a gobierno (G2G) que provee opiniones técnicas acerca de planos de construcción para la agencia que emite las licencias de negocios, por parte de la agencia de obras públicas, (4) un servicio de gobierno a empleados (G2E) que provee capacitación para funcionarios, y (5) un servicio de gobierno a visitante (G2V) que emite visas para entrar a un país. La Tabla 4 muestra doce servicios públicos ofrecidos a ciudadanos y ocho servicios públicos ofrecidos a empresas. Todos estos servicios públicos son considerados comunes por los miembros de la Unión Europea [WD04].

Muchos otros servicios públicos son entregados por el gobierno a sus clientes además de los ya mencionados en la Tabla 4. Existen varias tipologías para clasificar a estos servicios. Una de ellas es provista por la Arquitectura de Gobierno Empresarial [PT04], que clasifica a los servicios públicos en cuatro categorías:

1) *Certificación* – Servicios que permiten a las administraciones públicas manifestar y certificar ciertos estados del mundo real. Por ejemplo, la emisión de documentos personales como certificados de nacimiento, matrimonio y defunción, pasaportes, documentos de identidad, certificados de antecedentes penales, etc.

2) *Control* – Servicios con los cuales las administraciones públicas aseguran la conformidad de las reglas, generalmente a través de la inspección del comportamiento de los clientes. Por ejemplo, controlando las

condiciones sanitarias de las empresas, controlando el estado de los mecanismos de prevención de incendios en lugares públicos, monitoreando la construcción de viviendas sociales, etc.

3) *Autorización* – Servicios con los cuales las administraciones públicas conceden permisos y aprobaciones a postulantes. Por ejemplo, la emisión de licencias para operar un establecimiento de alimentos o bebidas, la emisión de permisos para importar y vender carnes de ave, registración de marcas, etc.

4) *Producción* – Servicios con los cuales las administraciones públicas administran infraestructura y entregan servicios públicos. Por ejemplo, la emisión de licencias para proveedores de servicio de Internet o la facturación a operadores de telecomunicaciones.

Tabla 4: Servicios Públicos Comunes para Ciudadanos y Empresas adoptados por la Unión Europea.

Servicios G2C	Servicios G2B
1) Impuesto sobre la renta 2) Búsqueda de trabajo 3) Beneficios de seguridad social 4) Documentos personales 5) Registro automotor 6) Permiso de construcción 7) Declaración a a la policía 8) Bibliotecas públicas 9) Certificado de nacimiento/matrimonio 10) Inscripción a enseñanza superior 11) Cambio de domicilio 12) Servicios relacionados con la salud	1) Contribuciones sociales de los empleados 2) Impuesto de sociedades 3) Impuesto al valor agregado 4) Inscripción de una nueva compañía 5) Presentación de datos estadísticos 6) Declaración de aduana 7) Permisos relacionados con el medio ambiente 8) Contratación pública

Para poder entregar servicios públicos, la interacción entre el gobierno y los clientes se lleva a cabo, cada vez más, a través de canales electrónicos y es soportada por las TIC. Nos referimos a servicios que son entregados con la ayuda de las TIC como Servicios Públicos Electrónicos (SEP). La siguiente definición ha sido adoptada [HE05].

Definición 8: Servicio Público Electrónico (SPE)

Un Servicio Público Electrónico es un Servicio Público que depende de las TIC para soportar la interacción entre los proveedores y los receptores del servicio.

Una definición alternativa [Sak02] de SPE es el uso de procesos de entrega electrónica para la información, programas, estrategias y servicios de gobierno. Ejemplos de SPE son: información pública disponible en portales de gobierno, notificaciones acerca de condiciones meteorológicas recibidas a través de SMS, y declaraciones fiscales y pago de impuestos hechos a través de Internet.

Además de apoyar las interacciones entre el gobierno y los clientes, las TIC pueden apoyar el proceso de negocios que soporta la producción de un servicio. Basado en el grado de apoyo de las TIC, la entrega de un SPE puede ofrecer diferentes grados de automatización. Por ejemplo, descargar un formulario de solicitud de un sitio web (interacción de una sola vía), descargar y enviar el formulario de solicitud (interacción de doble vía), o completar toda la transacción, incluyendo el pago en línea.

El nivel de madurez de la entrega de SPE puede ser evaluado en base al grado de automatización del proceso de negocios que lo produce. Existen varios modelos que pueden medir la madurez de la entrega de SPE. A continuación, describimos el modelo publicado por United Nations Public Administration Network (UNPAN) [Ron02]. El modelo, representado en la Figura 9, propone los siguientes cinco niveles de madurez:

1) *Emergente* – Una presencia formal en la web es establecida a través de sitios web de gobierno aislados, publicando información estática relacionada al gobierno, como números de teléfono y otros medios de contacto de oficiales de gobierno. Esta fase permite a los clientes acceder a información sin necesidad de visitar agencias.

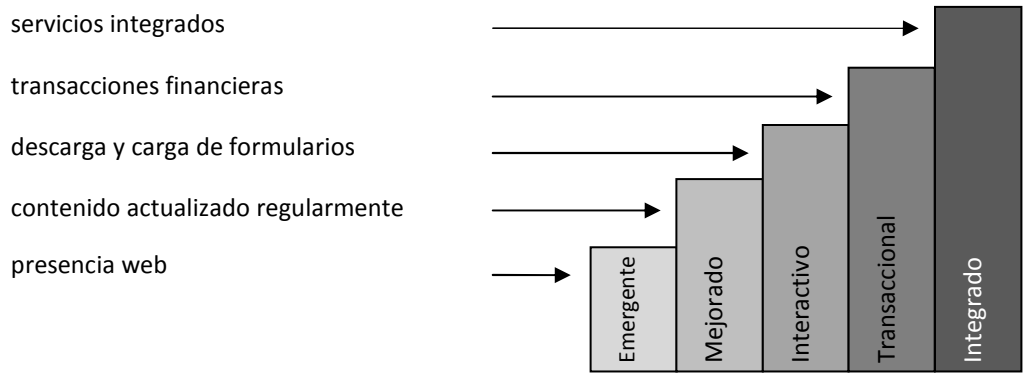
2) *Mejorado* – Aumenta la cantidad de sitios web oficiales, los mismos son regularmente actualizados y contienen información dinámica, como publicaciones, legislaciones y boletines informativos. Se proveen enlaces entre sitios webs, especialmente entre sitios del gobierno nacional y de los gobiernos locales. También se proveen servicios de búsquedas.

3) *Interactivo* – Provee acceso en línea a una gran variedad de agencias de gobierno y los SPE están disponibles a clientes de gobierno a través de sus sitios web. También se proveen servicios más avanzados, como por ejemplo, correos electrónicos a funcionarios, descarga y presentación de formularios de solicitud, y búsquedas en bases de datos especializadas.

4) *Transaccional* – Se pueden ejecutar transacciones completas y seguras a través de un sitio web, como, por ejemplo, solicitar pasaportes, obtener visas, registrar nacimientos y fallecimientos, solicitar permisos y licencias, y pagar impuestos, cuotas y facturas. A este punto, un cliente puede completar toda la transacción electrónicamente. Sin embargo, esto también requiere la implementación de soluciones técnicas más avanzadas, como, por ejemplo, sitios seguros y firmas digitales.

5) *Integrado* – Un único punto de acceso, p. ej. un portal de gobierno, provee todos los SPE para clientes. Con los límites organizacionales removidos en el ciberespacio, los servicios son ofrecidos basándose en necesidades comunes, generalmente eventos de vida para los ciudadanos y episodios de negocios para empresas. Esta etapa requiere la integración de los procesos de back-office.

Figura 9: Modelo de Madurez de Gobierno Electrónico



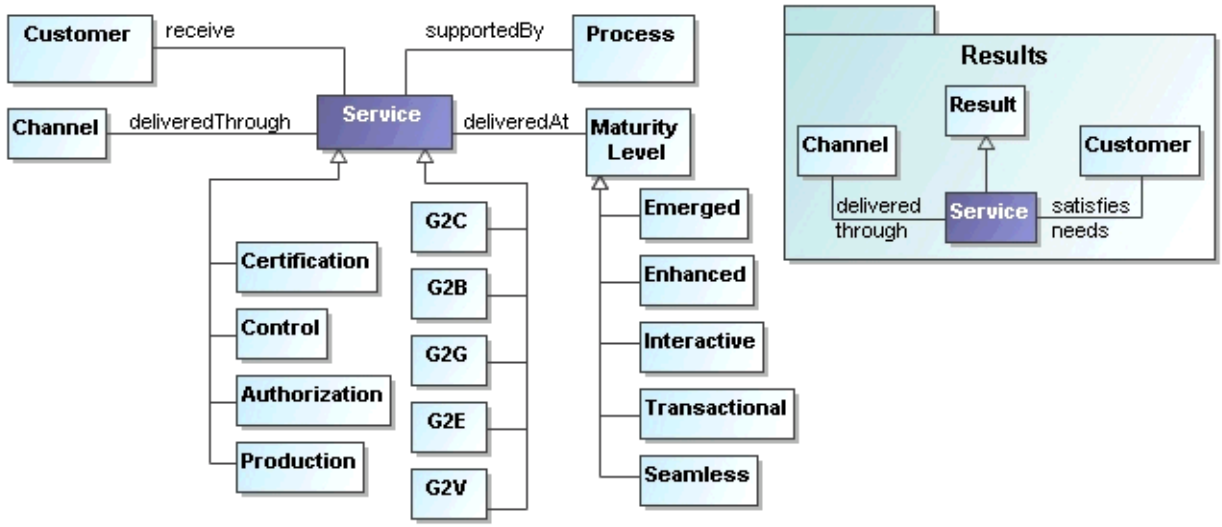
Otros modelos bien conocidos que pueden evaluar el nivel de madurez de SPE son provistos por Gartner [BD00], OECD [FML03] y el gobierno australiano [TN99]. El modelo Gartner identifica etapas de Publicación, Interacción, Transacción e Integración. El modelo OECD define etapas de Información, Interacción, Transacción y Transformación. El modelo del gobierno australiano comprende etapas de Información, Interacción, Transacción y Datos Compartidos. Los tres modelos combinan las dos primeras fases del modelo UNPAN (Emergente/Mejorado) en una fase única llamada Publicación o Información.

La etapa de madurez más alta de los cuatro modelos presentados, si bien difieren en sus nombres – Integrado, Integración, Datos Compartidos, y Transformación, coinciden con el concepto – la provisión de servicios integrados. En esta etapa, las agencias de gobierno comparten información y cooperan entre ellas para entregar SPE. Se llega a este nivel, cuando se logra una integración total de funciones y servicios a través de límites organizacionales. Coincide con el concepto de Servicios Integrados (Definición 3) – un servicio público completo entregado en colaboración por distintas agencias de gobierno a través de un contacto único. Desde el punto de vista del cliente, un servicio integrado es entregado por una sola organización virtual.

Cuatro características principales caracterizan a los servicios integrados: (1) diseñados basados en las necesidades de los clientes, (2) entregados usando múltiples canales, (3) accedidos a través de un portal único, y (4) entregados de manera proactiva y no reactiva – ofrecidos por el gobierno en vez de solicitados por los clientes. Por ejemplo, la entrega integrada de licencias de negocios (Sección 1.3) comprende las siguientes características: (a) los postulantes son capaces de encontrar toda la información relacionada con el servicio en el portal de gobierno; (b) los postulantes son capaces de enviar formularios de solicitud y documentos necesarios a través del portal; (c) los postulantes son notificados en caso que los documentos enviados estén incompletos; (d) todos los servicios gobierno a gobierno (G2G) relacionados, como por ejemplo los provistos por DSSOBT, CB, IC y SS a IACM, son automáticamente requeridos, procesados y coordinados; (e) los postulantes reciben notificaciones acerca de inspecciones en el edificio a través de los canales seleccionados – celulares, fax, correo electrónico, SMS, etc.; (f) los postulantes desconocen las agencias involucradas en la entrega de servicios; y (g) los postulantes son capaces de seguir el estado de sus solicitudes en cualquier momento durante todo el proceso a través del portal de gobierno.

La Figura 10 muestra los principales conceptos del dominio relacionados al concepto Servicio. El paquete nombrado Resultados ubica al concepto en el dominio general. Un Servicio es recibido por un Cliente, enviado a través de un Canal y producido por un Proceso de negocios. Los Servicios pueden ser clasificados según diferentes criterios. Por ejemplo: en base al tipo de tarea llevada a cabo por el gobierno como proveedor de servicios – Certificación, Control, Autorización y Producción; a los receptores de servicios –G2C, G2B, G2G, G2E y G2V; y otros. Basándose en los niveles de apoyo de las TIC, los Servicios son entregados en diferentes Niveles de Madurez: Emergente, Mejorado, Interactivo, Transaccional e Integrado, según el modelo de UNPAN.

Figura 10: Conceptos de Dominio Relacionados con Servicio



El Ejemplo 3 muestra un servicio integrado a empresas provisto por el gobierno sueco, publicado por e-Government Good Practice Framework [Mat05]. El servicio recibió el premio eEurope en el año 2006.

Ejemplo 3: Servicio Integrado

El Servicio AX es una solución única para solicitar reembolsos de exportación y para declarar exportaciones. La mejor práctica en servicios integrados en el año 2006, el servicio AX se entrega conjuntamente por la Junta de Agricultura Sueca y el Servicio Aduanero Sueco. Ambas agencias comparten la responsabilidad de manejar los reembolsos de exportación. Los principales accionistas del Servicio AX son compañías suecas que se encargan del comercio internacional de comestibles que reúnen los requisitos necesarios para las tasas de reembolsos de exportación. AX provee un punto de entrada único (AX-Board) que integra los procesos de back-office de ambas agencias.

Los resultados del proyecto incluyen: (1) cinco formularios que se debían llenar previamente, fueron reemplazados por una única declaración; (2) la cantidad de visitas oficiales requeridas fueron reducidas de 3 a 1; (3) los tiempos del servicio aduanero dedicados a inspecciones documentarias y físicas de los productos por los cuales se solicitaban las

tasas de reembolso de exportación fueron reducidos en un 50%; y (4) el tiempo que dedicaba la Junta de Agricultura Sueca para llevar a cabo el proceso de reembolso fue reducido en un 40%. Este servicio se encuentra disponible en: http://www2.tullverket.se/tid_demo/asp/tvSMS_Login1.asp

2.2.3 Canales

Las Agencias entregan Servicios a Clientes a través de Canales. Para poder aumentar la accesibilidad, estos canales deberían estar disponibles desde cualquier lugar – hogares, escuelas, agencias, etc., así como también cuando los receptores están en movimiento; deberían soportar diferentes formas de entrega – sitios web, ventanillas, teléfonos fijos, celulares; y deberían estar a toda hora disponibles. Por ejemplo, proveer acceso a servicios a través de un sitio web permite un auto-servicio por parte de los clientes, mientras que entregar esos servicios a través de celulares facilita la localización de servicios. A continuación definimos canal de entrega.

Definición 9: Canal de Entrega
Un Canal de Entrega es un medio usado por un proveedor de bienes o servicios para interactuar con, y para entregar tales bienes y servicios a sus clientes.

Ejemplos de canales de entrega son: ventanillas, teléfonos fijos, faxes, call centres, sitios web, correo electrónico, celulares, mensajería instantánea (IM), servicio de mensajes de texto (SMS), telecentros, respuesta de voz interactiva (IVR), televisión interactiva (iTv), etc. Entre tales canales, algunos están soportados por las nuevas tecnologías, mientras que otros no. Considerando esto, los canales de entrega se clasifican en: canales tradicionales - como las ventanillas, teléfonos fijos o faxes; y en canales electrónicos - como los sitios web, correo electrónico, celulares, mensajería instantánea, SMS, telecentros, IVR, iTv, etc.

Entre la variedad de canales, no todos son apropiados o efectivos para entregar servicios públicos a ciertos segmentos de clientes. Por ejemplo, mientras que los SMS pueden ser efectivos para entregar servicios a los jóvenes, no se aconseja para entregar servicios a las personas de tercera edad. Para poder determinar los tipos de canales apropiados para entregar servicios a ciertos segmentos de clientes, sus características deberían ser evaluadas. Ejemplos de características de canales son [IDA04][Sha00]:

1) *Conexión* – Si la interacción entre el cliente y el proveedor es sincrónica (interacción directa) o asincrónica (interacción indirecta). El grado de conexión puede depender de la manera en que esté implementado el canal. Por ejemplo: un canal indirecto como el correo electrónico puede ser considerado como directo si se implementa una respuesta automática.
2) *Velocidad* – El tiempo requerido para entregar el servicio a través de un canal.
3) *Accesibilidad* – El número de clientes capaces de acceder a un canal.
4) *Inclusión* – Cuán efectivo es el canal para sortear brechas sociales, y para facilitar la participación y la inclusión.
5) *Seguridad y Privacidad* – El nivel de seguridad usado para proteger información confidencial enviada a través de un canal.

Decisiones acerca de la elección de los canales apropiados para entregar un determinado servicio público pueden hacerse en base a que las organizaciones adopten iniciativas de enfoque al cliente, como parte de una estrategia de entrega de servicios más amplia. La definición de Estrategia de Entrega de Servicios provista por el Gobierno del Reino Unido se presenta a continuación [Jon04].

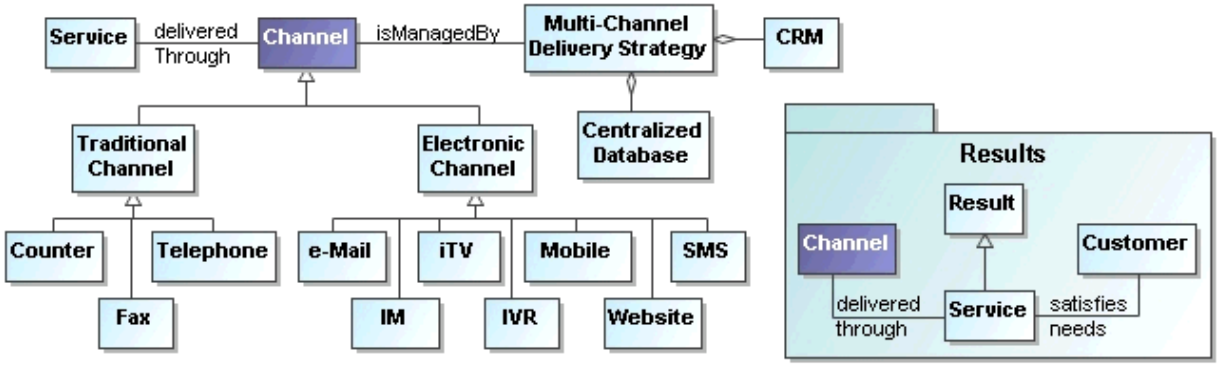
Definición 10: Estrategia de Entrega de Servicios
La Estrategia de Entrega de Servicios representa un grupo de decisiones de negocios de una organización acerca de cómo sus servicios y productos deberían ser entregados a sus clientes.

Particularmente, una Estrategia de Entrega a través de Múltiples Canales es una Estrategia de Entrega de Servicios que utiliza varios canales, que soporta las interacciones a través de los distintos canales basándose en una base de datos

central, y que administra los datos de clientes a través de herramientas Customer Relationship Management (CRM). Las características de una Estrategia de Entrega a través de Múltiples Canales [IDA04] incluyen: aplicaciones de front-office integradas, que acceden a información almacenada en un repositorio central; información de clientes y los servicios solicitados por ellos siempre disponible e idéntica para todos los canales; y si bien el proceso de entrega puede involucrar múltiples sesiones, cada sesión se puede llevar a cabo a través de cualquier canal que disponga de acceso al repositorio central.

Conceptos de dominio relacionados con el concepto Canal se presentan en la Figura 11. El paquete nombrado Resultados ubica al concepto en el dominio general. Los Servicios son entregados a través de Canales, administrados por una Estrategia de Entrega a través de Múltiples Canales. Dicha estrategia se implementa a través de una Base de Datos Centralizada y una herramienta CRM. Los canales pueden ser tradicionales - como las ventanillas, faxes o teléfonos fijos; o electrónicos - como el correo electrónico, IM, iTv, IVR, celulares, sitios web o SMS.

Figura 11: Conceptos de Dominio Relacionados con Canal



El Ejemplo 4 presenta una experiencia de Canadá, donde un portal inalámbrico es integrado a otros canales [Can07].

Ejemplo 4: Entrega de Servicios a través de Múltiples Canales

El portal inalámbrico canadiense es un proyecto que invita a los ciudadanos a probar servicios disponibles y a que brinden sus observaciones. El portal está designado para complementar canales ya existentes; ya que la información accesible a través del portal se encuentra disponible a través de: Portal del Gobierno Canadiense, el número telefónico 1-800 O-Canada, y la red de Centros de Acceso de Servicios de Canadá.

Para acceder a informaciones y servicios del gobierno a través del portal, los dispositivos móviles, como los celulares con acceso a Internet o los dispositivos digitales personales (PDAs), deben acceder a wap.gc.ca o a www.gc.ca. De esta forma, logran acceso a servicios como:

- *Tiempo de espera en la frontera* – Tiempo estimado para cruzar la frontera terrestre entre Canadá y Estados Unidos en lugares determinados.
- *Compañías Canadienses* – Acceso a una base de datos con información de proveedores canadienses y canales de distribución, para determinar la competencia, para formar sociedades, y para descubrir empresas de exportaciones.
- *Convertidor de Moneda* – Conversión de y a dólares canadienses.
- *Tasas de Cambio* – Acceso a las tasas de cambio actuales.
- *Indicadores Económicos* – Acceso a las últimas estadísticas de la población de Canadá, las tasas de desempleo e inflación, y el Producto Bruto Interno (PBI).
- *Números de Teléfono de los Empleados del Gobierno* – Acceso a un directorio integrado de funcionarios del gobierno federal.

2.3 Capacidades

Para poder entregar los Resultados esperados (Sección 2.2), una Agencia (Sección 2.1) debe utilizar y desarrollar varias Capacidades. Esta sección se dedica a definir tales Capacidades.

Las siguiente Capacidades son consideradas necesarias, particularmente para que las agencias puedan implementar y participar en Gobierno Integrado: (1) Colaboración – capacidad para trabajar con otras agencias; (2) Asociación – capacidad para asociarse con los sectores privados y no gubernamentales para poder adquirir fondos y adoptar soluciones de TIC innovadoras; (3) Integración – capacidad para hacer más efectivo el uso de recursos disponibles; (4) Coordinación – capacidad para coordinar iniciativas entre agencias. Las siguientes secciones explican detalladamente estas capacidades.

2.3.1 Colaboración

El desarrollo de servicios integrados requiere la colaboración entre agencias de gobierno. Por ejemplo, considerar el servicio de emisión de licencias de negocios para comidas y bebidas (Ejemplo 1). Para poder entregar el servicio en una forma integrada, IACM colabora con DSAL, DSSOPT, IC, CB y SS. De esta manera, no se le requiere al postulante que visite cada una de las agencias para solicitar un servicio, sino que se le ofrece una repuesta integrada (un-solo-gobierno) y una experiencia interactiva.

La colaboración entre agencias de gobierno, particularmente para la entrega de servicios integrados, provee beneficios claros al gobierno y a sus clientes por igual. Desde el punto de vista del gobierno, como debe llevar a cabo todos los procesos para entregar varios servicios a clientes individuales, mantiene una mirada integral de cómo los clientes están siendo servidos. Desde el punto de vista del cliente, el gobierno se muestra como una única organización para la entrega de servicios públicos, sin importar la agencia o nivel de gobierno que está involucrado en la producción de los servicios.

La entrega de servicios públicos es solo un ejemplo de la colaboración entre agencias de gobierno. Otros ejemplos son: (1) Portales Únicos – ofrecer servicios a través de portales únicos requiere la colaboración entre todas las agencias involucradas para agrupar e integrar los servicios ofrecidos, según los eventos de vida para los ciudadanos o episodios de negocios para las empresas; (2) Mejoras a la Eficiencia – eliminando procesos o bases de datos duplicados que son mantenidos por varias agencias, a través de la designación de una agencia como responsable para mantener y proveer acceso a datos y procesos integrados; (3) Software de Infraestructura Común – haciendo disponible un software de infraestructura de gobierno que provea funcionalidad común para autenticación, seguimiento de solicitudes, notificación, etc., a agencias individuales para que desarrollen y ejecuten sus SPE, reduciendo la necesidad de las agencias de desarrollar sus propias soluciones, y apoyando la colaboración entre agencias para que desarrollen servicios integrados.

Ejemplos previos demostraron colaboraciones entre agencias al nivel de producción de un servicio. Sin embargo, la colaboración también es requerida en diferentes etapas del desarrollo de sistemas y en diferentes aspectos de la entrega de servicios. Por ejemplo, OECD [FML03] identifica los siguientes tipos de colaboración para la entrega de servicios integrados, los tres primeros se enfocan en diferentes etapas de desarrollo, y los últimos dos se enfocan en temas específicos del servicio:

1) *Fase de Planificación* – Diseño de políticas, estándares de entrega, implementación de métodos y cronogramas, y coordinación de la compra de servicios y equipos;

2) *Fase de Modelado* – Coordinación de políticas y procedimientos que aseguren que el desarrollo de servicios integrados cubre las necesidades y expectativas de ciertos grupos de clientes;

3) *Fase de Implementación* – Adaptación de sistemas de back-office a una interfaz integrada para clientes;

4) *Cuestiones de Clientes* – Acuerdos en la calidad de servicio, presentación de materiales, asignación de responsabilidades para resolver problemas, quejas y apelaciones, toma de decisiones en conjunto para casos concretos; y

5) *Cuestiones de Entrega* – Consenso en una política de servicio de entrega consistente para todos los canales.

Otra perspectiva sobre las colaboraciones es la naturaleza de las asociaciones, ya que diferentes patrones de colaboración y requerimientos pueden ser identificados basados en ella. Según esto, las siguientes colaboraciones pueden ser identificas:

1) *Colaboración Horizontal* – Diferentes agencias en un mismo nivel de gobierno ejecutan procesos de negocios en forma conjunta. Esta colaboración requiere la comunicación de información y la coordinación de procesos entre agencias.

2) *Colaboración Vertical* – Agencias en diferentes niveles de gobierno colaboran en la provisión de servicios públicos integrados. En general, esto incluye la actualización y mantenimiento de un repositorio de información central para cubrir amplios intereses sociales, incluyendo la identificación de ciudadanos, declaraciones fiscales, asuntos de seguridad y salud, etc. La colaboración vertical requiere la comunicación de información y la coordinación de procesos entre agencias.

3) *Colaboración Intersectorial* – Un paso de un proceso de negocios o el proceso de negocios completo, usado para la producción de un servicio público, puede ser ejecutado por una entidad ajena al gobierno – sector privado o no gubernamental. Además de los requerimientos para las colaboraciones horizontales y verticales, la colaboración intersectorial requiere el cumplimiento de políticas con respecto a la calidad de servicios y la continuidad de la entrega.

4) *Colaboración Transnacional* – Varias administraciones públicas de diferentes países colaboran en la entrega de servicios integrados que cruzan los límites nacionales. Tal colaboración requiere interoperabilidad semántica para superar diferencias legales, de procedimiento, o lingüísticas entre países.

Un asunto clave para considerar cuando se establece un ambiente colaborativo es que la colaboración no surge naturalmente, sino que tiene que ser establecesida cuidadosamente. Basándose en experiencias prácticas, se propone un conjunto de recomendaciones para implementar colaboraciones exitosas para Gobierno Electrónico [FGW06]:

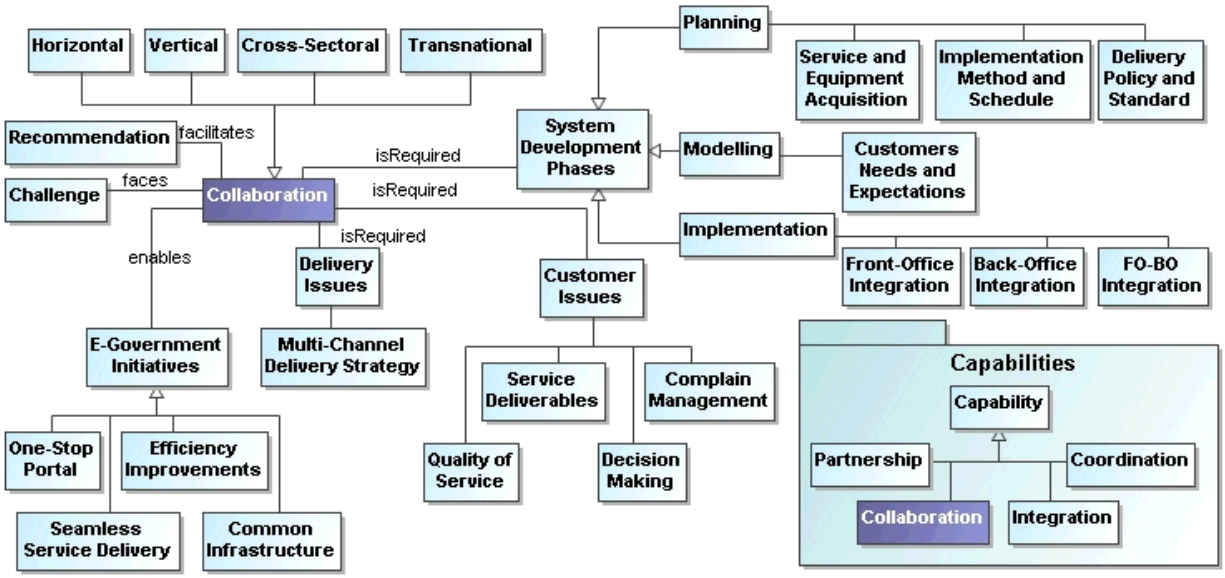
- Crear oportunidades para colaborar a fin de tratar situaciones de crisis y otros eventos precipitantes;
- Construir un entendimiento compartido de objetivos y propósitos entre grupos de colaboradores;
- Cultivar un equipo de campeones entre los grupos de colaboradores;
- Evaluar la preparación y facilitar la participación en colaboraciones;
- Resaltar las oportunidades para combinar información de múltiples fuentes, dentro de los límites legales;
- Desarrollar un modelo de negocio para una viabilidad a largo plazo;
- Detallar el entendimiento de cómo la información debe ser intercambiada y usada;
- Considerar tecnologías de vanguardia, pero aceptando la realidad de todo lo legado;
- Aceptar muchas opiniones informadas acerca de que herramientas de software usar y elegirlas cuidadosamente;
- Adherir a estándares y, en lo posible, ayudar a establecerlos.

Finalmente, aunque la colaboración es el principal facilitador para desarrollar Gobierno Integrado, establecer un ambiente colaborativo en un contexto de gobierno es todo un desafío. Los desafíos cruzan dimensiones legales, financieras, sociales, organizativas y tecnológicas. Los últimos dos tipos de desafíos se explican en la Sección 2.5.

La Figura 12 representa los principales conceptos de dominio relacionados con el concepto Colaboración entre agencias de gobierno. El paquete nombrado Capacidades ubica al concepto en el dominio general. Se identifican cuatro tipos de Colaboración: Horizontal, Vertical, Intersectorial y Transnacional. Varias Iniciativas de Gobierno Electrónico requieren Colaboración ente agencias, por ejemplo: Portales Únicos, Entrega de Servicios Integrados, Mejoras a la Eficiencia, e Infraestructura Común. La Colaboración se requiere cuando se entregan servicios y también durante todos las Fases de Desarrollo de Sistemas, incluyendo: (1) Planificación – decidiendo acerca de Adquisición de Servicios y Equipos, Implementación de Métodos y Horarios, y la Políticas de Entrega y Estándares; (2) Modelado: Identificación de Necesidades y Expectativas del Cliente; y (3) Implementación – llevando a cabo la integración de Front-Office, Back-Office y Front-Back-Office. La Colaboración también se necesita para decidir sobre Asuntos de Entrega de Servicios, por ejemplo la Estrategia de Entrega a través de Múltiples Canales, y sobre los Asuntos de Clientes, como por ejemplo, Calidad de Servicios, Servicios que se Entregan, Administración de Quejas y Toma de

Decisiones en casos individuales. Además, la Colaboración en gobierno se enfrenta con algunos Desafíos conocidos, pero existen Recomendaciones de mejores prácticas para superarlos.

Figura 12: Conceptos de Dominio Relacionados con Colaboración



El Ejemplo 5 ilustra un caso de colaboraciones horizontales, verticales e intersectoriales en gobierno.

Ejemplo 5: Colaboración Horizontal, Vertical e Intersectorial en Bélgica.

El sistema de seguridad social belga [CK04] provee un repositorio rico de ejemplos de colaboración en la entrega de Gobierno Integrado. La conexión de aproximadamente 2000 instituciones de seguro social en el gobierno y el sector privado es una exitosa combinación de integración de Back-Office y soluciones de Portales Únicos. Con muchos niveles de gobierno involucrados en la provisión de beneficios de seguros sociales, el desarrollo de políticas y la financiación son responsabilidades del gobierno federal, mientras que la implementación la realizan los gobiernos regionales y locales.

El sistema integra oficinas de cinco niveles de gobierno:

1) Federal;
2) Comunidad - flamencos, franceses, alemanes;
3) Regional - Flandes, Valona, Bruselas;
4) Provincial - excepto Bruselas, las regiones están subdivididas en 5 provincias cada una; y
5) Municipal - 589 municipalidades, todas auto-administradas.

Atravesando tales niveles, un workflow integrado ha sido desarrollado para asegurar que todas las instituciones de seguridad social y las compañías del sector privado estén conectadas las unas a las otras y a su vez al portal de seguridad social [Bel08]. Dedicados a los ciudadanos, las empresas y las instituciones públicas, el portal contiene servicios integrados, más de 4.000 hojas de información y 34 transacciones operativas. Se provee una red para facilitar el intercambio de información entre el público (de todos los niveles) y las instituciones privadas, todas capaces de consultar mutuamente sus bases de datos e intercambiar hasta 185 tipos de mensajes electrónicos diferentes. En el año 2005, más de 500 millones de mensajes fueron intercambiados a través del sistema.

El próximo ejemplo presenta el caso de colaboración transnacional e intersectorial con el objetivo de crear un sistema de Compras Electrónicas para obras sociale en la Unión Europea [EC05]. El proyecto ha sido desarrollado por socios de Bélgica, Alemania, Grecia, Italia, España y el Reino Unido, incluyendo proveedores de obras sociales del sector público y privado. Un problema concreto que se tuvo que resolver fue la falta de estándares en los sistemas de catalogación, a

niveles nacionales o de la UE, que fuera aceptado y usado por todos los sistemas de obras sociales nacionales y varios miembros de la UE [EIS07].

Ejemplo 6: Colaboración Intersectorial e Inter-Sistemas para Compras Electrónicas en la Unión Europea

El sistema optimizado de e-Procurement (EPOS) es un proyecto que apunta a establecer un servicio de compras electrónico pan-europeo para el sector de obras sociales. El sistema soporta la búsqueda de productos de salud y comercialización de los vastos proveedores de salud. Los usuarios son hospitales y proveedores de equipos y artículos de consumo, incluyendo proveedores farmacéuticos de toda Europa.

EPOS facilita a hospitales públicos el monitoreo de procesos de compra, reduciendo ciclos de obtención, resaltando la transparencia, reduciendo costos para recursos compartidos, soportando la toma de decisiones a través de una base de conocimiento compartida, y logrando economía de escalas. Desde el punto de vista de los proveedores de salud, EPOS les facilita la administración de sus contractos, exhibición de sus productos a través de catálogos electrónicos y participación en negociaciones y acuerdos de precios.

Los principales components del sistema son:

1) Subsistema de Gestión de Catálogo
2) Subsistema de Planificación de Demanda
3) Subsistema de Planificación de Licitaciones
4) Subsistema de Subastas
5) Subsistema de Workflow
6) Sistema de Gestión de Documentos
7) Subsistema de Seguridad

El sistema EPOS implementa un repositorio central de información relacionado a la salud, organizado en categorías. La solución adoptada para la implementación del catálogo se basa en estándares GMDN (General Medical Device Nomenclature) y estándares CPV (Common Procurement Vocabulary).

2.3.2 Asociación

Una amplia gama de aplicaciones técnicas debe ser construida para poder desarrollar un Gobierno Electrónico, desde la simple provisión de redes de computadoras y hardware, a través del desarrollo de aplicaciones de software complejas para la entrega de servicios públicos, hasta la entrega de infraestructura de TIC a gran escala para facilitar el acceso a Internet a todos los ciudadanos en todo el país. Todos estos esfuerzos requieren habilidades técnicas. Sin embargo, las TIC no es la competencia central de los gobiernos. Los gobiernos no tienen ni la experiencia, ni la habilidad o los recursos para participar en el desarrollo innovador de productos de TIC, la construcción de aplicaciones de software complicadas o la entrega de infraestructura de TIC.

Para enfrentar estos desafíos, los gobiernos están buscando nuevas maneras de financiar, implementar y entregar proyectos de TIC asociándose con el sector privado. La racionalidad depende de las fortalezas complementarias de cada socio para lograr los resultados esperados – con socios de gobierno trabajando para cumplir con las necesidades de los ciudadanos y los socios privados sabiendo como innovar a través del uso de las TIC; juntos pueden entregar más valor a los ciudadanos.

Sociedades Públicas-Privadas (SPP) es la asociación entre organizaciones de los sectores públicos y privados. SPP se basa generalmente en un acuerdo entre el gobierno y una empresa privada para compartir riesgos y recompensas de un proyecto que involucra la producción y la entrega de bienes y servicios públicos. Por ejemplo, el gobierno de Australia [Aus06] define SPP como un método de compra que involucra el uso de capital del sector privado para financiar activos que son usados por una agencia de gobierno para entregar resultados. En la tesis, adoptamos otra definición provista por el Consejo Canadiense para las Sociedades Públicas-Privadas [Can08a], como se presenta a continuación.

Definición 11: Sociedades Públicas-Privadas (SPP)
SPP es un proyecto corporativo entre los sectores públicos y privados, construido con la experiencia de cada socio, que satisface necesidades claramente definidas a través de una asignación apropiada de recursos, riesgos y recompensas.

SPP puede producir beneficios para los socios involucrados de ambos sectores, público y privado. Beneficios a socios del sector público incluyen [FML03]: (1) acceder a habilidades especializadas que pueden ser difíciles o muy costosas de mantener en el gobierno; (2) compartir servicios y recursos con socios del sector privado para obtener beneficios por la economía de escala; (3) obtener financiación del sector privado para desarrollar y operar nuevos servicios públicos; (4) involucrar a socios del sector privado para que descubran oportunidades de innovación y mejoras en la eficiencia; y (5) compartir riesgos de proyecto con socios del sector privado. En general, asociarse con el sector privado permite al gobierno dedicarle más atención y recursos a políticas centrales y a asuntos de gobierno. Los beneficios para los socios del sector privado incluyen [FML03][Par03]: (1) entregar servicios públicos a través de infraestructura privada y de ahí crear oportunidades de negocio; (2) aprender acerca del dominio de aplicación del gobierno y las necesidades del sector público; (3) evaluar oportunidades de negocios seguras y de inversiones a largo plazo; y (4) generar nuevos negocios con la certeza y la seguridad garantizadas por un contrato de gobierno.

Un conjunto de mejores prácticas en SPP han sido desarrolladas por los gobiernos del Reino Unido, Estados Unidos y Australia [Can08b]. Por ejemplo, el gobierno del Reino Unido adoptó una estrategia general de SPP [Tre08a] cubriendo una amplia gama de estructuras de negocios y acuerdos de sociedades a través de Iniciativa Financiera Privada (IFP) [Tre08b]. La IFP es una parte de la estrategia de gobierno para entregar servicios públicos en colaboración con el sector privado bajo acuerdos mutuos, como proyectos en conjunto, concesiones y subcontratación. Otros países como Italia, Japón, Holanda e Irlanda están considerando adoptar el modelo, y todos consultan al gobierno del Reino Unido acerca del IFP [Tre08a].

Se pueden usar diferentes modelos para analizar marcos de SPP para Gobierno Electrónico. A continuación, describimos dos modelos complementarios provistos por los gobiernos de Australia y Canadá. El Modelo Financiero de SPP definido por el gobierno de Australia se centra en la definición de inversiones y beneficios para ambos socios. El Modelo Operacional de SPP definido por el gobierno de Canadá se centra en la definición de responsabilidades asumidas por el sector privado.

El Modelo Financiero de SPP comprende las siguientes cinco categorías. Cada una está ilustrada por un ejemplo:

1) *Ingresos por Publicidad y Patrocinio* – En este arreglo, el gobierno puede recibir honorarios a cambio de: (1) anuncios directos de una compañía en un sitio web del gobierno; (2) marketing indirecto, como, por ejemplo, analizando los hábitos de gastos de los usuarios en base a información obtenida de sitios web oficiales; y (3) acuerdos de patrocinio.

Ejemplo 7: Modelo Financiero SPP – Ingresos por Publicidad y Patrocinio
El Servicio Postal de los Estados Unidos (USPS) [USP08] provee un sitio web para registrar el cambio de domicilio de un cliente. El sitio web fue creado a través de una SPP con una compañía privada que recibe ingresos a través de publicidad directa en el sitio web y por patrocinar un sitio web vinculado que provee una guía a las personas que se mudan, ofreciéndoles servicios adicionales.

2) *Financiación en base a Aranceles* – Este arreglo depende de aranceles cobrados por acceder a servicios informacionales y transaccionales, y a información especializada basada en suscripciones. Los ingresos obtenidos son compartidos entre los socios.

Ejemplo 8: Modelo Financiero SPP – Financiación en base a Aranceles
El gobierno estatal del estado de Arizona estableció una SPP [AG02] con IBM para integrar todos los SPE dentro de un acceso único en Internet [Ari02]. IBM proveyó la infraestructura para el portal empresarial y además

desarrolló la aplicación por sobre tal infraestructura. Además, asignó una inversión inicial de USD 1.5 millones para hardware y software. La financiación en curso para el portal proviene de:

- Aranceles de Suscripción – Aranceles de suscripción anual para servicios mínimos destinados a empresas y profesionales, que oscilan desde USD 50 a USD 100.
- Aranceles por Comodidad – Aranceles complementarios para información comercialmente valiosa, a un costo de USD 1.00 a USD 7.00 por uso.
- Aranceles por Transacción – Aranceles cobrados por servicios y actividades, como licencias, títulos, matrículas y renovaciones, oscilando entre USD 1.25 a USD 5.00 por uso.

3) *Ahorro de Gastos Compartido* – Los ahorros de gastos generados luego de implementar la solución se comparten entre los socios de los sectores públicos y privados, los mismos son usados para financiar la inversión inicial. El socio del sector privado cubre parte del gasto de desarrollo de una solución y, a cambio, recibe una parte de los ahorros generados por la nueva solución.

Ejemplo 9**:** Modelo Financiero SPP – Ahorro de Gastos Compartido

El Ministerio de Educación estadounidense lleva a cabo un proyecto de modernización de la Oficina de Asistencia Financiera a Estudiantes a través de una SPP con Accenture, AFSA Data Corp, y KPMG Consulting. El acuerdo integra el sistema directo de préstamos e-Servicing en programas de ayuda financiera para estudiantes. El nuevo sistema e-Servicing posibilita a prestatarios ver sus cuentas de Crédito Directo en línea, hacer pagos por Internet o a través de un proveedor de servicios, recibir notificaciones por correo electrónico, y llevar a cabo transacciones en línea o por teléfono. Mientras que el consorcio privado paga por el desarrollo e implementación del sistema, sus ingresos se calcularán en cómo satisfacen las medidas de desempeño definidas en el contracto de cincos años, en particular si puede generar el nivel de ahorros esperado o no. El costo del proyecto fue de USD 6.5 millones. Con el uso del nuevo servicio se espera que se ahorren USD 79.1 millones en un período de cinco años.

4) *Ingresos Compartidos* – Los ingresos generados por un mejor servicio desarrollado y operado a través de este tipo de SPP son usados para financiar las inversiones hechas por el sector privado.

Ejemplo 10: Modelo Financiero SPP – Ingresos Compartidos

El Departamento Impositivo del estado de Virginia (VATAX) lanzó el Proyecto de Asociación Virginia en sociedad con la compañía CGI, buscando reemplazar el núcleo del sistema de contabilidad y rediseñar los procesos centrales de negocios del Departamento. La solución implementada proveyó una serie de servicios relacionados a los impuestos, por ejemplo, un servicio que permite archivar la declaración de impuestos; un servicio que permite el registro de nuevas empresas; un servicio que provee acceso a una base de datos de políticas; y un servicio para el intercambio de correspondencia, el establecimiento de planes de pagos, y la generación y el levantamiento de embargos preventivos. Todos los servicios implementan una "vista única" del contribuyente para el Departamento. La asociación es operada bajo el modelo de ingresos compartidos. Luego de financiar el contrato, CGI recibe compensaciones a medida que se obtiene ingresos adicionales. En marzo de 2004, ocho meses antes de tiempo, se generaron USD 198 millones en ingresos, cantidad suficiente para pagar la totalidad del contrato.

5) *Entrega Completa del Servicio* – Bajo este tipo de SPP, el sector privado es vinculado como contratista para que asuma ciertas responsabilidades del gobierno.

Ejemplo 11: Modelo Financiero SPP – Entrega Completa del Servicio

En 2001, una sociedad conjunta, Liverpool Direct Limited (LDL) [LDL08] fue formada entre el Consejo Municipal de la ciudad de Liverpool (19.9%) y British Telecom (80.1%). LDL provee una amplia gama de servicios de consultoría en TIC y entrega servicios públicos de punta a punta totalmente integrados a los residentes de la

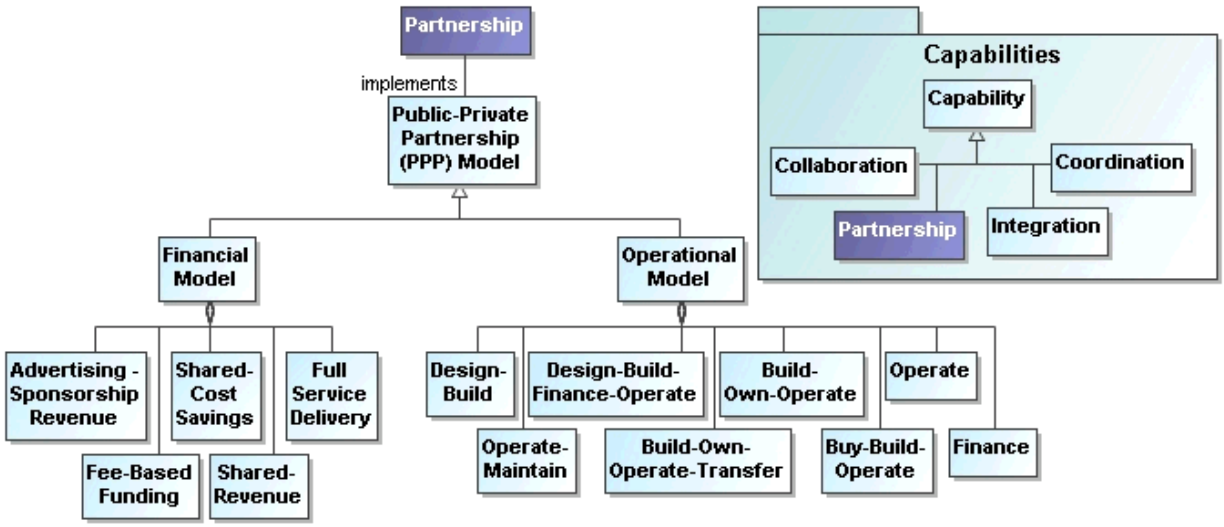
ciudad de Liverpool. En el arreglo de la SPP, LDL es responsable por los servicios de TIC, recursos humanos, ingresos, beneficios y servicios de contacto al cliente. Como tal, ha sido una fuerza motriz en la transformación de los servicios municipales dentro de la ciudad de Liverpool. La SPP le ha dado a la ciudad de Liverpool los beneficios de implementar tecnología de vanguardia con la experiencia del sector privado, mientras el sector público retiene el control de la provisión de servicios.

Complementando al Modelo Financiero de SPP, el Modelo Operativo de SPP [Can08a] se enfoca en las responsabilidades asignadas al sector privado. Este modelo define las siguientes opciones para la formulación de SPP:

1) *Diseño-Construcción* - El sector privado diseña y construye servicios de infraestructura, generalmente dentro de un precio fijo, y se transfiere el riesgo por exceso en los gastos al socio del sector privado.
2) *Operación-Mantenimiento* - El sector privado opera un activo público por un tiempo determinado, cobrando una cuota por su uso y mantenimiento.
3) *Diseño-Construcción-Financiación-Operación* - El sector privado diseña, financia y construye un servicio bajo un contrato a largo plazo. Luego de que se completa la construcción, opera el servicio durante la duración del contracto. Al finalizar el contrato, el servicio es transferido al sector público.
4) *Construcción-Posesión-Operación* - El sector privado construye, posee y opera un complejo o servicio en perpetuidad. Las restricciones públicas están especificadas en el contrato y periódicamente supervisadas por un ente regulador.
5) *Construcción-Posesión-Operación-Transferencia* - Un operador del sector privado recibe una franquicia para financiar, diseñar, construir y operar un servicio por un período de tiempo determinado. Durante la operación, tiene derecho a cobrar aranceles a los usuarios. Luego de finalizado el período, la posesión del servicio es transferida al sector público.
6) *Compra-Construcción-Operación* - Un activo público es transferido a una entidad privada o cuasi-pública para ser operada, mantenida y mejorada por un período de tiempo determinado.
7) *Operación* - Un operador privado recibe una licencia o el derecho de operar un servicio público durante un período determinado.
8) *Financiación* - Un socio del sector privado financia un proyecto o usa un mecanismo como, por ejemplo, un leasing a largo plazo para financiarlo.

Como se describió, existe una amplia gama de opciones para que los gobiernos puedan fomentar la participación del sector privado en actividades del sector público, incluyendo iniciativas de Gobierno Electrónico. Sin embargo, no hay ningún tipo de modelo que cumpla con todos los requisitos para encuadrar en todas las circunstancias. Luego de evaluar los diferentes modelos, cada gobierno debe definir su propio modelo, en base a las condiciones locales.

Figura 13: Concepto de Dominio Relacionado con Asociación



La Figura 13 representa los principales conceptos de dominio relacionados con el concepto de Asociación. El paquete nombrado Capacidades ubica al concepto en el dominio general. Una Asociación se implementa bajo un modelo de SPP. Existen dos modelos de SPP - Financiero y Operativo. El Modelo Financiero de SPP ofrece cinco opciones: Publicidad y Patrocinio, Financiación en base a Aranceles, Ahorro de Costos Compartido, Ingresos Compartidos y Entrega Completa de Servicios. El Modelo Operativo de SPP ofrece ocho opciones: Diseño-Construcción, Operación-Mantenimiento, Diseño-Construcción-Financiación-Operación, Construcción-Posesión-Operación, Construcción-Posesión-Operación-Transferencia, Compra-Construcción-Operación, Operación y Financiación.

2.3.3 Integración

Como se explicó en la Sección 2.3.1, las agencias de gobierno deben colaborar para poder entregar servicios integrados, y tales colaboraciones requieren la integración de información y procesos. Sin embargo, integrar información generada por diferentes agencias, generalmente registrada en diferentes formatos, e integrar sus procesos de negocios, generalmente soportados por diferentes plataformas tecnológicas, es difícil y complejo. El requerimiento clave es la interoperabilidad - la habilidad del software y de los procesos de negocios para intercambiar información y para compartir información y conocimientos [BCDF06]. A continuación se presenta la definición de interoperabilidad [MZ95].

Definición 12: Interoperabilidad
Interoperabilidad es la habilidad para intercambiar funcionalidad y datos interpretables entre dos entidades.

Usualmente, se definen tres niveles de interoperabilidad [IDA04]:

1) *Técnica* – Cubre aspectos técnicos para conectar sistemas de computadoras y servicios, incluyendo cuestiones de interfaces abiertas, servicios de interconexión, presentación e integración de datos, middleware, y servicios de seguridad. Interoperabilidad Técnica se ocupa de la habilidad para conectar sistemas de computación.
2) *Semántica* – Asegura que el significado exacto de información intercambiada es compartido y entendido por las aplicaciones de software. Se ocupa de que las aplicaciones de software puedan interpretar correctamente la información intercambiada.
3) *Organizativa* – Define y modela procesos y objetivos de negocio. Interoperabilidad Organizativa se ocupa de la habilidad de las organizaciones para colaborar, a pesar de las diferentes estructuras, sistemas y procesos internos.

La Interoperabilidad es facilitada a través de dos enfoques. El primero se basa en Marco de Interoperabilidad - un conjunto de requerimientos para sistemas de TIC y procesos de negocios para que puedan comunicar e intercambiar información. El segundo enfoque se basa en Arquitectura Empresarial - un plano conceptual que define como una organización puede lograr mejorar sus objetivos actuales y futuros. Muchos países considerados líderes en Gobierno Electrónico han desarrollado sus propios Marcos de Interoperabilidad o Arquitecturas Empresariales para implementar estrategias de Gobierno Electrónico. Ambos enfoques son explicados e ilustrados a continuación. La definición de Marco de Interoperabilidad [IDA04] se presenta debajo.

Definición 13**:** Marco de Interoperabilidad
El Marco de Interoperabilidad es un conjunto de estándares y pautas que describen como diferentes organizaciones han aceptado, o les han pedido que acepten, interactuar entre ellas.

Uno de los mejores ejemplos de Marcos de Interoperabilidad proviene del Reino Unido, quienes desarrollaron el Marco de Interoperabilidad de Gobierno Electrónico eGIF [UK07]. Las características principales del marco están representadas a continuación y sus componentes pueden verse en la Figura 14.

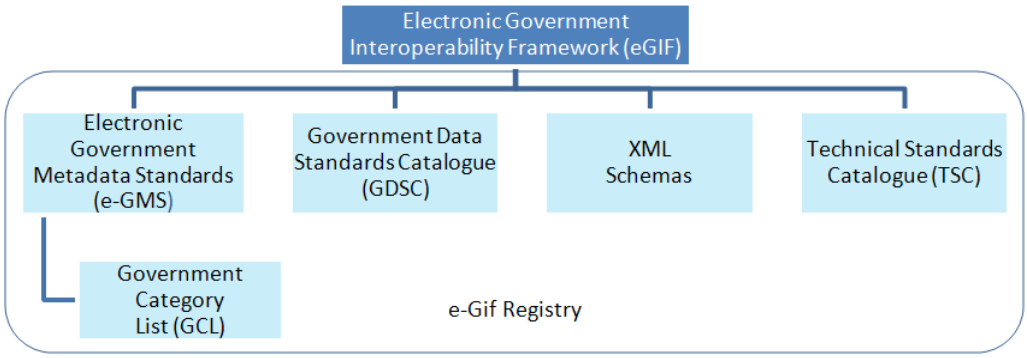
Ejemplo 12: Marco de Interoperabilidad de Gobierno Electrónico del Reino Unido – eGIF

eGIF define un conjunto de políticas y especificaciones técnicas que regulan los flujos de información a través del gobierno y del sector público. eGIF cubre cuatro aspectos: (1) interconectividad, (2) integración de información, (3) meta-datos para administración de contenidos y (4) acceso a servicios electrónicos. Las principales decisiones políticas que sustentan a eGIF incluyen:

- Adopción universal de especificaciones comunes usadas en Internet y en la World Wide Web;
- Adopción de XML como estándard para administración e integración de datos;
- Adopción de un navegador web como la interface principal;
- Definición de metadatos para recursos de información de gobierno;
- Adopción del Estándares de Metadatos de Gobierno Electrónico basado en el modelo Dublin Core [DCMI08]; y
- Desarrollo de la Lista de Categorías de Gobierno - una lista de categorías para clasificar recursos de gobierno.

Los componentes del eGIF son los Estándares de Metadatos de Gobierno Electrónico (e-GMS), el Catálogo de Estándares de Datos de Gobierno (GDSC), Esquemas XML, y el Catálogo de Estándares Técnicos (TSC).

Figura 14: Marco de Interoperabilidad de Gobierno Electrónico – eGIF



La mayoría de los marcos de interoperabilidad de Gobierno Electrónico desarrollados y mantenidos por países individuales coinciden en los siguientes estándares comunes [IDA04]:

1) *Transporte* – estándares para redes – Local Area Networks (LAN) y Wide Area Networks (WAN);
2) *Presentación de Información* – estándares para archivos, hipertextos, transferencia de mensajes y conjuntos de caracteres;
3) *Estándares de Internet* – nombres de dominio, navegadores web y presentadores;
4) *Integración de Información* – estándares basados en tecnologías XML;
5) *Modelado de Información* – UML y RDF son los estándares más usados para este propósito;
6) *Transformación de Información* - se usa mayormente eXtensible Stylesheet Transformation (XSLT) [W3C99a]; y
7) *Metadatos* – generalmente se adopta el modelo Dublin Core, con posibilidad de extensiones nacionales.

El Segundo enfoque para facilitar interoperabilidad es Arquitectura Empresarial. A continuación, se presenta la definición [GAO06].

Definición 14: Arquitectura Empresarial

Arquitectura Empresarial es una guía para cambios organizacionales definidos usando modelos, palabras, gráficos u otras representaciones, que describe en términos de negocios y tecnológicos cómo una organización funciona en la actualidad, cómo pretende funcionar en el futuro, y qué planes tienen para transformar su estado actual al estado futuro.

Generalmente, una Arquitectura Empresarial está organizada en un grupo de modelos, cada uno proveyendo diferentes perspectivas o vistas de la arquitectura. Cada uno de estos modelos especifica un grupo de elementos cohesivos – procesos, estructura de información, actividades, métodos, productos u otros artefactos, que son de interés a un grupo particular de interesados.

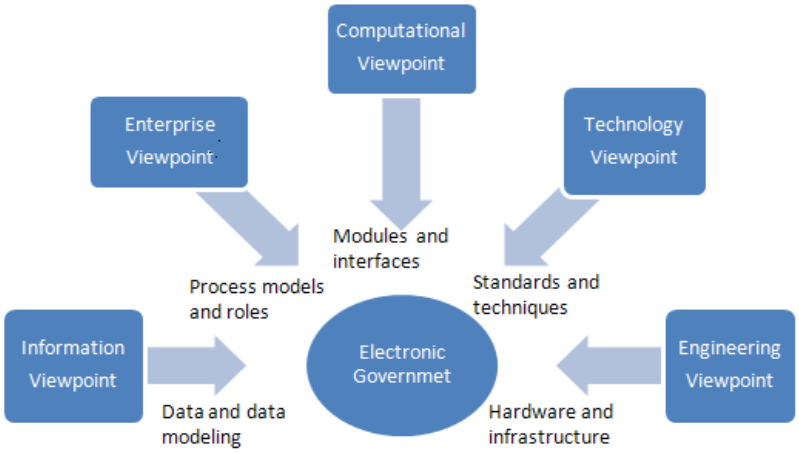
Uno de los ejemplos más conocidos de Arquitecturas Empresariales adoptadas por gobiernos es SAGA - Estándares y Arquitecturas para Aplicaciones de Gobierno Electrónico [KBST06], publicada y adoptada por el gobierno de Alemania. Las características principales de SAGA se pueden ver a continuación.

Ejemplo 13: Estándares y Arquitecturas para Aplicaciones de Gobierno Electrónico de Alemania – SAGA

SAGA provee un grupo de estándares para implementar Gobierno Electrónico. Para describir las aplicaciones de Gobierno Electrónico, se basa en el Modelo de Referencia de Procesamiento Distribuido Abierto (RM-ODP) [Ray93]. Siguiendo el enfoque RM-ODP, define las siguientes cinco perspectivas de modelado, las cuales se representan en la Figura 15:

1) *Punto de Vista de la Empresa* – define procesos y roles;
2) *Punto de Vista Computacional* – define módulos e interfaces;
3) *Punto de Vista Técnico* – define estándares y técnicas;
4) *Punto de Vista de la Ingeniería* – define hardware e infraestructura; y
5) *Punto de Vista de la Información* – define datos y modelos de datos

Figura 15: Estándares y Arquitecturas para Aplicaciones de Gobierno Electrónico de Alemania – SAGA

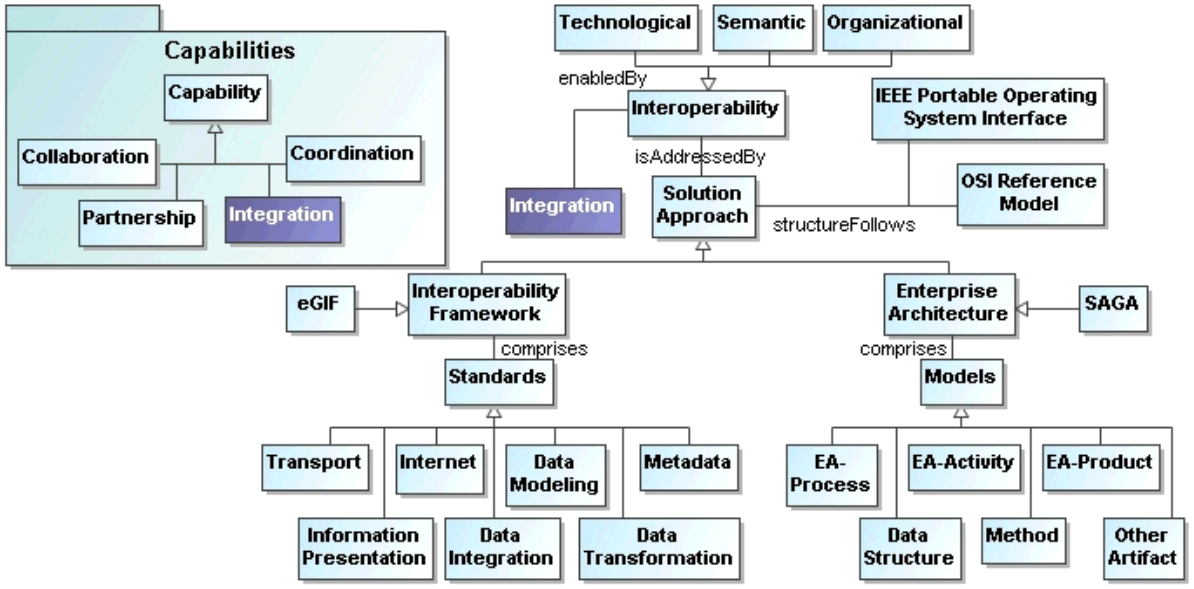


Para poder comprender mejor los estándares de interoperabilidad, estos fueron clasificados en [Gui04]: aquellos que se basan en modelo OSI – modelo de referencia de Interconexión de Sistemas Abiertos [Mil81] con estándares organizados en capas (como eGIF), o los que se basan en la Interface de Sistema Operativo Portable de la IEEE [IEEE98] – con estándares organizados alrededor de servicios (como SAGA).

La Figura 16 representa los principales conceptos de dominio relacionados con el concepto de Integración. El paquete nombrado Capacidades ubica al concepto en el dominio general. La Interoperabilidad permite la Integración, y está clasificada en Técnica, Semántica y Organizativa. La Interoperabilidad es abordaba a través de Soluciones de Interoperabilidad. Una es Marco de Interoperabilidad y la otra es Arquitectura Empresarial. La primera comprende estándares para Transporte, Presentación de Información, Estándares de Internet, Integración de Información, Modelado de Información, Transformación de Información y Metadatos. La segunda comprende modelos para Procesos, Estructuras de Información, Actividades, Métodos, Productos, etc. eGIF es un ejemplo muy conocido de Marco de Interoperabilidad, mientras que SAGA es un ejemplo de Arquitectura Empresarial. Las distintas soluciones

de interoperabilidad se pueden estructurar siguiendo una arquitectura de capas (modelo de referencia OSI) o una arquitectura basada en servicios (Interface de Sistema Operativo Portable de la IEEE).

Figura 16: Conceptos de Dominio Relacionados con Integración



El siguiente ejemplo ilustra un esfuerzo de integración de Estonia [EU08].

Ejemplo 14: Arquitectura de Gobierno Electrónico de Estonia

La estrategia para el desarrollo de Gobierno Electrónico en Estonia fue primero desarrollar una arquitectura funcional para definir un transporte seguro de datos, funcionalidad básica para sistemas de información distribuidos, y varios componentes de hardware y software como portales, infraestructura de clave pública (PKI), bases de datos de gobierno y sistemas de información. Esta arquitectura constituyó la base para el desarrollo de cientos de servicios que surgieron más tarde.

La arquitectura fue desarrollada dentro del marco del proyecto X-Road, con el objetivo de interconectar bases de datos del gobierno de Estonia. En la primera etapa, consultas a las bases de datos y las respuestas fueron comunicadas a través del ambiente de X-Road. En la segunda etapa, todo tipo de documento electrónico escrito en XML es intercambiado de manera segura utilizando Internet a través del ambiente del X-Road. Al mismo tiempo, X-Road se convirtió en el esqueleto para los servicios de Gobierno Electrónico. Las tres capas de la arquitectura, de la inferior a la superior, son:

1) *Bases de Datos* – Incluye varias bases de datos para mantener registros de pasaportes, registros de población, registros de tráfico, etc, soportados por bases de datos Oracle, Progress y MySQL.
2) *Información de Tráfico* – Soportado por X-Road para intercambiar mensajes usando SOAP y XML Remote Procedure Call. Los mensajes son intercambiados por los componentes principales, como servidores de seguridad, servidores centrales y portal del ciudadano.
3) *Servicios* – Soporta varios SPE como Beneficio para Padres, Mi Vehículo y Mis Multas. Los servicios son descriptos usando WSDL y publicados usando UDDI.

El marco fue usado para desarrollar los servicios de tarjetas de identidad de Estonia, usando tecnología PKI para identificación, autorización y firmas digitales. También se usó para desarrollar portales web para ciudadanos, funcionarios y empresarios, con alrededor de 700 SPE ofrecidos por gobiernos centrales y locales. Por ejemplo, Beneficio para Padres integra información de cinco sistemas: (1) Portal del Ciudadano, (2) Registro de Seguridad Social, (3) Registro de Población, (4) Sistema de Información del Fondo de Seguro de Salud , y (5) Sistema de Información de Impuestos y Aduana.

2.3.4 Coordinación

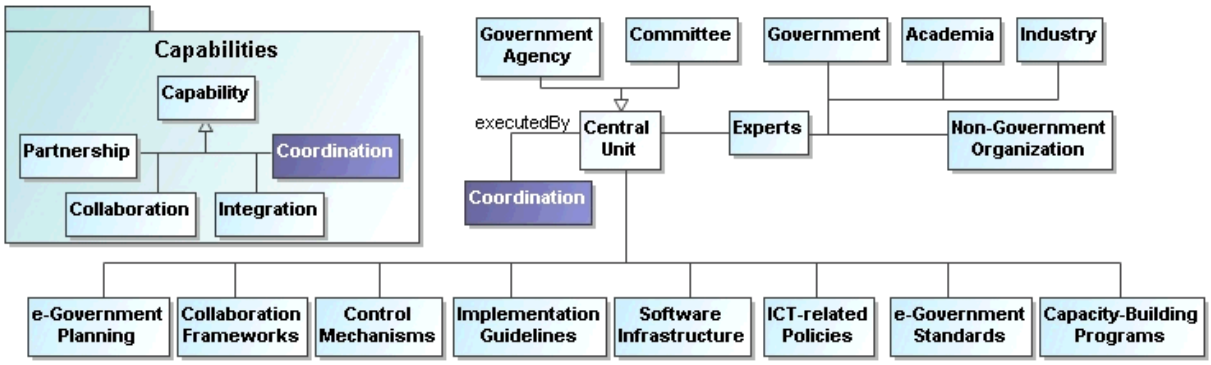
Si bien Gobierno Integrado promueve la transformación de agencias de gobierno, de jerarquías a redes, es muy difícil para ellas lograr el nivel de transformación requerido sin el apoyo organizativo adecuado. La experiencia de líderes de Gobierno Electrónico muestra claramente la necesidad de centralizar la planificación, liderazgo y administración. Australia, Alemania, Nueva Zelanda, Corea del Sur, Reino Unido y los Estados Unidos han creado oficinas centrales de coordinación para liderar, administrar y promover iniciativas de Gobierno Electrónico. Una unidad central de coordinación es también una característica de las estrategias de Gobierno Electrónico promovido por la OECD [FML03].

Generalmente, una unidad central de coordinación se encuentra dentro de la administración pública o conectada a sociedad de información más amplia. Dos enfoques comunes usados para organizar unidades centrales de coordinación son: (1) una agencia de gobierno dedicada a la coordinación de Gobierno Electrónico o (2) un comité compuesto de directores de agencias o Chief Information Officers (CIOs). Cualquiera sea el enfoque, la participación de representantes de organizaciones industriales, académicas o no gubernamentales en la estructura central de coordinación ha demostrado ser efectiva [EJOK07].

Típicas responsabilidades asignadas a una unidad central de coordinación son [EJOK07]: orientar y controlar la implementación en todo el gobierno de proyectos de Gobierno Electrónico, crear un marco para la colaboración entre agencias para asegurar interoperabilidad y la no duplicación de esfuerzos y recursos, desarrollar infraestructura compartida de Gobierno Electrónico, mantener las actividades de Gobierno Electrónico alineadas a visiones y estrategias de gobierno más amplias, y desarrollar e implementar políticas de gobierno relacionadas al uso de TIC.

La Figura 17 presenta los principales conceptos de dominio relacionados con el concepto de Coordinación. El paquete nombrado Capacidades ubica al concepto en el dominio general. La Coordinación se ejecuta generalmente por una Unidad Central, que puede ser una Agencia o Comité. La Unidad Central tiene la responsabilidad de: Planificar iniciativas de Gobierno Electrónico, diseñar y adoptar Marcos de Interoperabilidad, Mecanismos de Control y Guías de Implementación; definir Políticas relacionadas con TIC, Estándares de e-Gobierno, y Programas de Capacitación. Puede incluir o recibir consejos de Expertos de Organizaciones de Gobierno, Académicas, Industriales o No Gubernamentales.

Figura 17: Conceptos de Dominio Relacionados con Coordinación



El Ejemplo 15 presenta un estudio de coordinación central de Australia

Ejemplo 15: Coordinación de TIC en Australia

El gobierno de Australia estableció en el año 2004 la Oficina de Gestión de Información del Gobierno Australiano (AGIMO) dentro de su Departamento de Finanzas y Administración, con la misión de convertir a Australia en un líder en la aplicación productiva de TIC a la administración, información y servicios de gobierno. Algunas de las funciones de AGIMO son:

- Trabajar con agencias para desarrollar estándares que integren servicios cruzando los límites de las agencias;
- Promover servicios públicos mejorados a través de técnicas de interoperabilidad y la integración de procesos de negocios;

- Introducir nuevos enfoques para distribuir información, publicaciones, servicios y programas de gobierno;
- Desarrollar y mejorar procesos de compras electrónicas en gobierno;
- Promover convenios de telecomunicaciones a través de todas las dependencias gubernamentales.
- Identificar y promover el desarrollo de infraestructura de TIC necesario para implementar las estrategias emergentes de todo el gobierno Australiano;
- Desarrollar un Marco de Autenticación de Gobierno Electrónico para asistir a ciudadanos y empresas en la verificación de comunicaciones electrónicas; y
- Administrar directorios en línea e impresos, manteniendo la totalidad de los sitios web de gobierno, y proveer guías para el uso en línea de la marca del gobierno australiano.

El sitio web de AGIMO es http://www.agimo.gov.au/.

2.4 Recursos

Las Agencias (Sección 2.1) necesitan Recursos para poder entregar los Resultados esperados (Sección 2.2) al mismo tiempo que utilizan Capacidades existentes y desarrollan otras nuevas (Sección 2.3). Si bien varios tipos de Recursos son requeridos, el enfoque de la tesis, en el soporte tecnológico para Gobierno Integrado, determina la selección de los recursos de mayor interés - Procesos y Tecnología. El primero se encarga de la ejecución colaborativa de varias actividades y tareas de diferentes agencias y organizaciones asociadas para producir y entregar servicios a los clientes. El segundo se encarga de software y sistemas de apoyo para permitir la ejecución eficiente de procesos inter-organizacionales, incluyendo la automatización de los pasos de los procesos de negocio, la integración de aplicaciones de software heterogéneas, la coordinación de la ejecución de procesos de negocios más allá de los límites organizacionales, y la innovación en la forma de entregar servicios. Los conceptos relacionados con los procesos están presentados en la Sección 2.4.1 y los conceptos relacionados con la tecnología están presentados en la Sección 2.4.2.

2.4.1 Procesos

La producción y entrega de un Servicio es apoyado por un Proceso de Negocios, definido a continuación [MWC99].

Definición 15: Proceso de Negocios

Un Proceso de Negocios es una serie de actividades de negocios, cada actividad dividida en tareas, que crean valor al transformar una entrada en una salida de mayor valor.

Ilustramos los conceptos principales en esta definición – actividades, tareas, entradas y salidas; en base al caso de estudio de Licencias de Negocios en el Ejemplo 1. El proceso de negocios representado en la Figura 3, muestra entre otras, las siguientes actividades: (1) `Solicitud` – presentación de un formulario y documentos necesarios; (2) `Evaluación de Completitud` – revisar el formulario y los documentos para ver si están completo; y (3) `Toma de Decisión` – tomar la decisión de otorgar o denegar una licencia. Cada una de estas actividades es ejecutada a través de secuencias de tareas relacionadas. Por ejemplo `Evaluación de Completitud` está dividida en: (2.1) revisar que la información en el formulario esté completa, (2.2) revisar que la foto de la fachada y el reporte del registro de propiedad hayan sido presentados y (2.3) revisar que todos los demás documentos necesarios hayan sido presentados y que todos estén completos. Las entradas requeridas incluyen: el formulario de aplicación, los documentos necesarios, y opiniones recibidas de otras agencias gubernamentales. Las salidas producidas por el proceso en su conjunto incluyen la notificación al postulante acerca del resultado del proceso y, si fue positivo, la licencia de negocios. El valor creado a través del proceso de negocios incluye la licencia que permite al postulante abrir un establecimiento de comidas y bebidas, y la garantía al público de que el nuevo establecimiento cumple con todas las reglamentaciones vigentes.

El objetivo para el apoyo de las TIC a los procesos de negocios de gobierno ha cambiado a través de los años. En los ´70, el uso de las TIC se concentró en la directa automatización de actividades y tareas manuales ejecutadas en unidades de gobierno. Luego, su objetivo fue el de mejorar los procesos de negocios previamente automatizados, en

especial para aumentar la eficiencia y para enfocarse en las necesidades de los clientes. Esto dio origen al enfoque administrativo que permitió rediseñar procesos de negocios de una manera controlada, llamado Reingeniería de Procesos de Negocios (RPN). El objetivo de RPN es simplificar los procesos, en particular las tareas que no crean valor agregado, e integrar a los clientes en la producción automatizada de valor. La definición es la siguiente [HC06].

Definición 16: Reingeniería de Procesos de Negocios (RPN)

RPN involucra el replanteamiento fundamental y el rediseño radical de los procesos de negocios para obtener dramáticas mejoras en las actuales medidas de desempeño críticas, como costo, calidad, servicio y velocidad.

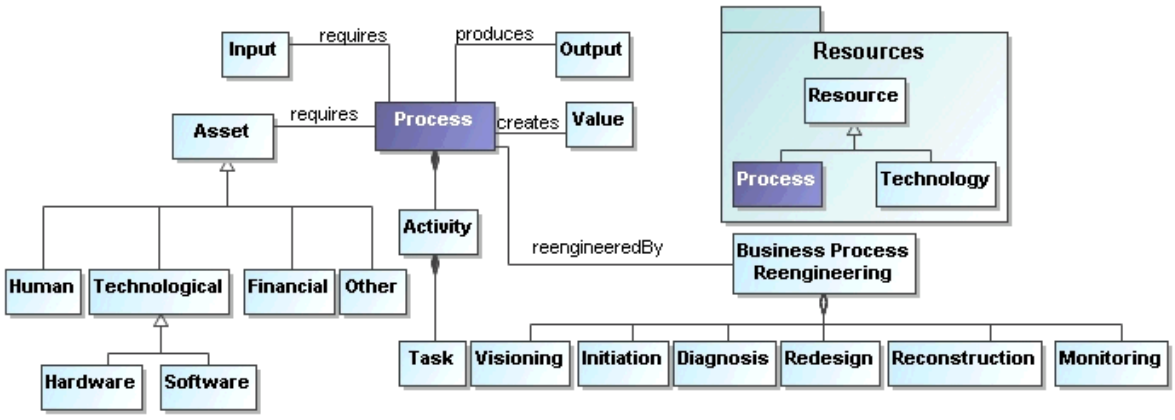
La RPN ha sido ampliamente utilizada en el Sector Público desde 1980s, cuando extensos programas de reformas fueron llevados a cabo bajo la era de NPM. La aplicación de RPN se convierte en crucial, casi obligatoria, para desarrollar Gobierno Integrado, a fin de: (1) implementar enfoque al cliente; (2) involucrar a nuevos socios colaboradores, como unidades de gobierno o compañías privadas; (3) eliminar tareas duplicadas ejecutadas por socios; (4) definir nuevas medidas de desempeño basadas en la adopción de nuevas soluciones; y (5) definir nuevos resultados basados en los requerimientos definidos por clientes y socios.

La metodología de RPN comprende seis etapas [KGT95]:

1) *Visión* – Crear consenso en la organización sobre la necesidad de procesos de reingeniería y cumplir con esta necesidad, a través de: construir apoyo gerencial, identificar las oportunidades de reingeniería, explicar las oportunidades creadas por las tecnologías avanzadas, y alinear soluciones tecnológicas con objetivos organizacionales más amplios.
2) *Iniciación* – Lanzar la actividad de reingeniería, definiendo objetivos de desempeño y creando un equipo de reingeniería.
3) *Diagnóstico* – Comprender el proceso de negocios existente (As Is), analizar su rendimiento, y evaluar como las TIC pueden contribuir para crear mayor valor.
4) *Rediseño* – Crear varias alternativas al proceso actual y que cumplan con los objetivos.
5) *Reconstrucción* – Implementar el proceso rediseñado superando los desafíos, como la resistencia al cambio.
6) *Monitoreo* – Monitorear el progreso de la actividad de reingeniería.

La Figura 18 ilustra los principales conceptos del dominio relacionados con el concepto de Proceso de Negocios. El paquete nombrado Recursos ubica al concepto en el dominio general. Un Proceso requiere algunas Entradas y Activos para producir Salidas y eventualmente para crear un Valor. Un Proceso comprende varias Actividades, y cada Actividad puede dividirse en varias Tareas. Los Activos requeridos pueden ser: Humanos, Tecnológicos (Hardware/Software) Financieros, y Otros. Un proceso es reorganizado a través de una metodología llamada RPN, que incluye las siguientes etapas: (1) Visión, (2) Iniciación, (3) Diagnóstico, (4) Rediseño, (5) Reconstrucción, y (6) Monitoreo.

Figura 18: Conceptos de Dominio Relacionados con Proceso



El Ejemplo 16 presenta un caso de estudio de RPN implementado por el Departamento de Hacienda de Malta [WD04].

Ejemplo 16: Reingeniería de Procesos de Negocios en Malta

El Departamento de Hacienda de Malta provee un paquete de servicios electrónicos a través de su sitio web: http://www.ird.gov.mt/. El portal ofrece servicios a contribuyentes, empleados y profesionales impositivos, incluyendo la presentación en línea de formularios, tales como el de Reembolsos de Impuestos y el de Declaración de Ganancias de los Empleados, y confirmaciones en línea sobre la recepción y validez de los mismos. El objetivo es intentar reducir el esfuerzo asociado a la presentación de formularios impositivos.

La provisión de estos servicios ha sido producto de un esfuerzo extenso de RPN, llevado a cabo en el Departamento de Hacienda, logrando los siguientes resultados:

- La fusión de los procesos de pagos y conciliación para los Impuestos Sobre la Renta y las Contribuciones al Seguro Social.
- Implementación del Sistema de Liquidación Final (FSS) basado en el concepto de retención de impuestos para trabajo e inversiones. Esto, sucesivamente, eliminó la necesidad de presentar la declaración de impuestos por parte de empleados cuyos ingresos provienen solamente de sus empleadores, ya que los impuestos han sido descontados en la fuente de origen.
- Reorganización de los procesos de back office para reestructurar la manera en que los estados contables y otros documentos financieros son presentados al Departamento de Haciendo y mantenidos por el Departamento.
- Importantes beneficios obtenidos por contribuyentes y el Departamento de Hacienda. Por ejemplo, los contribuyentes se beneficiaron a través de la reducción de interacciones requeridas con la Administración Pública, resultando en una importante reducción de costos para hacer negocios. Además, el Departamento de Hacienda aumentó su eficiencia en el cobro de impuestos y disponibilidad de información para contribuyentes.

Las últimas estadísticas muestran que un 83% de las declaraciones de impuestos corporativas han sido presentadas en línea, mientras que el 63% de las organizaciones con más de nueve empleados, eligieron los servicios en línea.

2.4.2 Tecnología

Muchos problemas tecnológicos deben ser resueltos cuando se implementa Gobierno Integrado. Por ejemplo, la parte de front office de los SPE requiere la integración de portales de acceso único con los sistemas de entrega por múltiples canales. El back office requiere la integración de aplicaciones de software que soportan la entrega del servicio de manera colaborativa y la coordinación de procesos de negocios inter-organizacionales. Adicionalmente, se requieren soluciones tecnológicas para proveer infraestructura con funcionalidad común usada en la entrega de SPE, por ejemplo, autenticación y notificación.

Entre la gran variedad de soluciones tecnológicas que se pueden utilizar mientras se implementa Gobierno Integrado, esta sección se enfoca en las que pueden contribuir a solucionar los problemas principales mencionados en la tesis - apoyo tecnológico para la colaboración en la entrega de Servicios Integrados. En secuencia, presentamos tres familias de tecnología relacionadas: Middleware Orientado a Mensajes (MOM), Arquitectura Orientada a Servicio (SOA) y Tecnologías Semánticas (Ontologías) Estas tecnologías son usadas directamente en la tesis. Específicamente, la solución técnica propuesta en esta tesis (G-EEG) es un ejemplo de MOM, y su única implementación [DEJ08] depende de Servicios Web (una implementación especial de SOA) para la comunicación, y de Ontologías para la interoperabilidad semántica.

MOM permite a aplicaciones de software intercambiar información asincrónicamente en forma de mensajes. Al utilizar una interfaz de programación de aplicaciones (API) suministrado por el MOM, las aplicaciones de software pueden producir y consumir mensajes, y transferirlos a través de colas de mensajes. La definición de MOM se presenta a continuación [SEI07].

Definición 17: Middleware Orientado a Mensajes (MOM)

MOM es un software que reside en ambas partes de la arquitectura cliente/servidor. Generalmente soporta llamados asincrónicos entre las aplicaciones cliente y servidor, con colas de mensajes que proveen un almacenamiento temporal cuando el programa de destino está ocupado o no conectado.

Un ejemplo concreto de la implementación de MOM es Java Message Service (JMS) [Sun08]. JMS provee servicios de mensajes a aplicaciones de Java a través de una API estándar adoptada por la industria. Productos comerciales muy conocidos basados en MOM también están disponibles, como el MSMQ [Dic98] de Microsoft, IBM WebSphere [IBM08] y WebMethods Enterprise [SAG08]. Además de los servicios de mensajes, estos productos ofrecen funcionalidades típicas requeridas por las aplicaciones de software que se comunican, como la autenticación, cifrado y descifrado, y enrutamiento de mensajes.

También existen varias soluciones basadas en MOM desarrolladas para resolver necesidades específicas de aplicaciones de Gobierno Electrónico, por ejemplo: (1) Government Gateway implementado por Microsoft como parte del Connected Government Framework [Mic07] que permite la autenticación de postulantes, autorización para acceder a servicios, presentación de documentos e integración de aplicaciones de software; (2) Inter-Agency Messaging Service [Col03] desarrollado por el gobierno de Irlanda para permitir el intercambio electrónico de información relacionada a eventos de vida de los ciudadanos, como el nacimiento, el matrimonio y fallecimiento, entre otros; y (3) Hermes Messaging Gateway [CEC07] desarrollado para el gobierno de Hong Kong SAR, permite la comunicación entre aplicaciones de software, a través del intercambio de mensajes en ebXML.

Aunque las soluciones basadas en MOM demostraron ser útiles para la conexión de aplicaciones, ofrecen una funcionalidad de mensajería fija, o al menos estática, incluyendo autenticación de usuarios, validación, cifrado y descifrado y enrutamiento de mensajes. Como alternativa, SOA y Servicios WEB ofrecen marcos para la construcción e integración de aplicaciones basadas en servicios. Los marcos permiten agregar la funcionalidad requerida en la forma de Servicios Web.

A continuación, se presenta la definición de SOA [W3C04a]

Definición 18: Service-Oriented Architecture (SOA)

SOA es una arquitectura de software, un modelo para componentes de software y sus relaciones, donde todas las tareas y procesos implementados por software son diseñados como servicios a ser consumidos a través de una red.

SOA es ampliamente adoptada como una solución tanto para la integración de información como para la integración de procesos: la integración de información está soportada al permitir que aplicaciones basadas en SOA, ejecutadas en diferentes plataformas tecnológicas, puedan intercambiar mensajes usando estándares abiertos; la integración de procesos está soportada al permitir componer complejos servicios basados en SOA a partir de simples servicios, y ejecutar los mismos a través de la orquestación de servicios. Un servicio - un recurso abstracto provisto y consumido a través de una red, que tiene la habilidad de ejecutar tareas [W3C04b], es el bloque básico de construcción de SOA.

Actualmente, los Servicios Web y sus tecnologías subyacentes se han convertido en el paradigma de implementación más común para SOA. A continuación, presentamos la definición de Servicios Web [W3C04b].

Definición 19: Servicio Web

Un Servicio Web es un sistema de software designado para soportar la interacción interoperable de máquina a máquina a través de una red. Tiene una interface descripta en un formato interpretable por una computadora - Web Services Description Language (WSDL), y puede interactuar con otros sistemas, de la manera que se especifica en su descripción, usando Simple Object Access Protocol (SOAP), con mensajes de SOAP serializados usando XML y transportados usando HTTP, y otros estándares Web involucrados.

Los tres estándares técnicos principales usados por los Servicios Web están basados en XML e incluyen:

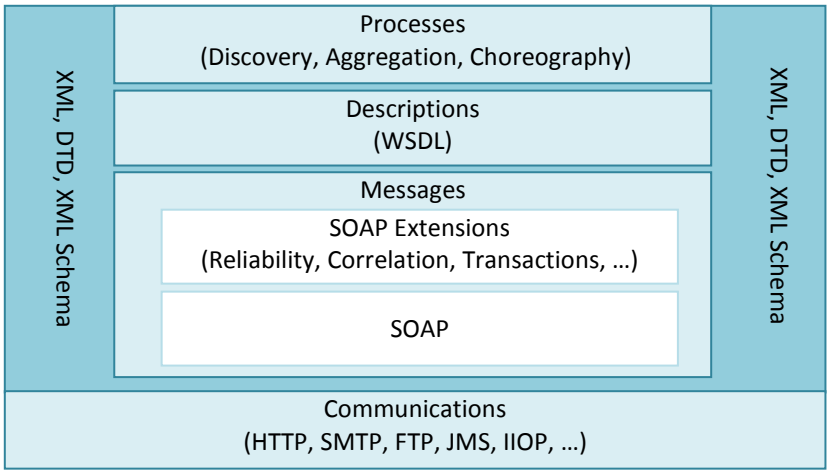
- *Simple Object Access Protocol* (SOAP) [W3C03] - Un protocolo liviano para el intercambio de información estructurada en un ambiente descentralizado y distribuido. SOAP define un marco extensible de mensajería que provee una construcción para los mensajes, de tal modo que los mismos puede ser intercambiados por una variedad protocolos de comunicación. SOAP ha sido diseñado para ser independiente de cualquier modelo de programación u otras semánticas específicas de implementación.

- Web Services Description Language (WSDL) [W3C01] - Un lenguaje XML que describe tres aspectos principales de un Servicio Web: (1) lo que hace un servicio - las operaciones llevadas a cabo con sus argumentos y resultados; (2) como se accede a un servicio - detalles acerca de formatos y protocolos a ser usados para acceder a sus operaciones; y (3) donde se encuentra un servicio - detalles de direcciones de red específicas de los protocolos, como, por ejemplo, la URL.

- *Universal Description, Discovery and Integration (UDDI)* [W3C01] - Un marco abierto para describir, descubrir e integrar servicios de negocios. Las especificaciones UDDI definen a un registro de Servicios Web y otros servicios electrónicos y no electrónicos. La información acerca de servicios está dividida en tres componentes: (1) páginas blancas – proveyendo direcciones, contactos e identificadores conocidos; (2) páginas amarillas – listado de clasificaciones industriales basadas en taxonomías; y (3) páginas verdes – proveyendo información técnica acerca de servicios.

Las tecnologías de Servicios Web se ordenan en una pila. En la capa de abajo, la comunicación se realiza a través de una variedad de protocolos, como HTTP, SMTP, FTP, JMS. Por encima de esta capa, una capa orientada a mensajes permite el intercambio de mensajes SOAP. En esta capa, funcionalidad adicional puede ser introducida por varias extensiones de SOAP, como por ejemplo, extensiones para asegurar confiabilidad y correlación de mensajes, implementar semántica de transacción, etc. La siguiente capa permite describir servicios usando WSDL. Finalmente, la capa de arriba permite descubrir y componer servicios. La base tecnológica para todas las capas es XML [W3C08], Document Type Definition (DTD) [W3C99b] y XML Schema [W3C04c]. La arquitectura se representa en la Figura 19.

Figura 19: Familia de Tecnologías de Servicios Web



[Cortesía W3C]

Dos fortalezas claves de SOA y su implementación mediante Servicios Web son: la adopción de XML para el intercambio de mensajes – facilitando interoperabilidad técnica, y la provisión de una arquitectura débilmente acoplada – facilitando el mantenimiento de las aplicaciones. Sin embargo, las mismas fortalezas pueden dar lugar a dos desafíos. El primero es la degradación de performance debido al uso de mensajes XML muy extensos durante las invocaciones de servicios web, especialmente en combinación con extensiones como WS Security [OAS06]. El segundo es que la arquitectura débilmente acoplada, que mantiene conexiones flexibles entre servicios, puede volverse inmanejable si no hay pautas claras para el uso y la composición de servicios.

Para la integración de procesos, SOA provee soporte para solucionar el problema de interoperabilidad en tres niveles:

técnico, semántico y organizacional. La Interoperabilidad Técnica es facilitada a través de adaptadores que elevan servicios arbitrarios a un protocolo de servicio común, con adaptadores especiales usados para convertir formatos de datos de sistemas legados a un protocolo común. La Interoperabilidad Semántica puede ser provista por servicios web especializados a través de tecnologías como, por ejemplo, eXtensible Stylesheet Language Transformations (XSLT) [W3C99a] o Tecnologías Semánticas como se explicada debajo. La Interoperabilidad Organizativa es soportada a través de la integración de servicios basándose en sistemas de workflow distribuidos, por ejemplo, Business Process Execution Language (BPEL) (BPEL) [OAS03]. Otro enfoque para apoyar a la Interoperabilidad Organizativa es a través de la Coreografía de Servicios Web, con reglas explícitas de interacción, como las expresadas en Web Services Choreography Description Language (WS-CDL) [W3C04d], aplicadas por todos las partes que colaboran.

SOA es usada en muchas soluciones de Gobierno Electrónico, como por ejemplo, Microsoft's Connected Government Framework [Mic07] y la infraestructura para la integración de servicios [TT05]. También se usa en el proyecto EU-Publi.com [CUI06] que define la arquitectura para facilitar la colaboración entre los empleados de las Administraciones Públicas de la Unión Europea.

En los últimos años, surgió una nueva área de investigación que combina Tecnologías Semánticas y la World Wide Web, llamada Web Semántica. La Web Semántica intenta proveer semántica a los datos que sea procesable por una computadora, a fin de facilitar el acceso a la información y poder hacer un mejor uso de los datos disponibles, siendo la Ontología el principal concepto subyacente para administrar el conocimiento de manera estructurada. A continuación se presenta la definición de ontología [Gru92].

Definición 20: Ontología
Ontología es una especificación de un vocabulario representacional para un dominio de discurso compartido – definiciones de clases, relaciones, funciones y otros objetos.

Las dos primeras tecnologías usadas para proveer semántica a los datos y permitir la construcción de Ontologías fueron el Resource Description Framework (RDF) [W3C04e] y Web Ontology Language (OWL) [W3C04f]. Ambos son lenguajes que representan información acerca de recursos en la Web. La mayor diferencia es que OWL provee una semántica formal para datos y más vocabulario que RDF y el RDF Schema (RDF-S) [W3C04g] asociado. Como resultado, facilita la interpretación automática de contenido web. OWL comprende tres sub-lenguajes con orden creciente de expresividad: OWL Lite – apropiado para describir jerarquías de clasificación y principales restricciones; OWL DL – garantiza máxima expresividad mientras mantiene completitud y decidibilidad computacional; y OWL Full – garantiza máxima expresividad y libertad sintáctica pero no garantiza completitud y decidibilidad computacional.

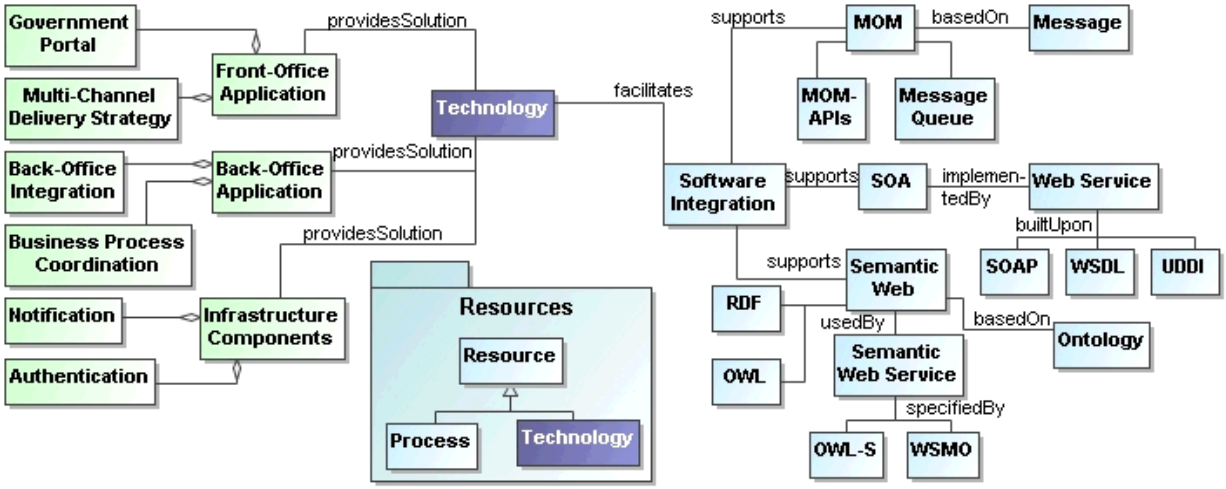
De la misma manera que la Web Semántica mejora la web al representar la semántica de datos explícitamente, puede también mejorar los Servicios Web al representar semánticamente la funcionalidad ofrecidas por ellos, resultando en el concepto de Servicios Web Semánticos. Existen varios marcos para representar Servicios Web Semánticos. Por ejemplo: OWL-based Web Services Ontology (OWL-S) [DAML07] y Web Service Modelling Ontology (WSMO) [W3C05]. OWL-S es una ontología basada en OWL que facilita la tarea de describir, descubrir, invocar, componer y monitorear Servicios Web. La Ontología se compone de tres partes: (1) pérfil del servicio – describiendo la funcionalidad del servicio, incluyendo sus limitaciones y la calidad del servicio (QoS); (2) modelo del servicio – ilustrando cómo puede usarse el servicio; y (3) Uso del Servicio – especificando cómo es posible comunicarse con el servicio usando protocolos de comunicación y formatos de mensajes. WSMO tiene el mismo objetivo que OWL-S, pero también provee descripciones semánticas para peticiones de servicios (objetivos) y mediadores. Provee cuatro elementos de modelado: Ontologías, Servicios Web Semánticos, Objetivos y Mediadores.

Recientemente, la aplicación de tecnologías de la Web Semántica al dominio de Gobierno Electrónico ha aumentado el interés de varios grupos de investigación. Por un lado, el Gobierno Electrónico puede proveer un banco de pruebas interesante para la investigación de la Web Semántica. Por otro lado, la Web Semántica puede permitir el desarrollo de soluciones poderosas y avanzadas para Gobierno Electrónico. Por ejemplo, capturando explícitamente la semántica de datos y servicios, se puede facilitar la integración y el descubrimiento de recursos web distribuidos que son usados en la implementación de Gobierno Integrado. Además, Gobierno Electrónico difiere del dominio de negocios electrónicos tradicionales, por el alto grado de formalidad y confiabilidad requeridos por las aplicaciones de gobierno

en áreas tales como: ley, regulaciones, seguridad, impuestos, etc. Por ende, el dominio de aplicación de Gobierno Electrónico provee casos de estudio interesantes para el desarrollo y uso de la Web Semántica [SGC06].

La Figura 20 representa los principales conceptos del dominio relacionados con el concepto de Tecnología, que se describen en esta sección. El paquete nombrado Recursos ubica al concepto en el dominio general. En la parte izquierda del diagrama hay ejemplos de requerimientos técnicos, típicos de Gobierno Electrónico que cubren: Aplicaciones de Front Office – Portal de Gobierno y Servicios de Entrega a través de Múltiples Canales; Aplicaciones de Back Office – Integración de Back Office y Coordinación de Procesos de Negocios; y Componentes de Infraestructura – Notificación y Autenticación. El lado derecho del diagrama muestra soluciones técnicas disponibles para encarar el problema de Integración de Software: Middleware Orientado a Mensajes (MOM), Arquitectura Orientada a Servicios (SOA) y tecnologías de la Web Semántica en base a Ontologías. MOM facilita el intercambio de Mensajes, es apoyado por Colas de Mensajes, y provee servicios a las aplicaciones de software a través de Interfaces (MOM-APIs). SOA es implementada por Servicios Web, construidos sobre tecnologías como SOAP, WSDL y UDDI. La Web Semántica es implementada usando RDF y OWL, y se extiende hacia Servicios Web Semánticos mediante OWL-S y WSMO.

Figura 20: Conceptos de Dominio Relacionados con Tecnología



Muchos ejemplos pueden ser provistos para el uso de tecnología como soporte para el desarrollo de Gobierno Integrado. El Ejemplo 17 fue elegido debido a su parecido con el problema presentado en la tesis. Presenta un servicio de mensajería que apoya la comunicación entre agencias de gobierno de Irlanda [Rea02][Col03].

Ejemplo 17: Servicios de Mensajería Inter-Agencias de Irlanda – IAMS

El Servicio de Mensajería Inter-Agencias (IAMS) soporta el intercambio de información relacionada a eventos de vida ente la Oficina de Registro General (GRO); Departamento de Asuntos Sociales, Comunitarios y Familiares (DSCFA); Oficina Central de Estadísticas (CSO); hospitales maternales y otras agencias de gobierno. El objetivo de IAMS es: (1) soportar intercambios claves de información relacionada a clientes, (2) proveer una infraestructura de mensajería única para solucionar problemas de interoperabilidad, (3) imponer estándares para datos y mensajería, (4) aumentar la capacidad para capturar y reutilizar información, y (5) estimular la confianza en las iniciativas de Gobierno Electrónico.

Estos objetivos son implementados de la siguiente manera: GRO mantiene a través de IAMS los detalles y enmiendas relacionadas a nacimientos, matrimonios, adopciones, nacidos muertos, divorcios y fallecimientos. La información relacionada a nacimientos es cargada a IAMS por los hospitales maternales y por DSCFA a través de navegadores web. GRO está legalmente obligada a proveerle a CSO, a través de IAMS, estadísticas de nacimientos, fallecimientos y matrimonios. CSO, junto con otras agencias gubernamentales, puede acceder a información provista por IAMS a través de navegadores web.

IAMS se basa en:

- XML – intercambio de mensajes;
- Estándares Abiertos – definición de datos y estándares de mensajes;
- Software de Código Abierto – usa soluciones concretas de software de código abierto, por ejemplo, Tomcat, como contenedor de servlets, Jboss, como servidor de aplicaciones, y MySQL, como motor de la base de datos; y
- Software Propietario – usa Microsoft BizTalk para manejar el proceso de negocios subyacente.

La funcionalidad básica implementada por IAMS es la siguiente: GPO acepta un mensaje de un evento de vida del servidor GRO BizTalk, valida el mensaje, y envía un recibo confirmando la recepción del mensaje. Si el mensaje es válido, es enviado a DSCFA, de lo contrario devuelve un mensaje de error. Funcionalidades específicas, como logging y confiabilidad de mensajees, son provistas por BizTalk.

2.5 Desafíos

Las secciones anteriores de este capítulo introdujeron el grupo de conceptos que define el dominio de Gobierno Integrado, desde Agencia (Sección 2.1) y los Resultados que se espera que produzca (Sección 2.2), a través de las varias Capacidades de las cuales una Agencia depende para participar en trabajos colaborativos (Sección 2.3), a los Recursos requeridos para producir los Resultados (Sección 2.4). Adicionalmente a esto, una Agencia debe superar ciertos desafíos bien conocidos, particularmente para la implementación de Gobierno Integrado. Algunos de estos desafíos fueron introducidos en la Sección 1.2: la falta de reconocimiento legal para procesos y documentos electrónicos, la falta de marcos presupuestarios para proyectos llevados a cabo por varias agencias, la falta de recursos humanos calificados, la falta de marcos y modelos con los cuales poder trabajar con el sector privado, la falta de herramientas de integración de software para resolver necesidades peculiares de los ambientes de TIC de gobierno, etc. Tales desafíos se pueden clasificar generalmente en: legales, financieros, sociales, organizacionales y tecnológicos. Esta sección explica en detalle los desafíos organizacionales (Sección 2.5.1) y tecnológicos (Sección 2.5.2) [EJ07]. Estos desafíos se presentan en la Tabla 5.

Tabla 5: Desafíos Tecnológicos y Organizacionales

T1	Acceso Único	O1	De Jerarquías a Redes
T2	Procesos Inter-Organizacionales	O2	Integración Horizontal
T3	Monitoreo de Cumplimiento de Políticas	O3	Integración Vertical
T4	Integración de Aplicaciones	O4	Integración Intersectorial
T5	Interoperabilidad Sintáctica	O5	Integración Inter-Sistemas
T6	Interoperabilidad Semántica		
T7	Subcontratación Flexible		
T8	Eco-sistema Dinámico		
T9	Entrega por Múltiples Canales		
T10	Requerimientos de Dependabilidad		

2.5.1 Desafíos Organizacionales

Para poder implementar Gobierno Integrado, el requerimiento clave es que las agencias puedan colaborar. Esto puede llevarse a cabo según distintos escenarios, cada uno presentando sus propios desafíos específicos. A continuación, exponemos cinco escenarios distintos: (01) De Jerarquías a Redes – esto permite la colaboración entre agencias en general; (02) Integración Horizontal – esto permite la colaboración entre agencias en diferentes áreas funcionales de gobierno; (03) Integración Vertical – esto permite la colaboración entre agencias ubicadas en diferentes niveles de gobierno; (04) Integración Intersectorial – esto permite la colaboración entre el sector público por un lado y los sectores privados y no gubernamentales por el otro; (05) Integración Inter-Sistemas – esto permite la colaboración entre diferentes administraciones públicas.

Cada uno de estos desafíos es explicado e ilustrado de la siguiente manera:

O1) *De Jerarquías a Redes* – La estructura vertical tradicional de gobierno presenta un desafío fundamental para Gobierno Integrado, y para la colaboración entre agencias en particular. Las barreras que impiden la colaboración generalmente incluyen: los funcionarios en los niveles más bajos de las estructuras de gobierno no tienen acceso a la información y no tienen poder para tomar decisiones, la información debe fluir hacia arriba y hacia abajo a través de la jerarquía para obtener autorizaciones y aprobaciones, las agencias son evaluadas únicamente por su desempeño individual y no están interesadas en el trabajo de otras agencias, etc. Como se explicó anteriormente, las TIC permiten la transformación de las estructuras de gobierno desde jerarquías a redes, ofreciendo naturalmente apoyo a la colaboración inter-agencias. Ejemplos de transformación son: empoderar a funcionarios de niveles más bajos en el gobierno con información y autoridad para tomar decisiones, y compartir datos e información entre agencias para reducir la carga de proveerlos a los ciudadanos y las empresas. Tales transformaciones involucran cambios a estructuras organizativas y procesos de negocios ejecutados por agencias de manera individual y colectiva. Dos enfoques útiles para encarar tales transformaciones son la Administración de Cambios [Kot96] y, como se explicó anteriormente, la Reingeniería de Procesos de Negocios (RPN).

Ejemplo 18: De Jerarquías a Redes para Licencias Empresariales

Reconsiderar Ejemplo 1 – emitir una licencia de negocios para administrar un establecimiento de alimentos y bebidas. Para que todas las agencias involucradas – IACM, DSAL, DSSOPT, CB, IC y SS realicen la entrega integrada de este servicio de manera conjunta, deben pasar por una transformación organizativa. En términos concretos, las agencias deben de forma conjunta: acordar los requerimientos para el servicio desarrollado, entender el proceso de negocios para la producción del servicio, determinar las responsabilidades individuales de cada agencia en los diferentes pasos de este proceso, y facilitar los cambios necesarios en las agencias para implementar este proceso. Como resultado, el proceso de negocio será ejecutado por todas las agencias de manera colaborativa, posiblemente con el soporte necesario para automatizar las interacciones entre ellas.

O2) *Integración Horizontal* – Cuando un cliente solicita un servicio público en una agencia y este servicio requiere interacciones con otras agencias, el cliente es el que tradicionalmente tiene que visitar cada agencia involucrada para recolectar información y documentos requeridos por las agencias que emiten el servicio. El proceso puede ser simplificado y mejorado desde el punto de vista de los clientes, siempre y cuando las agencias responsables de diferentes partes del proceso subyacente (perteneciendo generalmente a diferentes áreas funcionales de gobierno) puedan colaborar en la entrega del servicio. Este escenario se llama Integración Horizontal. El principal desafío para la integración horizontal es que las agencias ubicadas en diferentes áreas funcionales de gobierno puedan colaborar: dividir y asignar roles, coordinar la ejecución de procesos colaborativos, compartir información, etc. Superar este desafío requiere, entre otras cosas: identificar competencias centrales presentes dentro de las agencias, ponerlas a disposición de los procesos inter-organizacionales, llevar a cabo reingeniería del proceso de negocios mediante la asignación de los diferentes pasos a las agencias que mejor los puedan ejecutar, y proveer una infraestructura técnica para permitir la coordinación del proceso y el intercambio de información.

Ejemplo 19: Integración Horizontal para Licencias Empresariales

Considere el Ejemplo 1 nuevamente. Integración Horizontal significa que IACM puede recibir solicitudes de licencias y requerir opiniones técnicas a DSAL, DSSOPT, CB, IC y SS, quienes posteriormente pueden cumplimentar tales requerimientos:

- DSAL provee una opinión acerca de las condiciones laborales;
- DSSOPT provee una opinión técnica acerca de planos de construcción;
- CB inspecciona el edificio con respecto a la prevención de incendios y formula una evaluación;
- IC opina acerca de la apariencia del edificio; y
- SS inspecciona el lugar y da una opinión acerca de las condiciones sanitarias;

O3) *Integración Vertical* – Esto incluye la recolección de información por parte de un nivel de gobierno, y la disponibilidad de esta información a agencias ubicadas en otros niveles de gobierno. Por ejemplo, la información administrada a nivel central, por ejemplo, registro de ciudadanos, usada por los gobiernos provinciales para obtener información demográfica acerca de residentes, evitando así la duplicación de información y de procesos. Similarmente, la información recolectada por gobiernos locales, por ejemplo, relacionadas a salud, usada por un

gobierno central para desarrollar políticas relacionadas a la salud. El desafío principal para la implementación de tal intercambio de información es la habilidad de las agencias para recolectar y mantener actualizada la información de los clientes y para compartir esa información con otras agencias. Superar este desafío requiere estándares y políticas a lo largo de todo el gobierno para la recolección, representación y mantenimiento de información; regulaciones a lo largo de todo el gobierno para gobernar cómo se deben compartir los datos; y una infraestructura técnica que permita el intercambio de información entre aplicaciones ejecutadas en diferentes niveles de gobierno.

Ejemplo 20: Integración Vertical para el Control Relacionado a Impuestos de las Licencias de Negocios
Considere el Ejemplo 1, con dos niveles de gobierno. Suponga que la Autoridad Impositiva del nivel nacional provee un registro central de todas las empresas con información acerca de sus registros de impuestos, y una política requiere que para abrir un nuevo negocio, la empresa debe haber resuelto todas sus obligaciones tributarias. Siendo las licencias de negocios emitidas a nivel local, las aplicaciones de software responsables por la emisión de tales licencias deben interactuar con el registro central de la Autoridad Impositiva para comprobar que los postulantes cumplan con la política de no tener deudas impositivas.

O4) *Integración Intersectorial* – El sector privado está cada vez más involucrado en la entrega de servicios públicos a través de sociedades públicas-privadas, como se explicó en la Sección 2.3.2. El desafío principal para la integración entre sectores públicos y privados es garantizar que los servicios estén siendo entregados según reglamentaciones y estándares bien definidos, y que la calidad y la constancia de la entrega de servicios estén garantizadas. Para superar este desafío, las sociedades públicas-privadas deberían cubrir varios temas de colaboración, y adicionalmente, se requiere una plataforma tecnológica que permita la ejecución de procesos colaborativos y el monitoreo de la entrega de servicios.

Ejemplo 21: Integración Intersectorial para Solicitudes de Licencias de Negocios
Considere el Ejemplo 1. Suponga que el Portal de Gobierno (GP) está provisto por una empresa privada que también provee la aplicación de Front Office (FOP) para el servicio de licencias de negocios. La entrega de licencias en este escenario requiere integración intersectorial de GP y FOP, con la aplicación de Back Office en IACM, y con las agencias que contribuyen al servicio – DSAL, DSSOPT, CB, IC y SS. Integración Intersectorial también puede requerir el establecimiento de estándares de performance, por ejemplo que DSAL, DSSOPT, CB, IC y SS reciban pedidos de opiniones a más tardar un día después de la presentación de solicitudes a través de GP, y por ende medir el desempeño de performance de la organización del sector privado.

O5) *Integración Inter-Sistemas* – Se requiere Integración Inter-Sistemas cuando diferentes administraciones públicas entregan servicios públicos que cruzan los límites nacionales. Un ejemplo típico es la entrega transnacional de diferentes tipos de servicios públicos en la Unión Europea (UE). El desafío principal es superar la diversidad y heterogeneidad de legislaciones, procedimientos, lenguajes, culturas, etc. que caracterizan a distintos miembros de la UE. Esto en general requiere la adopción de estándares y políticas regionales, y el establecimiento de infraestructuras técnicas para abordar la heterogeneidad semántica y lingüística a través de traducción y mapeo de vocabularios.

Ejemplo 22: Integración Inter-Sistemas para la Entrega de Licencias de Negocios
Considere el Ejemplo 20. Suponga que el servicio de licencias empresariales se provee en una región donde los gobiernos entregan servicios transnacionales de manera conjunta. Siendo permitidas las empresas de diferentes países de la región a postularse para este servicio, se les requiere a las empresas registrarse en sus países de origen antes de solicitar el servicio. La Integración Inter-Sistemas involucra la comunicación entre servicios de licencias llevadas a cabo por gobiernos locales y registros de empresas de gobiernos centrales en varios países de la región. En particular, para cada empresa registrada en el exterior que se postule para el servicio, la colaboración es necesaria entre el gobierno local que recibe la solicitud y registro central del país de donde se encuentra registrado el postulante.

2.5.2 Desafíos Tecnológicos

Los desafíos organizacionales presentados en la Sección 2.5.1 requieren una combinación de soluciones organizativas y tecnológicas para superarlos. Lo último, a su vez, da lugar a ciertos desafíos tecnológicos. Basado en los cinco desafíos organizacionales presentados anteriormente, se definen los siguientes desafíos tecnológicos: (T1) proveer acceso único a toda la información y a todos los servicios de gobierno; (T2) coordinar la ejecución de procesos inter-organizacionales; (T3) entregar servicios públicos en conformidad con las políticas y estándares predefinidas por el gobierno. (T4) integrar aplicaciones de software ejecutadas por diferentes organizaciones asociadas; garantizar interoperabilidad sintáctica (T5) y semántica (T6) entre aplicaciones de software ejecutadas por diferentes organizaciones asociadas; (T7) permitir una flexible subcontratación de servicios públicos a organizaciones privadas y no gubernamentales; (T8) sostener un dinámico ecosistema de organizaciones públicas y no públicas para la entrega de servicios públicos; y (T9) facilitar la entrega de servicios públicos usando múltiples canales, tradicionales y electrónicos. Ver Tabla 1.

La Tabla 6 presenta posibles relaciones entre desafíos organizacionales y tecnológicos. Por ejemplo, desafíos organizacionales O1, O2 y O3 están relacionados a los siguientes desafíos tecnológicos: T2 – varias agencias están involucradas en la entrega de servicios y se requiere la coordinación de procesos inter-organizacionales; T4 – se deben integrar todas las aplicaciones de front office, mid office y back office; T5 y T6 – las aplicaciones deben compartir el interpretar correctamente la información. O4 está relacionado con: T3 – socios del sector privado deben entregar servicios públicos según las regulaciones de gobierno; T7 – socios pueden cambiar con el paso del tiempo, pero sin interrumpir el servicio de entrega; y T8 – socios de servicios pueden ser descubiertos a través de páginas blancas y amarillas. Finalmente, la integración inter-sistemas está relacionada a T5 y T6 debido a una heterogeneidad legal, lingüística, técnica y organizativa presente en diferentes administraciones públicas. Además, todos los desafíos organizacionales están relacionados a T10 – las aplicaciones deben ser construidas a partir de componentes confiables.

Tabla 6: Relación entre Desafíos Organizacionales y Tecnológicos

		T1 Acceso Único	T2 Procesos Inter-Organizacionales	T3 Monitoreo Cumplimiento Políticas	T4 Integración de Aplicaciones	T5 Interoperabilidad Sintáctica	T6 Interoperabilidad Semántcia	T7 Subcontratación Flexible	T8 Eco-sistema Dinámico	T9 Entrega por Múltiples Canales	T10 Requerimientos Dependabilidad
O1	De Jerarquías a Redes		■		■	■	■				■
O2	Integración Horizontal		■		■	■	■				■
O3	Integración Vertical		■		■	■	■				■
O4	Integración Intersectorial			■				■	■		■
O5	Integración Inter-Sistemas					■	■				■

A continuación, presentamos las razones de cada desafío tecnológico, algunos problemas técnicos a encarar para implementar una solución al desafío, y un ejemplo:

T1) *Acceso Único* – El Gobierno Integrado asegura que los clientes puedan acceder a información y servicios ofrecidos por el gobierno a través de un único punto de acceso. El principal problema tecnológico para proveer acceso único es cómo recolectar y consolidar información originada en diferentes agencias de gobierno, cómo enviar solicitudes presentadas a través del portal a las aplicaciones de software responsables de procesarlas, y cómo consolidar información de estas aplicaciones de software y ponerla a disponibilidad en el portal de acceso único.

Ejemplo 23: Ofreciendo Licencias de Negocio a través de un Portal de Acceso Único
Considere el Ejemplo 1. Suponga que la aplicación de Front Office (FO) para el servicio de licencias es operada por IACM. Para poder ofrecer este servicio a través de un portal de acceso único, un postulante debería poder presentar su solicitud y los documentos necesarios usando el portal de acceso único del gobierno, que luego debería enviar la solicitud a FO en IACM. Luego, debe permitirle al postulante realizar el seguimiento del estado de su solicitud.

T2) *Procesos Inter-Organizacionales* – Para poder entregar Servicios Integrados, la ejecución colaborativa del proceso de negocios subyacente debe ser posible. Debido a que múltiples agencias están involucradas en la entrega del servicio, la coordinación se lleva a cabo cruzando los límites organizacionales. Con este fin, se necesita una infraestructura técnica para permitir tal coordinación y para mantener información acerca de los pasos de ejecución del proceso: dónde se ejecuta un paso, por quién, cuál es el estado actual de cada instancia del proceso, cuándo avanzar los pasos en cada instancia del proceso, cómo asegurar una vista consistente de todo el proceso por parte de las distintas agencias, cómo responder a expectativas y fracasos, etc.

Ejemplo 24: Ofreciendo Licencias de Negocio a través de Procesos Inter-Organizacionales
Considere el Ejemplo 1. La ejecución del proceso de negocio en la Figura 3 requiere la coordinación de tareas llevadas a cabo por el portal y por las aplicaciones front office y back office en la agencia responsable por la emisión de las licencias (IACM), y las tareas llevadas a cabo por todas las agencias que colaboran en la producción del servicio – DSAL, DSSOPT, CB, IC y SS.

T3) *Monitoreo de Cumplimiento de Políticas* – Para poder entregar servicios públicos integrados colaborativamente y cumplir con los atributos de calidad requeridos, cada socio debe cumplir con las políticas y estándares definidos para la entrega de servicios y para la ejecución del proceso de negocios subyacente. La conformidad con tales políticas y estándares se convierte en crucial cuando organizaciones privadas y no gubernamentales están involucradas. Para monitorear tal conformidad y reaccionar ante los casos descubiertos de no conformidad, se requiere una infraestructura técnica que soporte el monitoreo en tiempo de ejecución de políticas predefinidas y que emita notificaciones cuando se descubre una no conformidad.

Ejemplo 25: Monitoreando el Cumplimiento de Políticas para Licencias de Negocios
Considere el Ejemplo 21, en donde una empresa privada provee el portal de gobierno y mantiene la aplicación de front office para el servicio de licencias provisto a través de ese portal. Una cierta cantidad de políticas relacionadas al servicio se definen para poder controlar la entrega correcta del mismo. Una de ellas especifica que la empresa debe requerir las colaboraciones a DSAL, DSSOPT, CB, IC y SS relacionadas al servicio, dentro de las 24 horas en que la solicitud es presentada a través del portal. La infraestructura de coordinación debería proveer un mecanismo para definir políticas y monitorear su cumplimiento, notificando al administrador de servicio acerca de todos los casos de no conformidad.

T4) *Integración de Aplicaciones* – Un problema de base para la entrega inter-organizativa de servicios públicos es la integración de aplicaciones de software que participan en procesos inter-organizacionales. Dependiendo de tecnologías legadas, tales aplicaciones son difíciles de mantener e integrar. Wrapping – reemplazar una interface entre aplicaciones legadas y los clientes usando tecnologías de fácil integración, es una manera de superar este desafío.

Ejemplo 26: Integrando Aplicaciones para Licencias de Negocios
Considere el Ejemplo 1. Suponga que la aplicación de back office (BO) usada por la agencia IACM fue construida con tecnologías legadas. Envolviendo está aplicación con una interface de fácil integración, por ejemplo usando XML, BO podría ser integrada con el portal de gobierno y la aplicación de front office, y las interacciones con todas las agencias colaboradoras – DSAL, DSSOPT, CB, IC y SS podrían ser automatizadas.

T5) *Interoperabilidad Sintáctica* – Un punto clave para la integración de aplicaciones de software heterogéneas es asegurar que sean capaces de intercambiar información a pesar de las diferencias en los formatos de datos usados. Un paso importante para superar este desafío es la amplia aceptación de XML como el formato común para el intercambio de datos.

Ejemplo 27: Interoperabilidad Sintáctica para el Servicio de Licencias de Negocios
Considere el Ejemplo 1. Luego de que todas las agencias participantes lleguen a un acuerdo por el uso de XML para el intercambio de mensajes, la aplicación de BO en IACM debería poder intercambiar mensajes con las aplicaciones en DSAL, DSSOPT, CB, IC y SS.

T6) *Interoperabilidad Semántica* – Mientras que las aplicaciones de software pueden ser capaces técnicamente de intercambiar mensajes, puede que no interpreten tales mensajes de la misma manera. Interoperabilidad Semántica garantiza que las aplicaciones comparten un entendimiento común de los conceptos intercambiados. Tecnologías de la Web Semántica proveen las herramientas para llevar a cabo esto.

Ejemplo 28: Interoperabilidad Semántica para el Servicio de Licencias de Negocios
Considere el Ejemplo 22. Para poder confirmar que una empresa postulada para la licencia está registrada en su país de origen, el servicio de licencias debería poder consultar los registros centrales de empresas en los diferentes países que participan del servicio. Como los registros pueden estructurar su información de diferentes maneras, y usar diferentes lenguajes y diferentes términos legales y técnicos, la interoperabilidad semántica es necesaria en este caso.

T7) *Subcontratación Flexible* – Involucrar al sector privado en la entrega de servicios públicos introduce simultáneamente eficiencia y riesgo en la entrega del servicio. Acuerdos deberían hacerse tanto a nivel técnico como regulatorios para asegurar que los roles en la entrega del servicio son bien asignados y que estos roles, lo mismo que las organizaciones que los cumplen, pueden cambiar en el tiempo sin causar interrupciones importantes en la entrega misma.

Ejemplo 29: Subcontratación Flexible para el Servicio de Licencias de Negocios
Considere el Ejemplo 21. Suponga que la aplicación de front office del servicio de licencias va a ser reemplazada por una nueva aplicación de software desarrollada por IACM. El cambio en el proveedor de servicio y el uso de una nueva interface (API) provista por la aplicación de front office por parte del portal no debería ser percibido, o al menos no debería ser un problema para los postulantes.

T8) *Ecosistema Dinámico* – Marcos regulatorios y técnicos deberían existir para permitir y mantener un ecosistema dinámico de organizaciones públicas y privadas que puedan trabajar conjuntamente en la entrega de servicios públicos. Los marcos deben ser capaces de resolver cambios constantes en el ecosistema – número de organizaciones, nuevos servicios, cambios de políticas, etc., al mismo tiempo, deben garantizar la entrega estable de servicios públicos. Además, debería ser posible monitorear tales ecosistemas para ver qué tipo de organizaciones – agencias, empresas privadas, ONGs, etc. están colaborando con la entrega de servicios, qué servicios proveen ellos, qué tipo de colaboraciones se llevan a cabo, cómo las asociaciones cambian, etc.

Ejemplo 30: Entregando Servicios de Licencias de Negocios a través de un Ecosistema
Considere el Ejemplo 21. El administrador del servicio de licencias que trabaja en IACM debería poder saber cuántas entidades hay actualmente presentes en el ecosistema, incluyendo empresas y organizaciones no gubernamentales que colaboran con la entrega del servicio de licencias, y cómo cambian con el paso del tiempo.

T9) *Entrega por Múltiples Canales* – Para maximizar el acceso, los servicios públicos deberían ser entregados a través de una variedad de canales tradicionales y electrónicos, como se explica en la Sección 2.2.3. Con nuevos métodos

de servicios de entrega y nuevos medios de entrega que emergen cada año, la automatización de la distribución de contenido a lo largo de diferentes canales es esencial, al igual que las estrategias explícitas para administrar múltiples canales, apoyado por marcos técnicos y regulatorios apropiados.

Ejemplo 31: Entregando Licencias de Negocios a través de Múltiples Canales
Considere el Ejemplo 1: Suponga que un postulante debe registrarse en el portal de gobierno antes de poder enviar la solicitud a través él. Durante la registración, él o ella pueden elegir el canal de preferencia para recibir notificaciones. Luego de solicitar el servicio de licencia, el postulante recibirá notificaciones a través del canal seleccionado.

T10) *Requerimientos de Dependabilidad* – El principal atributo de calidad para cualquier sistema de soporte de Gobierno Integrado es la dependabilidad – el sistema lleva a cabo sus funciones apropiadamente y entrega a los usuarios los resultados esperados. En particular, los siguientes siete atributos de dependabilidad son todos relevantes y son lo que se espera de cualquier solución técnica [SDC06]:

1) *Disponibilidad* – Una agencia siempre debe poder requerir la colaboración de sus socios.
2) *Confiabilidad* – Las comunicaciones entre socios colaboradores deben ser confiables, con la garantía de completitud e integridad para la información intercambiada por parte de la infraestructura subyacente.
3) *Seguridad* – Las colaboraciones solamente pueden ser solicitadas por y provistas a socios autorizados y autenticados, con la garantía de confiabilidad e integridad de la información intercambiada.
4) *Puntualidad* – Las colaboraciones solicitadas deben ser provistas a tiempo.
5) *Supervivencia* – La colaboración debería ser posible, aún con descensos en la performance, ante fallas accidentales o ataques intencionados a la infraestructura de TIC subyacente.
6) *Recuperabilidad* – La colaboración deberá ser mantenida luego de detectar errores de proceso por parte de los socios y se deberá asegurar la posterior reparación de sus efectos.
7) *Mantenibilidad* – Incluir nuevas características en la infraestructura TIC subyacente o la reparación de características existentes deberá consumir una cantidad acotada de recursos.

Ejemplo 32: Requerimientos de Dependabilidad para la Entrega de Licencias de Negocios
Considere el Ejemplo 1. Los siguientes escenarios ilustran como los atributos de dependabilidad se refieren a la entrega del servicios de licencia: (1) Disponibilidad – el portal es capaz de aceptar y enviar solicitudes presentadas a la Autoridad Otorgante de Licencias de IACM en todo momento; (2) Confiabilidad – el intercambio de información entre las agencias que colaboran con la producción del servicio se lleva a cabo de manera confiable; (3) Seguridad – la información intercambiada entre IACM y DSAL acerca de la seguridad en el ambiente de trabajo se mantiene confidencial; (4) Puntualidad – pedidos de colaboración emitidos por IACM a todos los socios del servicios son entregados a tiempo; además, IACM recibe las respuestas a tiempo; (5) Supervivencia – las agencias son capaces de colaborar a pesar de problemas técnicos limitados que afectan la infraestructura TIC subyacente; (6) Recuperabilidad – la colaboración entre agencias deberá recuperarse luego de errores pasajeros, por ejemplo, luego de enviar información inválida a la aplicación de front office de IACM; y (7) Mantenibilidad – la búsqueda de una opinión adicional para la emisión de licencias no requiere una reingeniería del sistema en general y evita causar interrupciones a los usuarios.

Capítulo 3

Trabajos Relacionados

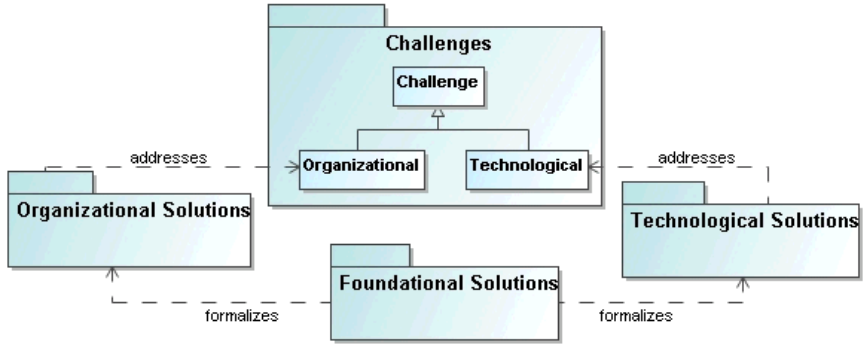
Siguiendo la introducción de los conceptos que definen el dominio de Gobierno Integrado, en el Capítulo 2 se presentaron varios desafíos organizacionales y técnicos para desarrollar Gobierno Integrado, considerando en particular, los relacionados con la colaboración entre agencias. El Capítulo 3 presenta una revisión de las herramientas, teorías, frameworks y software, que pueden dar solución, o parte de la misma, al desafío mencionado. Clasificamos dichas soluciones como organizacionales, tecnológicas y fundacionales. Una solución organizacional es un instrumento – framework, política, estándar, herramienta, etc., que facilita la colaboración en el gobierno, particularmente en el intercambio de información. Una solución tecnológica es cualquier artefacto relacionado al software – aplicaciones desarrolladas por el cliente o compradas, arquitecturas de software, etc., que integran aplicaciones ejecutadas por diferentes socios colaboradores. Una solución fundacional es cualquier herramienta formal – teoría, lenguaje, método, etc., capaz de soportar soluciones tecnológicas u organizacionales. También se propone un marco de evaluación para comparar diferentes tipos de soluciones y aplicarlo a las soluciones descriptas en este capítulo.

La estructura del capítulo es la siguiente: la Sección 3.1 provee una visión general de las soluciones existentes e introduce un marco de evaluación para compararlas. La Sección 3.2 presenta seis soluciones organizacionales concretas incluyendo marcos de interoperatibilidad y arquitecturas empresariales. La Sección 3.3 presenta siete soluciones tecnológicas concretas: soluciones de propósito general, soluciones específicas para Gobierno Electrónico, arquitecturas de software y frameworks de desarrollo. La Sección 3.4 introduce cuatro lenguajes para soportar rigurosamente el desarrollo de soluciones organizacionales y tecnológicas. Cada solución presentada en las secciones 3.2, 3.3 y 3.4 es evaluada usando el marco introducido en la Sección 3.1. Los resultados de la evaluación, incluyendo comparaciones, son presentados en la Sección 3.5.

3.1 Marco de Evaluación

Como se mencionó anteriormente, el desarrollo de Gobierno Integrado debe lidiar con desafíos legales, financieros, sociales, organizacionales y tecnológicos. Existen varias soluciones para tratar los desafíos organizacionales y tecnológicos. Este capítulo presenta una lista parcial de dichas soluciones, como también unas pocas soluciones fundacionales – lenguajes y métodos, que pueden ser usadas. En la Figura 21 se describen las soluciones organizacionales tratando los desafíos organizacionales, las soluciones tecnológicas tratando los desafíos técnicos y las soluciones fundacionales sustentando el desarrollo de las soluciones organizacionales y tecnológicas.

Figura 21: Trabajos Relacionados – Soluciones Existentes



En el resto de la sección se definirá un marco de evaluación para comparar y analizar los diferentes tipos de soluciones, en dos partes: (1) características generales – atributos comunes a todos los tipos de soluciones, y (2) características específicas – atributos que aplican específicamente a una solución organizacional, técnica o fundacional.

Las características generales comprenden cuatro atributos: (1) nombre, (2) tipo, (3) proveedor y (4) elementos de la solución. Un proveedor es una compañía, gobierno, institución o persona responsable de proveer o mantener la solución. Las características específicas se definen separadamente para las soluciones organizacionales, tecnológicas y fundacionales:

- Las soluciones organizacionales se evalúan usando tres atributos específicos:

  - *Propósito* – El uso planeado de la solución.
  - *Vistas* – Diferentes visiones usadas por la solución para modelar elementos del dominio: (a) Vista de la Información – herramientas para modelar datos y meta-datos usados por procesos y aplicaciones; (b) Vista del Proceso – herramientas para modelar procesos de negocio ejecutados por una organización; (c) Vista de la Tecnología – herramientas para modelar recursos relacionados a la tecnología informática como hardware, software, redes, dispositivos etc., usados por una organización; y (d) Vista de la Organización – herramientas para modelar estructuras organizacionales, recursos humanos, roles, asociaciones y otras.
  - *Interoperabilidad* – En qué medida la solución soporta tres tipos de interoperabilidad: (a) Técnica, (b) Semántica y (c) Organizacional.

- Las soluciones tecnológicas se evalúan usando tres atributos específicos:

  - *Arquitectura* – Patrones arquitectónicos – los componentes principales que conforman la solución y las interacciones entre ellos.
  - *Atributos No Funcionales* – Evaluación de atributos de calidad del software, seleccionados en términos de: (a) confiabilidad, (b) portabilidad, y flexibilidad, que se refina en: (c) mensajes auto-descriptos, (d) paradigmas de comunicación, (e) mecanismos de distribución y (f) limitaciones. En total, los siguientes seis atributos de calidad son evaluados.

    1) *Confiabilidad* – Habilidad de la solución para enviar mensajes en circunstancias normales o anormales, garantizando el envío de mensajes a pesar de la no disponibilidad ocasional de los receptores.
    2) *Portabilidad* – Habilidad para implementar la solución en múltiples plataformas tecnológicas;
    3) *Mensajes Auto-descriptos* – Flexibilidad de la solución para definir el contenido de los mensajes, por ejemplo si la solución aplica un formato propietario o a un formato abierto para auto-definir mensajes.
    4) *Paradigma de Comunicación* – Soporte para comunicaciones sincrónicas y asincrónicas;
    5) *Mecanismo de Distribución* – Soporte para diferentes mecanismos de envío: punto-a-punto – envío de mensajes a un receptor, o publicar-suscribir – envío de mensajes a diferentes receptores;
    6) *Limitaciones* – Restricciones en el uso de la solución, por ejemplo el tamaño máximo de mensaje o el tamaño de las colas.

  - *Atributos Funcionales* – Evaluación de funcionalidad adicional provista por la solución, tales como:

    a) *Servicio de Recupero* – Recepción de los mensajes en un orden determinado por el receptor;
    b) *Servicio de Auditoría* – Almacenamiento de los mensajes en tránsito para una acceso futuro;
    c) *Servicio de Validación* – Verificación de la conformidad de los mensajes con una sintaxis predefinida;
    d) *Servicio de Transformación* – Modificación de los mensajes aplicando un conjunto de reglas predefinidas;
    e) *Servicio de Autenticación* – Permiso para sólo por participantes autorizados puedan comunicarse;
    f) *Servicio de Nombrado* – Asignación de nombres únicos a participantes de la comunicación y estructuras;
    g) *Servicio de Ubicación* – Búsqueda y ubicación de participantes y estructuras de comunicación;
    h) *Servicio de Transacción* – Soporte para asegurar la finalización satisfactoria de transacciones;
    i) *Servicio de Composición* – Soporte para la composición de servicios; y
    j) *Servicio de Desarrollo* – Soporte para el desarrollo y la distribución de nuevos servicios.

- Las soluciones fundacionales son evaluadas usando cuatro atributos:

  - *Estilo de Especificación* – El estilo de descripción del comportamiento aplicado por las soluciones: basado en estado y basado en acciones [AL93]. En el primero, el comportamiento de un sistema es modelado como una secuencia de estados; un estado representa una asignación de valores a variables. En el segundo, el comportamiento es modelado como una secuencia de acciones; una acción representa un evento que dispara un cambio de estado. Aunque ambos enfoques están relacionados, difieren en fundamentos y aplicabilidad. Las especificaciones basadas en estados están basadas en lógica – la especificación es expresada como una fórmula lógica; y son apropiadas para sistemas de uso intensivo de datos. Las especificaciones basadas en acciones están basadas en álgebra – la especificación es un objeto que puede ser procesado algebraicamente [AL93]; y son ampliamente usadas para especificar sistemas distribuidos y concurrentes.
  - *Semántica* – El enfoque aplicado para definir la semántica del lenguaje: operacional – el significado es expresado en términos de cambios de estado causados por ejecución de acciones; denotacional – el significado es expresado en términos de objetos matemáticos que representan el efecto de computaciones; y axiomático – el significado es definido a través de afirmaciones que especifican el efecto de computaciones dado el cumplimiento de ciertas propiedades [NN99].
  - *Refinamiento* – El enfoque de desarrollo aplicado para derivar una especificación abstracta en una más concreta, o refinar una especificación en una implementación. Hay dos tipos usuales de refinamiento aplicado a soluciones fundacionales: modelos de sub-clase – el conjunto de modelos que satisface una especificación abstracta es un súper conjunto del modelo de conjuntos que satisface una especificación concreta, y preservación de propiedades – todas las propiedades de una especificación abstracta se mantienen como propiedades válidas de la especificación concreta [DGJM02].
  - *Herramientas* – Las herramientas provistas por la solución para soportar el desarrollo de procesos. Por ejemplo, herramientas para verificar la consistencia de especificaciones, para animar especificaciones, para probar propiedades de una especificación, para refinar y documentar una especificación o generar código.

Las tres secciones siguientes analizan soluciones organizacionales, tecnológicas y fundacionales respectivamente; mientras que la última sección compara las soluciones presentadas en cada categoría.

3.2 Soluciones Organizacionales

Siguiendo a la Sección 2.3.3 en la que se introducían Marcos de Interoperabilidad y Arquitecturas Empresariales como dos enfoques principales a los esfuerzos de integración en gobierno, esta sección presenta tres ejemplos de Marcos de Interoperabilidad – Marco de Interoperabilidad de Gobierno Electrónico (e-GIF) (Sección 3.2.1), Marco de Interoperabilidad de Gobierno Electrónico de Nueva Zelanda (NZ e-GIF) (Sección 3.2.2), y Marco Europeo de Interoperabilidad (EIF) (Sección 3.2.3), y tres ejemplos de Arquitecturas Empresariales – de Zachman (Sección 3.2.4), Estándares y Arquitecturas para Aplicaciones de Gobierno Electrónico (SAGA) (Sección 3.2.5), y Arquitectura Empresarial Federal (Sección 3.2.6). Las soluciones organizacionales están descriptas en la Figura 22. Para cada solución se aplica el marco de evaluación definido en la Sección 3.1.

Figura 22: Trabajos Relacionados – Soluciones Organizacionales

3.2.1 Marco de Interoperabilidad de Gobierno Electrónico (e-GIF)

El Marco de Interoperabilidad de Gobierno Electrónico (e-GIF) [UK07] fue publicado por la Unidad de Gobierno Electrónico de la Oficina de Gabinete del Reino Unido. e-GIF comprende dos componentes principales: (1) el marco de e-GIF – un documento que cubre declaraciones de políticas de alto nivel y regímenes para manejar e implementar políticas y para asegurar el cumplimento de las mismas; y (2) el registro de e-GIF – un repositorio que comprende Estándares de Metadatos de Gobierno Electrónico (e-GMS) , Lista de Categorías de Gobierno (GCL), Catálogo de Estándares de Datos de Gobierno (GDSC), Esquemas XML, y Catálogo de Estándares Técnicos (TSC). Más detalles sobre e-GIF fueron provistos en la Sección 2.3.3.

e-GIF tiene las siguientes características específicas:

- *Propósito* – Habilitar flujo continuo de información a través de organizaciones públicas del Reino Unido.
- *Vistas* – e-GIF provee elementos de modelado para tres vistas – de la Información, del Proceso y de la Tecnología. Por ejemplo, soporta la Vista de la Información a través de Estándares de Metadatos de Gobierno Electrónico (e-GMS), Lista de Categoría del Gobierno (GCL), Catálogo de Estándares de Datos de Gobierno (GDSC), y Esquemas XML, los que son usados para definir conceptos, modelos y datos. Por ejemplo, e-GMS provee elementos y esquemas para crear y refinar metadatos para recursos de información del gobierno. El Catálogo de Estándares Técnicos (TSC) soporta la Vista del Proceso a través de estándares para varias áreas verticales tales como reportes financieros, órdenes de compra, sistemas de facturación, y otras. La Vista de la Tecnología es soportada por los estándares de Tecnología Informática incluidos en el TSC, como especificaciones para estaciones de trabajo, teléfonos celulares, tarjetas inteligentes, etc. La Vista de la Organización no es cubierta por e-GIF.
- *Interoperabilidad* – e-GIF soporta interoperabilidad técnica a través de políticas y especificaciones que gobiernan el flujo de información a través del gobierno y el sector público en general, tales como especificaciones para inter-conectividad, servicios web, integración de datos, manejo de contenido de metadatos e identificadores. Soporta parcialmente la interoperabilidad semántica. Por ejemplo, e-GMS puede especificar de qué manera las agencias del sector público en el Reino Unido deberían etiquetar el contenido de las páginas web y documentos, para facilitar que dichos contenidos sean manejables, ubicables y compartibles. También provee guías para la especificación de semántica del intercambio electrónico de datos y servicios de mensajería. e-GIF también soporta interoperabilidad organizacional. Por ejemplo, provee especificaciones para determinadas áreas de negocio incluyendo e-Aprendizaje, e-Salud y servicios sociales, finanzas y comercios, compras y logística. También provee guías para la distribución de servicios públicos a través de múltiples canales.

3.2.2 Marco de Interoperabilidad de Gobierno Electrónico de Nueva Zelanda (NZ e-GIF)

El Marco de Interoperabilidad de Gobierno Electrónico de Nueva Zelanda (NZ e-GIF) [NZ08a] fue desarrollado por la Comisión de Servicios del Estado del Gobierno de Nueva Zelanda. NZ e-GIF consiste de tres documentos: (1) Estándares – la estructura del marco y los estándares definidos, (2) Políticas – las políticas para desarrollar y administrar el marco, y (3) Recursos – la historia, referencias y motivaciones de NZ e-GIF, y la lista de URLs de todos los estándares usados por el marco. Los estándares están estructurados en tres capas: (1) Red – estándares para transporte de datos; (2) Integración de Datos – estándares para habilitar el intercambio e interpretación de datos entre sistemas heterogéneos, (3) Servicios de Negocio – estándares para especificar el mapeo de datos elementales a información de negocios útil, y (4) Presentación y Acceso – estándares y guías para el acceso a sistemas de negocios. Cuatro componentes verticales se aplican a todas las capas: (1) Seguridad – estándares de seguridad aplicables a varias capas de un sistema y etapas en su desarrollo; (2) Mejores Prácticas – estándares para comprender y administrar el contexto para intercambio de información; (3) Servicios de Gobierno Electrónico – componentes de infraestructura provistos por oficinas coordinadoras para ser usados a lo largo de todo el gobierno; y (4) Servicios Web – componentes estandarizados y métodos para integrar aplicaciones basadas en la web. Finalmente, incluyendo todas las capas, los componentes de Gerenciamiento y Gobernabilidad son usados para administrar el marco.

NZ e-GIF tiene las siguientes características específicas:

- *Propósito* – Proveer una infraestructura tecnológica basada en un modelo de capas, para clasificar estándares de tecnología informática.

- *Vistas* – NZ e-GIF provee elementos de modelado para las cuatro vistas. Soporta parcialmente la Vista de la Información a través de estándares presentes en la capa de Integración de Datos, como estándares para conjuntos de caracteres como ASCII [CU81] o UTF-8 [IETF98], o estándares para datos estructurados como XML [W3C08] o DTD [W3C99c]. Adicionalmente, la capa de Servicios de Negocio soporta la Vista de la Información a través de estándares para descubrimiento de metadatos, como el Localizador de Datos del Gobierno de Nueva Zelanda [NZ08b], el Tesauro del Localizador de Datos del Gobierno de Nueva Zelanda [NZ08c] o RDF [W3C04e], y estándares para espacios de nombres y direcciones. La Vista del Proceso está soportada a través de elementos contenidos en la capa de Acceso y Presentación, como recomendaciones para diseño de sitios web, presentación y mantenimiento, estándares para autenticación, y estándares incluidos en los componentes verticales como los Servicios de Gobierno Electrónico o Servicios Web. La Vista de la Tecnología es soportada a través de estándares definidos en la capa de Red, como estándares para transporte de datos y transferencia de archivos, o estándares incluidos en el componente de Seguridad, por ejemplo integración de datos o infraestructura de clave pública. La Vista de la Organización es soportada por elementos del componente de Gerenciamiento y Gobernabilidad incluidos en el documento de políticas del marco.
- *Interoperabilidad* – NZ e-GIF provee soporte para interoperabilidad técnica, por ejemplo a través de estándares provistos en la capa de Red e Integración de Datos y aquellos en el componente de Seguridad. Provee soporte parcial para interoperabilidad semántica, por ejemplo la capa de Servicios de Negocio especifica estándares para datos estructurados, metadatos y modelo de datos. Adicionalmente, NZ e-GIF soporta intercambio de datos en dominios específicos tales como manejo de relaciones de clientes y reportes de negocio. Sin embargo, no da soporte para interoperabilidad organizacional.

3.2.3 Marco Europeo de Interoperabilidad (EIF)

El Marco Europeo de Interoperabilidad (EIF) [IDA04] fue publicado por IDABC – Interoperable Delivery of European e-Government Services to Public Administrations, Businesses and Citizens. EIF consiste de: (1) recomendaciones para los servicios de Gobierno Electrónico – guías para el desarrollo de los servicios de Gobierno Electrónico, con foco en las interacciones a través de las fronteras nacionales de Europa; (2) interacciones pan-europeas – entre clientes de gobierno, administraciones públicas de los estados miembros de la Unión Europea (UE), e instituciones de la UE; y (3) recomendaciones de interoperabilidad – recomendaciones para el tratamiento de interoperabilidad técnica, semántica y organizacional.

EIF tiene las siguientes características específicas:

- *Propósito* – Complementar, no reemplazar, las guías de interoperabilidad nacional agregando la dimensión pan-europea.
- *Vistas* – EIF provee elementos de modelado para las cuatro vistas. La Vista de la Información es soportada por medio de acuerdos sobre diccionarios de datos, tesauros multilenguaje, y vocabularios de XML relacionados con servicios pan-europeos. La Vista del Proceso es soportada formalizando tres tipos de interacciones: (1) entre clientes de un estado miembro y las administraciones de otros estados miembros o instituciones europeas; (2) entre administraciones de diferentes estados miembros; y (3) entre instituciones europeas de un estado miembro y administraciones de otros estados miembros. La Vista de la Tecnología es soportada a través de estándares relacionados con la tecnología informática incluidos en el marco, por ejemplo estándares sobre transporte de mensajes, seguridad y servicios de red. Finalmente, la Vista de la Organización es parcialmente soportada por recomendaciones organizacionales, tales como la formalización de las expectativas de las varias administraciones públicas que contribuyen a la provisión de servicios pan-europeos de Gobierno Electrónico.
- *Interoperabilidad* – EIF soporta la interoperabilidad técnica. Por ejemplo, a través de recomendaciones técnicas para el intercambio de datos entre varias agencias de la UE, estándares para aplicaciones de front-office en relación a la presentación e intercambio de datos, accesos a través de múltiples canales, etc. El marco identifica cuestiones serias de interoperabilidad semántica y provee guías para tratarlas a nivel pan-europeo. Por ejemplo, se cubren aéreas tales como acuerdos en diccionarios de datos relacionados con servicios pan-europeos y la adopción de semánticas comunes en base a vocabularios XML. Adicionalmente, algunos documentos de soporte de EIF, tales como Estrategia de Interoperabilidad Semántica, describen activos semánticos – diccionarios, tesauros multilenguaje, tablas de referencias cruzadas y mapeo, ontologías y servicios, y proveen guías y estrategias para planificar e implementar interoperabilidad semántica. Finalmente, EIF soporta interoperabilidad

organizacional, por ejemplo ofreciendo recomendaciones organizacionales para llevar a cabo los tres tipos de interacciones para proveer servicios pan-europeos como se describió anteriormente.

3.2.4 Arquitectura Empresarial de Zachman

La Arquitectura Empresarial de Zachman [Zac87], como su nombre lo indica es una arquitectura empresarial desarrollada por J.A. Zachman. La arquitectura está estructurada como una matriz de cinco filas y seis columnas. Las filas representan los roles de diferentes grupos de interesados involucrados en la definición de la arquitectura – Planificador, Propietario, Diseñador, Constructor y Subcontratista, con las siguientes áreas de interés: el Planificador está interesado en el alcance de la arquitectura, el Propietario en el modelo de negocio, el Diseñador en el modelo del sistema, el Constructor en el modelo tecnológico y el Subcontratista en el modelo detallado. Las columnas describen las preguntas que se supone cada descripción de la arquitectura debe contestar – Qué, Cómo, Dónde, Quién, Cuándo y Por Qué, contestadas utilizando las siguientes descripciones: Qué - Datos, Cómo - Función, Dónde - Red, Quién – Gente, Cuándo - Tiempo y Por Qué – Motivación. Inicialmente, Zachman pensaba que cada celda de la matriz es independiente de las otras, por ejemplo los datos siendo independientes de las funciones o las ubicaciones de red, de este modo, brindando un modelo de mayor flexibilidad y claridad. Sin embargo, él reconoció más tarde que cada celda está relacionada con otras celdas en la misma fila. Otros autores también señalaron la existencia de relaciones entre celdas ubicadas en diferentes filas [MPS04]. La arquitectura propuesta por Zachman fue utilizada como fundamento para el análisis y desarrollo de varias otras arquitecturas empresariales.

La Arquitectura Empresarial de Zachman tiene las siguientes características específicas:

- *Propósito* – Proveer una estructura lógica para clasificar y organizar representaciones descriptivas de una empresa.
- *Vistas* – La arquitectura provee elementos de modelado para las cuatro vistas. Soporta la Vista de la Información a través de descripciones incluidas en la Perspectiva de Datos – entidades de negocio, entidades de datos y definiciones de datos. Soporta la Vista del Proceso usando descripciones incluidas en tres perspectivas: Perspectiva de Función – procesos de negocio, arquitecturas de aplicación y modelos de diseño de sistemas; Perspectiva del Tiempo – calendarios y modelos de control; y Perspectiva de la Motivación – planes de negocio y modelos de reglas de negocio. Soporta la Vista de la Tecnología usando descripciones incluidas en la Perspectiva Dónde – sistemas de logística de negocios, arquitecturas de sistemas distribuidos y arquitecturas tecnológicas. Finalmente, soporta la Vista de la Organización usando descripciones incluidas en la Perspectiva de la Gente – modelos de flujo de trabajo y arquitecturas de interface humana.
- *Interoperabilidad* – La arquitectura de Zachman soporta parcialmente la interoperabilidad técnica usando plantillas para agregar aspectos técnicos de la arquitectura. Soporta interoperabilidad semántica a través de modelos semánticos incluidos en la Perspectiva de Datos, descripción de procesos como parte de la Perspectiva del Propietario, y descripciones de los participantes como parte de la Perspectiva de la Gente. En cuanto a la interoperabilidad organizacional, la arquitectura no define el propósito de las interacciones entre los actores, pero permite la especificación de los participantes y los procesos internos del negocio. Algunas características están parcialmente soportadas, como la posibilidad de definir miembros externos relacionados a una organización, pero sin capturar su rol. No se consideran otras características, como interacciones con otras organizaciones.

3.2.5 Estándares y Arquitecturas para Aplicaciones de Gobierno Electrónico (SAGA)

Los Estándares y Arquitecturas para Aplicaciones de Gobierno Electrónico (SAGA) [KBST06] es una arquitectura empresarial definida por la Junta Coordinadora y Asesora para Tecnología Informática en la Administración (KBSt unit) del Ministerio Federal del Interior, Alemania, en cooperación con AG y Fraunhofer-Institut für Software- und Systemtechnik (IIST). SAGA describe aplicaciones distribuidas de Gobierno Electrónico basadas en el Modelo de Referencia de Procesamiento Distribuidos Abierto (RM-ODP) [Ray93]. SAGA modela las aplicaciones de Gobierno Electrónico usando cinco puntos de vista diferentes:

1) *Punto de Vista de la Empresa* – Específica el propósito, alcance, procesos y políticas adoptadas por una aplicación, junto con todo el entorno y el propósito de la aplicación dentro de la organización. Incluye las definiciones de procedimientos, reglas, actores y sus roles, involucrados en procesos soportados por aplicaciones.

2) *Punto de Vista de la Información* – Trata la estandarización y el modelado de datos – especifica la estructura y semántica de la información y actividades que pueden realizarse sobre objetos de información y restricciones que se aplican.
3) *Punto de Vista de la Computación* – Describe la estructura de las aplicaciones distribuidas de Gobierno Electrónico usando una arquitectura de tres capas para los servicios y una de cuatro capas para los sistemas. Promueve el uso de Arquitectura Orientada a Servicios, y refiere a Seguridad Integrada en las Aplicaciones de Gobierno Electrónico como una solución de seguridad estándar. También fomenta el reuso y la integración a través del framework One-For-All (OFA), el cual provee bloques de funcionalidad a ser usados como parte de los servicios, sistemas y componentes de infraestructura en las aplicaciones de Gobierno Electrónico.
4) *Punto de Vista de la Ingeniería* – Describe el soporte de sistema requerido para objetos distribuidos, incluyendo las unidades en donde los objetos son ejecutados, tales como hardware y comunicaciones, y toda clase de plataformas para sistemas distribuidos.
5) *Punto de Vista de la Tecnología* – Define estándares para arquitectura de TI y seguridad de datos, y otras tecnologías concretas seleccionadas para implementar sistemas.

Más detalles acerca de SAGA se presentaron en la sección 2.3.3. Las características específicas de SAGA son:

- *Propósito* – Proveer un conjunto de estándares para soportar el desarrollo de Gobierno Electrónico en Alemania en términos de: interoperabilidad, reusabilidad, estándares abiertos, reducción de costos y riesgos y escalabilidad.
- *Vistas* – SAGA provee elementos de modelado para las cuatro vistas. Soporta: Vista de la Información a través del Punto de Vista de la Información; Vista del Proceso a través del Punto de Vista Computacional; Vista de la Tecnología a través de los Puntos de Vista de la Ingeniería y de la Tecnología; y Vista de la Organización a través del Punto de Vista de la Empresa.
- *Interoperabilidad* – SAGA soporta interoperabilidad técnica a través de los Puntos de Vista de la Tecnología y de la Ingeniería, por ejemplo proponiendo XML Schema para construir descripciones estructuradas de datos o UML para modelar datos en aplicaciones orientadas a objeto. Soporta interoperabilidad semántica a través del Punto de Vista de la Información, por ejemplo proponiendo guías para estandarizar modelos de datos y asignar semántica a los archivos XML y documentos intercambiados entre agencias. Soporta interoperabilidad organizacional proponiendo guías para el tratamiento de asuntos centrales de la organización (procesos y su gobernanza) dentro del Estado Federal Alemán; para estandarizar enfoques de distribución de servicios optimización de procesos, entrenamiento de personal y participación de usuarios en todo el gobierno; y para proveer guías acerca de asuntos legales relativos al uso de firmas electrónicas, protección de datos y accesibilidad. La mayoría de los aspectos organizacionales son descriptos en el Punto de Vista de la Empresa.

3.2.6 Arquitectura Empresarial Federal (FEA)

La Arquitectura Empresarial Federal (FEA) [FEA07] es una arquitectura empresarial definida por la Oficina de Administración del Programa FEA de la Oficina de Administración y Presupuesto (OMB), Estados Unidos de América. FEA comprende cinco modelos interrelacionados. Todos ellos usan jerarquías para desarrollar y refinar categorías de clasificación para sus dominios respectivos. Dichas categorías de clasificación les permite a las agencias de gobierno identificar funciones de negocio, relacionar mediciones de performance y componentes de servicio con funciones, y relacionar estándares y especificaciones con los componentes de servicio necesarios para soportar las funciones de negocio. Los cinco modelos de FEA incluyen:

1) *Modelo de Referencia del Negocio* (BRM) – Una perspectiva funcional de operaciones del gobierno para permitir la identificación y comparación de funciones a través de agencias, define una jerarquía de tres niveles de categorías: áreas de negocio (nivel superior), líneas de negocio dentro de áreas de negocio (nivel medio) y sub funciones para líneas de negocio (nivel inferior).
2) *Modelo de Referencia de Performance* (PRM) – Una medición de performance de iniciativas de tecnologías de la información, define una jerarquía de cuatro niveles: Áreas de Medición, Categorías de Medición, Agrupamientos de Medición, e Indicadores de Medición.
3) *Modelo de Referencia de los Componentes de Servicio* (SRM) – Una clasificación funcional basada en negocios, de los componentes de servicio con respecto al soporte que proveen para los negocios de gobierno y para el cumplimiento de los objetivos de performance, define una jerarquía de tres niveles para categorizar: Dominios de Servicios, Tipos de Servicios y Componentes.

4) *Modelo de Referencia de Datos* (DRM) [FEA05] – Un marco basado en estándares para poder compartir y re-usar la información a través del gobierno federal; comprende descripciones estándares y procesos de localización de datos comunes, y promueve prácticas uniformes para administración de datos. DRM define tres áreas de estandarización a través de las cuales los datos pueden ser descriptos, categorizados y compartidos – Descripción de Datos, Contexto de Datos, y Áreas de Datos Compartidos.
5) *Modelo de Referencia Técnica* (TRM) – Una clasificación técnica basada en componentes, usada para identificar estándares, especificaciones y tecnologías que soportan y permiten la entrega de componentes de servicio y capacidades. TRM está construida sobre un esquema de categorización de tres niveles: Áreas de Servicio, Categorías de Servicio y Estándares de Servicio.

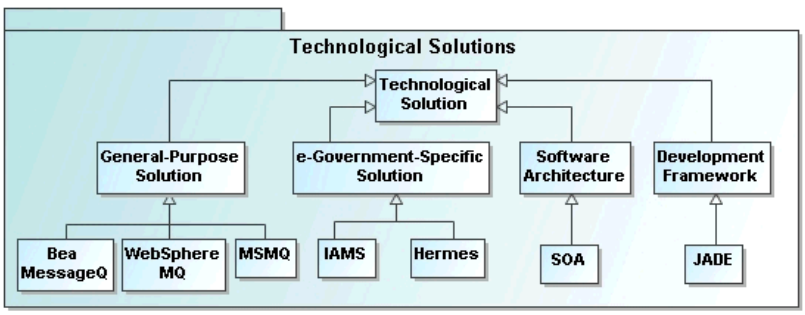
FEA tiene las siguientes características particulares:

- *Propósito* – Asistir en el desarrollo y mantenimiento de arquitecturas empresariales inter-agencias.
- *Vistas* – Provee elementos de modelado para las cuatro vistas. En base a la descripción de FEA provista anteriormente, la Vista de la Información es soportada por el Modelo de Referencia de Datos, la Vista del Proceso es soportada en forma conjunta por el Modelo de Referencia del Negocio, el Modelo de Referencia de los Componentes de Servicio y el Modelo de Referencia de Performance; mientras que la Vista de la Tecnología es soportada por el Modelo de Referencia Técnica, y la Vista de la Organización es soportada por el Modelo de Referencia del Negocio.
- *Interoperabilidad* – Soporta interoperabilidad técnica a través del Modelo de Referencia Técnica, por ejemplo proponiendo estándares para manejar sesiones de comunicación punto-a-punto, incluyendo protocolos de acceso y entrega. Soporta interoperabilidad semántica aplicando esquemas de clasificación estándar y taxonomías en todos sus modelos de referencia. Por ejemplo, el Modelo de Referencia de Datos soporta semántica para datos compartidos en base a tres áreas de estandarización – descripción de datos, contextos y datos compartidos. Adicionalmente, una ontología concreta – la Ontología del Modelo de Referencia de FEA, ha sido desarrollada para soportar el intercambio efectivo de datos entre agencias. Un soporte parcial a la interoperabilidad organizacional es provisto por el Modelo de Referencia del Negocio y el Modelo de Referencia de los Componentes de Servicio. El primero permite una vista funcional e inter-agencias de las líneas de negocio del gobierno federal, proveyendo taxonomías de funciones de negocio. El segundo identifica servicios horizontales asociados con líneas de negocios y áreas específicas de performance. FEA no considera el modelado de procesos.

3.3 Soluciones Tecnológicas

Las soluciones tecnológicas descriptas en esta sección están divididas en: soluciones de propósito general, soluciones específicas para Gobierno Electrónico, arquitecturas de software y frameworks de desarrollo. Las soluciones de propósito general incluyen productos de integración de software tales como: Bea MessageQ [BEA00] (Sección 3.3.1), WebSphere MQ [IBM08a] (Sección 3.3.2) y Microsoft Message Queue (MSMQ) [Dic98] (Sección 3.3.3). Las soluciones específicas para Gobierno Electrónico comprenden productos de software que tratan necesidades específicas de gobierno como: IAMS Messaging System [Rea02] (Sección 3.3.4) y Hermes Messaging Gateway [CEC07] (Sección 3.3.5). Finalmente, la Arquitectura Orientada a Servicios [Erl05] (Sección 3.3.6) y JADE [TI08a] (Sección 3.3.7) son presentadas como ejemplos de arquitecturas de software y frameworks de desarrollo, respectivamente. La Figura 23 describe las soluciones tecnológicas presentadas y su categorización.

Figura 23: Trabajos Relacionados – Soluciones Tecnológicas



3.3.1 BEA MessageQ

BEA MessageQ [BEA00] es un middleware orientado a mensajes desarrollado por BEA Systems. Siendo un middleware de solución general que provee conectividad a un amplio rango de plataformas de múltiples vendedores de TI, le permite a aplicaciones distribuidas intercambiar mensajes asincrónicos, ofreciendo un conjunto de APIs para servicios de mensajería que pueden ser usadas por aplicaciones cliente para enviar y recibir mensajes usando colas persistentes (almacenadas en disco) o no persistentes (almacenadas en memoria).

BEA MessageQ tiene las siguientes características específicas:

- *Arquitectura* – Posee una arquitectura centralizada con tres componentes principales: (1) Message Queue – un contenedor para los mensajes enviados y recibidos por aplicaciones cliente; (2) Message Queuing Group – un conjunto de Message Queues que residen en un solo sistema, identificadas por direcciones lógicas únicas, y que comparten un conjunto común de recursos; y (3) Message Queuing Bus – una colección de Message Queuing Groups ubicados dentro de una sola red, proveyendo colas de mensajes a aplicaciones cliente, que son usados por éstas para enviar y recibir mensajes. El intercambio de mensajes entre aplicaciones cliente conectadas a diferentes Message Queuing Groups es permitido a través de conexiones inter-grupales, configuradas por un administrador del sistema. Mientras que se puede configurar más de un Message Queuing Bus en la misma red, las aplicaciones clientes deben estar ligadas al mismo Message Queuing Bus para poder comunicarse entre sí. Éste Message Queuing Bus provee a cada aplicación cliente una cola única que contiene todos los mensajes que le fueron enviados.
- *Atributos No Funcionales* – BEA MessageQ satisface los atributos no funcionales como sigue: (1) Confiabilidad – distribuye los mensajes a la cola de destino siguiendo dos modos – recuperable y no recuperable, en el primero, al almacenar los mensajes en un disco local garantiza la distribución confiable de los mismos a pesar del hecho de que la aplicación receptora, el nodo que la aloja o la red entera estén temporalmente no disponibles, mientras que en el segundo modo el mensaje se pierde si no es inmediatamente enviado a la cola de destino, a menos que sea recuperado por un procedimiento de error específico del emisor; (2) Portabilidad – puede ser ejecutado en plataformas de TI que corren diversos sistemas operativos, tales como OpenVMS, UNIX y Windows y puede interactuar con SAP R/3, mainframes IBM, sistemas heredados, y productos MQSeries; (3) Mensajes Auto-descriptos – usando el Lenguaje de Manipulación de Campos (FML) [BEA08] y una colección de funciones del lenguaje C que provee pares de atributos y valores para que los mensajes sean procesados por la aplicación receptora, cambiar el contenido del mensaje no requiere modificaciones intrusivas al código de la aplicación; (4) Paradigma de Comunicaciones – BEA MessageQ soporta comunicaciones sincrónicas y asincrónicas; (5) Mecanismo de Distribución – soporta mecanismos punto-a-punto y publicar-suscribir para la entrega de mensajes; y (6) Limitaciones – maneja mensajes que no excedan los 4MB y hasta 32.000 Message Queuing Groups, donde cada grupo puede contener hasta 999 colas de mensajes [Ser02].
- *Atributos Funcionales* – BEA MessageQ satisface los atributos funcionales como sigue: (a) Servicio de Recupero es provisto permitiendo a los receptores leer mensajes en forma selectiva según los valores de ciertos atributos del mensaje definidos por el emisor, tales como identificador de correlación, tipo de mensaje, clase o prioridad del mensaje; (b) Servicio de Auditoría es provisto; (c) Servicio de Validación no es provisto; (d) Servicio de Transformación no es provisto; (e) Servicio de Autenticación no es provisto; (f) Servicio de Nombrado es provisto, permitiendo a las aplicaciones referirse a las colas de mensajes por nombre en lugar de direcciones, definidas en forma estática – dentro de archivos de configuración, o dinámica – en tiempo de ejecución; (g) Servicio de Localización es provisto, devolviendo la dirección correspondiente a un nombre de cola dado; (h) Servicio de Transacción no es provisto; (i) Servicio de Composición no es provisto; y (j) Servicio de Desarrollo no es provisto – BEA MessageQ no permite el desarrollo de nuevos servicios.

3.3.2 WebSphere MQ

WebSphere MQ [IBM08a] es un middleware orientado a mensajes desarrollado por IBM, un producto de software que permite que aplicaciones distribuidas en diferentes plataformas de TI intercambien información a través de mensajes utilizando una solo API.

WebSphere MQ tiene las siguientes características específicas:

- *Arquitectura* – Posee una arquitectura distribuida que comprende tres components: (1) Message Queue – un contenedor para los mensajes enviados y recibidos por las aplicaciones, (2) MQ Queue Manager – un conjunto de Message Queues ubicadas en el mismo computador y para los cuales se asegura la entrega confiable de mensajes, y (3) Canal – un link de comunicaciones de una vía entre los MQ Queue Managers. MQ Queue Manager provee servicios de mensajería para las aplicaciones de software que requieren comunicarse, permitiéndoles el acceso a colas de mensajes para almacenar y recuperar mensajes. La aplicación puede intercambiar mensajes a través de colas de mensajes administradas por el mismo MQ Queue Manager – las comunicaciones son manejadas en forma local usando memoria inter-proceso, o por diferentes MQ Queue Managers – las comunicaciones son manejadas en forma remota a través de Canales. Los MQ Queue Managers tienen una cola de mensajes distinguible llamada Dead-Letter Queue o Undelivered Message Queue (cola de mensajes no entregados), para almacenar mensajes que no pudieron ser enviados al destino previsto. Aunque múltiples MQ Queue Managers pueden ser ubicados en un solo servidor, se suele distribuirlos en varios servidores y plataformas de TI operados por diversos sistemas operativos [IBM08b][DB05].
- *Atributos No Funcionales*– WebSphere MQ satisface los atributos no funcionales como sigue: (1) Confiabilidad – cuando los mensajes son intercambiados entre colas manejadas por el mismo MQ Queue Manager no se pierden ni se entregan más de una vez, cuando son intercambiados entre diferentes MQ Queue Managers, los mensajes pueden ser enviados de manera persistente – no se pierden ni se entregan más de una vez, o de manera no persistente – pueden ser perdidos si falla la comunicación entre los MQ Queue Managers; (2) Portabilidad - WebSphere MQ release v6.0 puede ser implementado sobre plataformas de TI con diversos sistemas operativos como AIX, HP-UX-PA-RISC, HP-UX Itaniun, Linux for System X (32-bit), Linux for System X (64-bit), Linux for S/390, Linux for System z, iSeries, Solaris SPARC, Solaris x86-64, Windows, y z/OS [IBM08c]; (3) Mensajes Auto-descriptos – un descriptor es asociado con cada mensaje conteniendo información de metadatos – identificador, emisor, tipo de mensaje para determinar si el mensaje requiere o no una respuesta, dirección de destino para enviar la respuesta, tiempo de expiración del mensaje, y representación de datos en el cuerpo del mensaje; (4) Paradigma de Comunicación – mientras que las comunicaciones asincrónicas son soportadas en forma nativa, si una aplicación requiere respuesta a un mensaje enviado, WebSphere MQ simula una comunicación sincrónica uniendo dos acciones asincrónicas; (5) Mecanismo de Distribución – soporta mecanismos punto-a-punto y publicar-suscribir para el envío de mensajes, el primero se ocupa de proveer a cada MQ Queue Manager un componente broker especial que construye la funcionalidad sobre las funciones básicas existentes; y (6) Limitaciones – se provee un parámetro para definir el tamaño máximo de mensaje, siendo por defecto de 64KB y el máximo de 100MB. Sin embargo, si se utiliza DB2 Universal Database, WebSphere MQ puede manejar mensajes de hasta 2GB [Yus04]. WebSphere MQ tambien provee un parámetro para limitar el número de mensajes permitidos en una cola. El parámetro no tiene límite, ya que las colas se manejan en memoria y son asociadas a un directorio en el disco duro. Si la memoria permitida es excedida, los mensajes son guardados temporalmente en el disco duro.
- *Atributos Funcionales* – WebSphere MQ satisface los atributos funcionales como sigue: (a) Servicio de Recupero es provisto, permitiendo a los receptores leer mensajes en forma selectiva según los valores de un identificador; (b) Servicio de Auditoría no es provisto para aplicaciones cliente, sólo para funciones internas del MQ Queue Manager; (c) Servicio de Validación no es provisto; (d) Servicio de Transformación no es provisto; (e) Servicio de Autenticación es provisto; (f) Servicio de Nombrado es provisto, asignando nombres a las colas de mensajes, MQ Queue Managers y Canales; (g) Servicio de Localización es provisto a través del WebSphere MQ Explorer – una interface gráfica de usuario para administrar los MQ Manager Queues y sus objetos relacionados; (h) Servicio de Transacción no es provisto; (i) Servicio de Composición no es provisto; y (j) Servicio de Desarrollo no está soportado. Aunque WebSphere MQ no provee servicios de validación o transformación por sí mismo, puede ser combinado con WebSphere Message Broker [DCGP05] que sí los provee.

3.3.3 Microsoft Message Queue (MSMQ)

Microsoft Message Queue Server (MSMQ) [Dic98][Mic08a] es un middleware orientado a mensajes desarrollado por Microsoft, una solución que permite la comunicación entre aplicaciones intercambiando mensajes asincrónicos. El nuevo release, MSMQ 3.0 provee funcionalidades mejoradas para aplicaciones cliente, tales como intercambio de mensajes usando SOAP y en Internet usando HTTP, y referencias a colas de mensajes usando URLs.

MSMQ tiene las siguientes características específicas:

- *Arquitectura* – Posee una arquitectura distribuida con tres componentes principales: (1) Application Queue – un contenedor para los mensajes enviados y recibidos por las aplicaciones; (2) System Queue – un contenedor sólo para los mensajes generados y mantenidos por MSMQ; y (3) MSMQ Server – un servidor para el manejo de colas de mensajes y el control del intercambio de mensajes. MSMQ administra tres tipos de Application Queues: (a) Message Queues – almacenan mensajes enviados y recibidos por aplicaciones; (b) Administration Queues – almacenan mensajes de acuse de recibo, tanto positivos como negativos, generados por MSMQ en nombre de las aplicaciones; y (c) Response Queues – almacenan respuestas enviadas por las aplicaciones. Tres tipos de System Queues son administradas: (d) Journal Queues – registran mensajes antes de que sean recibidos por las aplicaciones, y mensajes enviados a otros computadores; (e) Dead-Letter Queues – almacenan los mensajes que no pudieron ser enviados; y (f) Report Queues – registran mensajes que permiten realizar el seguimiento del progreso de los mensajes según son transmitidos a través de la empresa. Los servidores MSMQ están asociados a sitios, los que representan un grupo físico de computadores que se comunican entre ellos de manera rápida y económica. Los sitios se conectan a través de vínculos. Se distinguen cuatro tipos de Servidores MSMQ: (i) Primary Site Controller (PSC) – un servidor que controla las copias maestras de información acerca de los computadores y colas manejadas en un sitio. (ii) Primary Enterprise Controller (PEC) – un servidor que mantiene información sobre la configuración de la empresa y certificados para autenticar mensajes, también responsable del ruteo de mensajes dentro y entre sitios; (iii) Backup Site Controller (BSC) – un servidor opcional que mantiene una copia de lectura de las bases de datos del PSC o el PEC, proveyendo además balance de carga y recupero ante fallas; y (iv) MSMQ Routing Server – un servidor para el ruteo dinámico de los mensajes y el manejo de colas intermediarias (colas para almacenar y despachar mensajes) [Mic08b].
- *Atributos No Funcionales*– MSMQ satisface los atributos no funcionales como sigue: (1) Confiabilidad – soporta dos tipos de envío de mensajes: expreso – los mensajes se almacenan en memoria hasta ser enviados, pero se pierden si falla el sistema o es reiniciado, y recuperables – los mensajes se almacenan en una copia de resguardo y no se pierden si falla el computador o el sistema. Es posible recibir mensajes duplicados en ambos tipos de envío. Cuando se intercambian mensajes recuperables, MSMQ soporta semántica transaccional – un mensaje puede ser almacenado o removido de una cola como parte de la transacción. Cuando los mensajes se intercambian como parte de una transacción, MSMQ asegura que serán enviados sólo una vez – los mensajes duplicados son eliminados y recibidos en orden cronológico. (2) Portabilidad – MSMQ está disponible en todas las plataformas Microsoft: CE 3.0, familia de Windows 2000 Server, Windows XP Professional y Windows Server 2003. (3) Mensajes Auto-descriptos – MSMQ usa XML como lenguaje por defecto para describir mensajes, con dos alternativas disponibles – Binary Message y ActiveX Message Formatters, ambos serializando objetos en un formato binario no legible. (4) Paradigma de Comunicación – MSMQ soporta el envío de mensajes asincrónicos y lee mensajes sincrónicos y asincrónicos. Por lo tanto, después de enviar un mensaje, una aplicación siempre retoma su ejecución, mientras que para recibir un mensaje la aplicación decide primero si leerlo en modo sincrónico o asincrónico. En modo sincrónico, la aplicación espera a tener un mensaje disponible, retomando su ejecución después de algún tiempo. En modo asincrónico, MSMQ provee tres mecanismos: i) función de callback – una función definida por el desarrollador e invocada por MSMQ para informarle a la aplicación que se ha recibido un mensaje, ii) evento disparado – un evento Windows para informarle a la aplicación que se ha recibido un mensaje o ha expirado el tiempo de espera; y iii) puerto de completitud – un objeto asociado a sockets o a manejadores de archivo para indicar la terminación de una operación de entrada/salida, usado para notificar a un proceso de Windows NT que se ha recibido un mensaje; (5) Mecanismo de Distribución – mientras que versiones anteriores de MSMQ sólo soportaban comunicaciones punto-a-punto, MSMQ 3.0 soporta el paradigma de publicar-suscribir por medio del envío de múltiples mensajes en tiempo real o enviando un mensaje a una lista de colas de distribución; (6) Limitaciones – MSMQ maneja mensajes de hasta 4MB, con 2GB de capacidad máxima para el almacenamiento de datos en las colas. Sin embargo, tal cantidad puede ser reducida por el código de MSMQ y la estructura interna de datos, permitiendo una capacidad máxima entre 1.4 y 1.6GB [Mic08c].
- *Atributos Funcionales* – MSMQ satisface los atributos funcionales como sigue: (a) Servicio de Recupero no es provisto – si bien sólo es posible recibir el primer mensaje de la cola, MSMQ ofrece una función para visualizar el contenido de un mensaje en la cola sin removerlo; (b) Servicio de Auditoría es provisto, permitiendo registrar mensajes antes de ser recibidos por las aplicaciones; (c) Servicio de Validación no es provisto; (d) Servicio de Transformación no es provisto; (e) Servicio de Autenticación es provisto, con mensajes autenticados con firmas digitales; (f) Servicio de Nombrado es provisto, permitiendo asignar nombre a las colas; (g) Servicio de Localización es provisto de la siguiente manera: los clientes MSMQ pueden usar el MSMQ Directory Service Discovery Protocol para obtener los servidores MSMQ disponibles, una vez que el servidor fue localizado, se pueden usar otros protocolos para obtener información sobre las colas [Mic08d]; (h) Servicio de Transacción es provisto; (i) Servicio de Composición no es provisto; (j) Servicio de Desarrollo no es provisto.

3.3.4 Servicio de Mensajería Inter Agencias (IAMS)

El Servicio de Mensajería Inter-Agencias (IAMS) [Col03] es una solución específica para Gobierno Electrónico desarrollada por Reach – una agencia del Gobierno de Irlanda [Rea08]. IAMS fue desarrollado para satisfacer una necesidad concreta del gobierno irlandés para la distribución de servicios electrónicos relacionados a eventos de vida. Como se explicó en el Ejemplo 17, IAMS soporta el intercambio de información correspondiente a eventos de vida entre la Oficina de Registro General (GRO); Departamento de Asuntos Sociales, Comunitarios y Familiares (DSCFA), Servicios de Identidad de Clientes (CIS), la Oficina Central de Estadísticas (CSO); y hospitales maternales. IAMS le permite a GRO actualizar los detalles de registro relacionados con nacimientos, casamientos, adopciones, nacidos muertos, divorcios y fallecimientos, y ayuda a CIS y a los hospitales maternales a actualizar datos relacionados a los nacimientos a través de navegadores. Adicionalmente, IAMS provee información estadística a CSO.

IAMS tiene las siguientes características específicas:

- *Arquitectura* – IAMS está basado en una estructura centralizada construida sobre su componente principal – el Núcleo de IAMS, el cual coordina el intercambio de mensajes a través de diversos canales, como HTTP/HTTPS, JMS, CORBA/RMI/DCOM, FTP/PSFTP y SMTP/POP3. A pesar del uso de canales múltiples, todos los servicios de mensajería son accesibles a través de la Red Privada Virtual del Gobierno de Irlanda (GVPN). Originalmente, IAMS se basó en el canal BizTalk [Mic08e] de Microsoft para soportar el intercambio de mensajes entre GRO y CIS, con los mensajes transferidos en el formato propietario de BizTalk. Apuntando a la adopción de estándares abiertos, mejoras en la interoperabilidad, y manejo de mecanismos publicar-suscribir para la entrega de mensajes, IAMS fue desarrollado para agregar esas facilidades sobre BizTalk. El núcleo de IAMS está basado en ZOPE [ZC08], un servidor de aplicaciones de código abierto para construir sistema de administración de contenidos, portales y aplicaciones personalizadas. Adicionalmente, IAMS usa el IIS de Microsoft como Servidor Web, el servidor BizTalk de Microsoft, el Microsoft SQL Server y MSMQ.
- *Atributos No Funcionales*– IAMS satisface los atributos no funcionales como sigue: (1) Confiabilidad – IAMS descansa en BizTalk para asegurar mensajería confiable, con el servidor de BizTalk enviando un recibo por cada mensaje recibido en el lado del receptor y enviando un mensaje repetidamente hasta que el recibo es recibido del lado del emisor, y con una entrega de única vez no asegurada, excepto por el uso de un identificador de mensaje contenido en la estructura del mensaje para detectar duplicados; (2) Portabilidad – IAMS es ejecutado sobre Windows 2000 Server y es probable que corra en otras plataformas Windows; (3) Mensajes Auto-descriptos – adopta XML como el estándar para la representación de mensajes, con el formato de mensaje incluyendo un sobre con contenido predefinido; (4) Paradigma de Comunicaciones – adicionalmente al soporte de comunicaciones asincrónicas, se supone que puede soprtar también modo sincrónico, ya que RMI es mencionado como un posible canal de comunicación para interactuar con el núcleo de IAMS, si bien esta facilidad carece de documentación; (5) Mecanismo de Distribución – El núcleo de IAMS soporta mecanismos punto-a-punto y publicar-suscribir para el envío de mensajes; y (6) Limitaciones – no hay información disponible.
- *Atributos Funcionales* – IAMS satisface los atributos funcionales como sigue: (a) Servicio de Recupero – no hay información para evaluar esta funcionalidad; (b) Servicio de Auditoría es provisto, con mensajes registrados usando la funcionalidad provista por BizTalk, y el recupero y monitoreo de los mensajes registrados es provisto usando una página ASP; (c) Servicio de Validación es provisto, por ejemplo con el contenido del sobre que incluye la dirección de origen, de destino y tipo de mensaje completamente validados; (d) Servicio de Transformación - no hay información para evaluar esta funcionalidad; (f) Servicio de Nombrado – es provisto en base a un servicio que mantiene los identificadores de origen y destino; (g) Servicio de Localización es provisto sobre las mismas bases del Servicio de Nombrado; (h) Servicio de Transacción - no hay información para evaluar esta funcionalidad; (i) Servicio de Composición no es provisto; y (j) Servicio de Desarrollo no es provisto.

3.3.5 Hermes Messaging Gateway (Hermes2)

Hermes Messaging Gateway (Hermes2) [CEC07] es una solución específica para Gobierno Electrónico desarrollada por el Centro de Desarrollo de Infraestructura de Comercio Electrónico (CECID) de la Universidad de Hong Kong para el Gobierno de Hong Kong. Hermes2 le permite a las aplicaciones de software intercambiar información en varios formatos, dentro y a través de las organizaciones.

Hermes2 tiene las siguientes características específicas:

- *Arquitectura* – Posee una arquitectura distribuida con cuatro componentes principales: (1) el núcleo Corvus-Core – un componente que provee la capa de transporte para intercambio de datos; (2) un repositorio – un componente que maneja la persistencia de la información; (3) ebXML-plug-in – un plug-in para dar formato a mensajes de acuerdo a las especificaciones del Servicio de Mensajes de ebXML 2.0 [OAS02]; y (4) AS2-plug-in – un pug-in para dar formato a mensajes de acuerdo a los estándares de AS2 (Applicability Statement 2) [Dru05]. El concepto principal relativo a las comunicaciones usado por Hermes es Asociación, un canal de comunicación de una vía para que una entidad se comunique con otra.
- *Atributos No Funcionales*– Hermes2 satisface los atributos no funcionales como sigue: (1) Confiabilidad – asegura mensajería confiable usando dos parámetros definidos para cada Asociación: Acuse de Recibo Requerido el cual le solicita al receptor enviar un mensaje al emisor, y Eliminación de Duplicados el cual solicita al receptor eliminar mensajes duplicados; (2) Portabilidad - Hermes2 puede ser implementado sobre plataformas de TI con Windows XP, Linux and Solaris; (3) Mensajes Auto-descriptos – Hermes2 ha adoptado ebXML y AS2 como estándares de mensajes; (4) Paradigma de Comunicación – maneja comunicaciones asincrónicas, sin embargo un parámetro de la Asociación permite especificar si el emisor requiere recibos sincrónicos o asincrónicos; (5) Mecanismo de Distribución – soporta envío de mensajes punto-a-punto; y (6) Limitaciones – no hay información disponible.
- *Atributos Funcionales* – Hermes2 satisface los atributos funcionales como sigue: (a) Servicio de Recupero es provisto con el parámetro de orden de mensaje garantizando para una Asociación dada que las aplicaciones receptoras obtendrán los mensajes en el orden en que fueron enviados; (b) Servicio de Auditoría no es provisto; (c) Servicio de Validación – no hay información disponible para evaluar esta funcionalidad; (d) Servicio de Transformación es provisto, los mensajes son transformados utilizando un formato de mensaje canónico; (e) Servicio de Autenticación no es provisto; (f) Servicio de Nombrado es provisto, se asignan nombres a Asociaciones dependiendo del uso de ebXML o AS2 para el formato de datos; (g) Servicio de Localización es provisto con una interface gráfica de usuario para manejar y consultar Asociaciones; (h) Servicio de Transacción – no hay información disponible; (i) Servicio de Composición no es provisto; y (j) Servicio de Desarrollo no es provisto.

3.3.6 Arquitectura Orientada a Servicio

La Arquitectura Orientada a Servicio (SOA) [Erl05] es un patrón de arquitectura de software sin propiedad definida. La característica de la arquitectura es que todas las tareas de software y los procesos implementados son diseñados como servicios a ser consumidos sobre la red. Detalles acerca de SOA y sus tecnologías subyacentes fueron presentados en la Sección 2.4.2.

SOA tiene las siguientes características específicas:

- *Arquitectura* – SOA tiene una arquitectura distribuida construida sobre el concepto de Servicio. Un Servicio es un recurso abstracto provisto y consumido sobre una red, capaz de ejecutar tareas que conforman una funcionalidad coherente desde el punto de vista del proveedor y receptor de servicios [W3C04b]. La arquitectura SOA consiste de tres roles y tres operaciones. Los roles son: (1) Proveedor de Servicio – una entidad que provee un servicio disponible en la red; (2) Solicitante de Servicio – una entidad deseosa de consumir un servicio disponible en la red; (3) Registro de Servicio – un repositorio que permite conectar proveedores y solicitantes de servicios. Las operaciones son: (a) Publicar – el acto de anunciar o publicitar un servicio; (b) Buscar – el acto de buscar un servicio que satisfaga ciertas condiciones; y (c) Ligar – el acto de crear relaciones cliente-servidor entre proveedores y solicitantes de servicios. Los roles y operaciones se relacionan como sigue: un proveedor de servicio publica un servicio en el registro de servicios; el registro de servicios los publicita; el solicitante de servicios busca un servicio que satisfaga ciertas condiciones; el solicitante encuentra el servicio requerido en el registro y lo vincula a su aplicación.
- *Atributos No Funcionales*– SOA satisface los atributos no funcionales como sigue: (1) Confiabilidad – mensajería confiable es soportada por WS-Reliability [OAS04], la extensión de SOAP responsable de asegurar que los mensajes no se pierdan, dupliquen or reordenen; (2) Portabilidad – SOA es una arquitectura de plataforma neutral, soportando servicios distribuidos sobre diferentes plataformas que pueden inter-operar a través de mensajes intercambiados en formatos estandarizados; (3) Mensajes Auto-descriptos – la implementación de Servicios Web de SOA ha adoptado SOAP como el protocolo de comunicación para intercambio de mensajes, con XML para el framework de mensajería extensible; (4) Paradigma de Comunicaciones – los Servicios Web soportan comunicaciones sincrónicas y asincrónicas; (5) Mecanismo de Distribución – los Servicios Web soportan el envío de mensajes bajo la modalidad punto-a-punto y publicar-suscribir, el primero siguiendo las especificaciones de

WS-Notification que recomienda interacciones comandadas por eventos y basadas en notificaciones, usadas por varios MOMs [OAS06b]; y (6) Limitaciones – la especificación de SOAP no provee limitaciones en cuanto a tamaño de mensaje, pero este tamaño está restringido por el tamaño máximo de mensaje soportado por el protocolo de comunicación subyacente. Por ejemplo, el intercambio de mensajes extensos usando HTTP requiere sintonizar el parámetro `maxRequestLength` en el archivo de configuración web. Adicionalmente, los mensajes SOAP aceptan archivos adjuntos pero restringen la medida de los mismos, por ejemplo a 4GB para cada anexo MIME. El comportamiento de los Servicios Web cuando se intercambian mensajes con adjuntos está especificado en la extensión – SOAP Messages with Attachments [W3C00].

- *Atributos Funcionales* – SOA no provee funcionalidades específicas por sí misma, pero provee un framework para construir aplicaciones. El framework permite incluir la funcionalidad requerida en la forma de un Servicio Web o un componente intermediario, a ser procesado por el servidor SOAP de acuerdo a los contenidos de los encabezados de los mensajes. Seguidamente analizaremos cómo SOA y particularmente su implementación de Servicios Web soportan funcionalidades específicas en tiempo de diseño y en tiempo de ejecución: (a) Servicio de Recupero puede ser agregado como un servicio web intermediario que almacena mensajes, permitiendo a las aplicaciones cliente recuperar los mismos en el orden en que fueron requeridos; (b) Servicio de Auditoría puede ser agregado como un componente intermediario; (c) Servicio de Validación puede ser agregado como un componente intermediario; (d) Servicio de Transformación puede ser agregado como un componente intermediario; (e) Servicio de Autenticación puede ser agregado implementando las especificaciones de WS-Security [OAS06a]; (f) Servicio de Nombrado es provisto por UDDI [OAS07a]; (g) Servicio de Localización es provisto por UDDI; (h) Servicio de Transacción puede ser implementado basado en WS-Coordination [OAS07b] – un framework extensible para coordinar las acciones de aplicaciones distribuidas, y WS-AtomicTransaction [OAS07c] – un framework para especificar transacciones atómicas o generalmente para coordinar actividades con propiedad de atomicidad (todo o nada); (i) Servicio de Composición es soportado a través de diferentes lenguajes usados para especificar procesos de negocio, tal como el Web Services Business Process Execution Language (WSBPEL) [OAS07d] y motores de workflow usados para ejecutar tales procesos, por ejemplo jBPM [RH08]; y (j) Servicio de Desarrollo no es provisto.

3.3.7 Java Agent Development Framework (JADE)

El Java Agent Development Framework (JADE) [TI08a][BCPR07] es un framework de desarrollo de código abierto, basado en agentes, construido por el laboratorio de Telecom Italia [TI08b]. JADE es una solución de middleware que ofrece servicios de alto nivel para el desarrollo y ejecución de aplicaciones basadas en agentes. Sigue los fundamentos y especificaciones de Foundation for Intelligent Physical Agents (FIPA) [FIPA08], las cuales proveen roles de agentes, ontología, y el Lenguaje de Comunicación de Agentes (ACL). JADE incluye librerías requeridas para desarrollar agentes de aplicación y el ambiente de ejecución que provee los servicios básicos que deben estar presentes y activos en los dispositivos, antes de que los agentes puedan ser ejecutados.

JADE tiene las siguientes características específicas:

- *Arquitectura* – JADE ofrece una arquitectura distribuida basada en agentes que comprende los siguientes componentes: (1) Agente – un proveedor de servicios para aplicaciones cliente, únicamente identificado por nombre, y capaz de proveer varios servicios; (2) Contenedor – una instancia del entorno de ejecución de JADE ejecutada sobre diferentes plataformas y que comprende un conjunto de agentes, donde los agentes en un contenedor pueden comunicarse a través de un Protocolo de Transporte de Mensajes (MTP) dependiente de la plataforma como RMI, IIOP, HTTP, etc. y se pueden mover entre diferentes contenedores mientras mantienen su estado; y (3) Plataforma JADE – todos los contenedores referenciados por el contenedor principal, que proveen un mayor grado de abstracción y así permiten ocultar a las aplicaciones y agentes los detalles del hardware subyacente, sistema operativo, red, etc. Adicionalmente, los agentes tienen tres roles: (a) Agent Management System (AMS) – el agente que controla el acceso y el uso de la plataforma; (b) Agent Communication Channel (ACC) – los agentes que proveen el contacto entre agentes dentro y fuera de la plataforma, soportando la interoperabilidad entre las mismas; y (c) Directory Facilitator (DF) – los agentes que ofrecen servicios de localización, por ejemplo en la forma de páginas amarillas.
- *Atributos No Funcionales*– JADE satisface los atributos no funcionales como sigue: (1) Confiabilidad – JADE ofrece mensajería confiable a través de agentes en el rol de ACC, ofreciendo servicios de mensajería confiables, ordenados y precisos como método de comunicación por defecto; también se puede usar JMS con JADE para

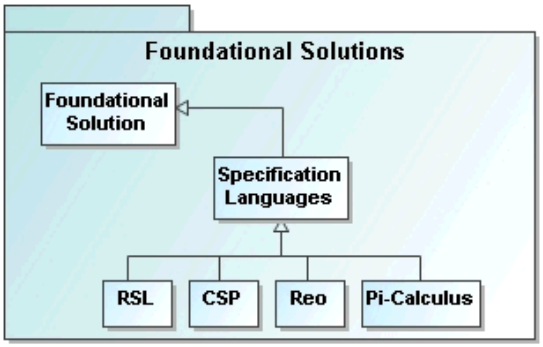
proveer mensajería confiable [CCL03]; (2) Portabilidad – JADE soporta intercambio de mensajes entre agentes residentes en distintas plataformas; (3) Mensajes Auto-descriptos – JADE adopta el estándar de Especificación de Estructura de Mensajes ACL [FIPA02] para representar estructuras de mensajes y proveer una API para el manejo de los mismos; (4) Paradigma de Comunicaciones – JADE soporta tanto comunicaciones sincrónicas como asincrónicas, con tres métodos de intercambio de mensajes dependiendo de la ubicación del agente: si los agentes están en el mismo contenedor se usa el mecanismo de eventos de JAVA; si los agentes se encuentran en distintos contenedores pero en el mismo host, se usa el Método de Invocación Remota (RMI); y si los agentes están ubicados en diferentes hosts se usan los protocolos estándar IIOP y OMG IDL [BGN04]; (5) Mecanismo de Distribución – JADE ofrece mecanismo punto-a-punto para el envío de mensajes; y (6) Limitaciones – el tamaño máximo de mensaje es determinado por la tecnología subyacente y no por JADE.

- *Atributos Funcionales* – JADE satisface los atributos funcionales como sigue: (a) Servicio de Recupero es provisto, con agentes capaces de recuperar mensajes de las colas de acuerdo a un contenido preestablecido en plantillas de mensajes; (b) Servicio de Auditoría es provisto (c) Servicio de Validación es provisto por medio de un agente dedicado de manejo de contenido [Fab06]; (d) Servicio de Transformación no es provisto; (e) Servicio de Autenticación es provisto a través de certificados asignados a los contenedores [Vit04]; (f) Servicio de Nombrado es provisto por la plataforma JADE para asegurar que cada agente tenga un nombre único [BCPR07]; (g) Servicio de Localización es provisto por medio de un servicio de páginas amarillas [BCPR07]; (h) Servicio de Transacción no es provisto; (i) Servicio de Composición es provisto a través de tecnologías adicionales construidas sobre JADE [LKM06] como JXTA [Sun08b]; y (j) Servicio de Desarrollo es provisto por el framework de desarrollo de JADE.

3.4 Soluciones Fundacionales

Las soluciones fundacionales presentadas en esta sección comprenden cuatro lenguajes de especificación formal: RAISE Specification Language (RSL) [RLG92] (Section 3.4.1), Communicating Sequential Processes (CSP) [Ros05] (Sección 3.4.2), Reo [Arb02] (Sección 3.4.3) y Pi-calculus [MPW89] (Sección 3.4.4). Todos ellos son capaces de describir sistemas distribuidos en diferentes niveles de abstracción, expresar el comportamiento de mensajería siguiendo comunicaciones sincrónicas o asincrónicas, y verificar especificaciones de bajo nivel con respecto a los de alto nivel. Los lenguajes están ilustrados en la Figura 24 debajo, y son explicados en secuencia de acuerdo al marco de evaluación presentado en la Sección 3.1.

Figura 24: Trabajos Relacionados – Soluciones Fundacionales



3.4.1 Lenguaje de Especificación RAISE

RAISE es la abreviatura de Rigorous Approach to Industrial Software Engineering. Comprende un lenguaje de especificación, un método de desarrollo y un conjunto de herramientas de desarrollo. Desarrollado por el RAISE Language Group en 1992, el lenguaje RSL de especificación de RAISE [RLG92] es un lenguaje de especificación formal de amplio espectro que permite la descripción y análisis riguroso de especificaciones abstractas, y su refinamiento gradual hacia especificaciones más concretas que pueden ser finalmente transformadas, posiblemente en forma automática, a código de programación. RSL soporte tres estilos de especificación: (1) aplicativo – especificaciones enteramente basadas en la definición de funciones y cómo se aplican a argumentos; (2) imperativo – especificaciones basadas en variables y en cómo cambian sus valores a través del tiempo; y (3) concurrente – las especificaciones

comprenden procesos que son ejecutados concurrentemente mientras se comunican por medio de canales. Adicionalmente, RSL permite escribir especificaciones orientadas a propiedades definidas en términos de estados abstractos y sus propiedades requeridas, y especificaciones orientadas a modelos definidos en términos de estados concretos y cambios de estado permitidos. Una especificación RSL está compuesta de módulos, donde cada uno contiene definiciones de tipo, valores, variables, canales, axiomas y otros módulos.

RSL tiene las siguientes características específicas:

- *Estilo de Especificación* – RSL permite escribir especificaciones basadas en estado y basadas en acciones. Por ejemplo, una especificación basada en estado puede ser escrita en lenguaje aplicativo, especificando propiedades del sistema por medio de definiciones de funciones implícitas usando pre- y post-condiciones: si los argumentos de una función satisfacen las precondiciones, el resultado debe satisfacer las post-condiciones. Tales especificaciones pueden ser también escritas en estilo imperativo, con funciones que especifican cómo puede cambiarse el estado de las variables. Las especificaciones basadas en acciones describen cómo el estado (abstracto) de un sistema y su comportamiento subsecuente, cambia después de realizar una serie de acciones (o invocación de funciones). En estilo concurrente, RSL permite especificar cómo se sincronizan las acciones a través de canales.

- *Semántica* – Existe una semántica formal de RSL, descripta en estilo operacional [BD93], denotacional [Mil93] y axiomático [RMG96], y sirven como base formal para razonar y refinar especificaciones escritas en RSL.

- *Refinamiento* – El principio de desarrollo en RAISE es Inventar y Verificar – a fin de refinar un modulo abstracto dado A en un módulo más concreto, el módulo B debería ser primero inventado, y luego probado que implementa (o es un refinamiento) de A. Para probar que B implementa A, dos propiedades se deben verificar: (1) Preservación de Propiedades – todas las propiedades verdaderas de A deben también ser verdaderas para B, y (2) Sustitutividad – una instancia de A puede ser reemplazada en una especificación mayor por una instancia de B, y la especificación resultante debería implementar la anterior.

- *Herramientas* – Existe un conjunto de herramientas que soportan el proceso de desarrollo en RAISE usando el lenguaje RSL. Hay herramientas disponibles para: (1) escribir especificaciones – librerías de emacs que reconocen la sintaxis de RSL; (2) controlar especificaciones – controlador de tipo y generador de condiciones de confianza para RSL; (3) documentar especificaciones – RSL pretty printer; (4) verificar especificaciones – traductores de RSL a demostradores de teoremas como PVS y SAL; (5) ejecutar especificaciones – traductores de RSL a lenguajes de programación como C++ o SML; y (6) integrar especificaciones a procesos de desarrollo de software – traducen diagramas de clases UML a RSL.

3.4.2 Communicating Sequential Processes

CSP es una notación y cálculo para razonar acerca del comportamiento de sistemas concurrentes [Hoa78]. El lenguaje ha evolucionado considerablemente desde su introducción, dando crecimiento junto con CCS [Mil80], a un área fructífera de investigación de álgebra de procesos [Ros05]. En CSP, un sistema es modelado como un conjunto de procesos que interactúan entre sí y con su entorno, intercambiando mensajes vía comunicaciones sincrónicas. Cada proceso tiene su propio estado, pero este estado no es visible desde el mundo exterior. CSP es, por lo tanto, construido sobre los conceptos de procesos y acciones. Un proceso toma parte en acciones, que en conjunto definen su alfabeto. El comportamiento de un proceso es especificado en términos de: series de acciones en las que el proceso puede llegar a tomar parte (traza), conjunto de acciones en las que un proceso no quiere tomar parte después de una traza dada (rechazo), y conjuntos de trazas después de la cual un proceso puede realizar una secuencia infinita de acciones internas (divergencias). A través de la consideración de traza y rechazos, se puede modelar no determinismo en CSP, cuando un proceso está deseoso de involucrarse o de rechazar ciertas acciones después de tomar parte de una traza. Dos suposiciones principales soportan las comunicaciones CSP: (1) las acciones son instantáneas y (2) las acciones sólo ocurren cuando el proceso y su entorno están ambos deseosos de tomar parte de ellas, y en tal caso, la acción debe ocurrir.

CSP tiene las siguientes características específicas:

- *Estilo de Especificación* – CSP soporta especificaciones basadas en acciones, donde el comportamiento de un proceso es especificado por sus trazas, rechazos y divergencias. CSP, al igual que otros lenguajes similares de álgebra de procesos, se enfoca sólo en la parte de comunicaciones de un sistema. Ellos describen y razonan sobre sistemas en términos de comunicaciones entre procesos y sus entornos, ignorando el estado interno del proceso.
- *Semántica* – El significado de un proceso CSP puede ser descripto matemáticamente en tres formas diferentes [Ros05]: operacional, denotacional y algebraico. Operacionalmente, un proceso es interpretado como un diagrama de transiciones etiquetado donde un proceso está involucrado en acciones tanto visibles (para su entorno) como ocultas (internas al proceso) y en cambios de un estado a otro, como resultado de las acciones. Diferentes semánticas denotacionales han sido definidas para CSP en términos de trazas, rechazos, divergencias, y otras, como se mencionó brevemente arriba. Adicionalmente, la semántica algebraica para CSP es provista por medio de leyes algebraicas [Sou84].

- *Refinamiento* – Diferentes relaciones de refinamiento pueden ser definidas para CSP dependiendo del modelo semántico aplicado, para relacionar una especificación abstracta (proceso) con una más concreta (otro proceso). Por ejemplo, (1) refinamiento de trazas – un proceso Q refina un proceso P si todas las trazas de Q son posibles trazas de P; (2) refinamiento de fallas – un proceso Q refina un proceso P, si refina la traza de P y es también más determinístico, rechazando no más acciones después de una traza de las que rechaza P; y (3) refinamiento de fallas/divergencias – un proceso Q refina un proceso P, si refina las fallas de P y Q no tiene más divergencias de las que tiene P [GGH05]. Estas relaciones de refinamiento pueden ser usadas para probar diferentes clases de propiedades: propiedades de seguridad con refinamiento de traza, propiedades de deadlock con refinamiento de fallas, y propiedades de liveness con refinamiento de fallas/divergencia [GGH05].

- *Herramientas* – El desarrollo de varias herramientas ha contribuido a la utilización de CSP, tales como: (1) FDR – una herramienta comercial para controlar condiciones de correctitud, como deadlock- y livelock-freedom como también propiedades de seguridad y de liveness; (2) ProBE – una herramienta que facilita un mayor entendimiento del comportamiento de un proceso mediante la interacción con él, en donde el usuario selecciona una traza y en respuesta observa cómo evoluciona el proceso; (3) Casper – una herramienta para producir descripciones CSP de protocolos de seguridad especificados en un lenguaje abstracto; y (4) vim - vi iMproved es un editor que soporta el proceso de escritura de especificaciones en CSP.

3.4.3 Reo

Reo es un lenguaje de coordinación de componentes basado en canales [AM02]. Está basado en cuatro conceptos principales: (1) instancias de componentes, (2) canales, (3) nodos y (4) conectores. Una instancia de componente es un conjunto no vacío de entidades activas como procesos, agentes, actores, etc. que puede comunicarse con otras entidades por medio de operaciones de entrada/salida sobre el (posiblemente dinámico) conjunto de canales a los que está conectado. Un canal es un medio de comunicación entre dos instancias de componentes, con exactamente dos extremos. Hay dos tipos de extremos de canal: (1) sink – envía datos hacia afuera del canal y (2) source – ingresa datos dentro del canal. Un nodo es un arreglo de extremos de canal donde los canales pueden ser conectados. Un conector es un conjunto de extremos de canal organizados en un gráfico de nodos y arcos, tal que: cero o más extremos de canal coinciden en cada nodo; cada extremo de canal coincide exactamente con un nodo, y existe un arco entre dos nodos si y solo si hay un canal cuyos extremos coinciden con cada uno de esos nodos. En Reo, un sistema es modelado como un conjunto de instancias de componentes ejecutadas en distintas ubicaciones, comunicándose por conectores que coordinan sus actividades, y capaces de moverse de una ubicación a otra. Tanto las instancias de componentes como los conectores son móviles. Reo se enfoca en los conectores y su composición, y no en las entidades unidas por el mismo. Un “lenguaje de pegamento” para coordinar procesos concurrentes [Arb02] puede ser usado para especificar la construcción composicional de conectores que coordinan instancias de componentes en sistemas basados en componentes.

Reo tiene las siguientes características específicas:

- *Estilo de Especificación* – Reo permite, por medio de conectores, la especificación de estructuras de comunicación entre instancias de componentes en un sistema basado en componentes. Podría ser usado para especificar propiedades de la arquitectura de tales sistemas y soporta especificaciones de su comportamiento basadas en acciones.

- *Semántica* – Dos semánticas operacionales se han definido para los conectores de Reo: una en términos de relaciones sobre timed data streams [AR02] y otras en términos de autómatas con restricciones [BSAR06].

- *Refinamiento* – Se puede aplicar refinamiento a conectores Reo. Usando autómatas con restricciones, es posible determinar si dos conectores tienen el mismo comportamiento observable o uno es un refinamiento del otro.

- *Herramientas* – Eclipse Coordination Tool [Sen08] es un conjunto de plug-ins para el entorno de desarrollo de Eclipse, comprendiendo herramientas visuales e integradas para soportar el modelado y construcción de circuitos Reo. Las herramientas permiten transformar los modelos creados en autómatas con restricciones, y luego en código Java.

3.4.4 Pi-Calculus

Pi-Calculus es un cálculo de procesos para especificar y analizar propiedades de sistemas que consisten de agentes que interactúan entre sí y cuya configuración o sus alrededores está cambiando continuamente [MPW89]. Pi-Calculus está basado en la noción de nombres, que son usados para identificar: procesos – abstracciones de threads de control independientes, y canales – vínculos abstractos entre dos procesos [Win02]. La sintaxis de Pi-calculus permite representar procesos, composición paralela de procesos, comunicación sincrónica entre procesos por medio de canales, creación de nuevos canales, y replicación de procesos. La diferencia entre Pi-Calculus y otros trabajos previos en álgebra de procesos como CCS o CSP, es que Pi-Calculus tiene la habilidad de intercambiar canales como datos a través de otros canales. Esta cualidad permite modelar la movilidad de procesos, reflejando los cambios en las estructuras de comunicación que conectan los procesos. En la práctica, Pi-Calculus es recomendado para describir procesos concurrentes que se comunican mediante el intercambio de mensajes.

Pi-Calculus tiene las siguientes características específicas:

- *Estilo de Especificación* – Pi-Calculus permite especificaciones basadas en acciones con el comportamiento del proceso descripto por medio de expresiones algebraicas que incluyen los nombres de las acciones (mensajes de entrada/salida) que el proceso puede ejecutar.

- *Semánticas* – Tres tipos de semánticas fueron definidas para Pi-Calculus: operacional – basada en sistemas de transición etiquetadas (LTS) [Sob07], denotacional - basada en categorías de functores [Sta96], y coalgebraicas – basadas en el mapeo del cálculo de procesos con axiomas estructurales de los modelos coalgebraicos [BM02].

- *Refinamiento* – A fin de comparar procesos, Pi-Calculus aplica bisimulación – un tipo de equivalencia de comportamiento entre procesos. Tres tipos de equivalencias de bisimulación fueron definidos para Pi-Calculus: temprana, tardía y abierta [Mil99][San93]. Debido a la naturaleza del pasaje de valores de Pi-Calculus, las bisimulaciones temprana y tardía difieren en el orden de las transiciones de comparación de valores de un proceso contra otro proceso: si la elección de comparar valores es hecha antes (temprana) o después (tarde) de la elección del siguiente estado del proceso. La bisimulación abierta fue introducida como una relación más fina que las bisimulaciones temprana y tardía y es preservada por todos los operadores de Pi-Calculus.

- *Herramientas* – Mobility Workbench es una herramienta para la manipulación y el análisis de sistemas concurrentes móviles especificados en Pi-Calculus usando equivalencias de bisimulación. [VMDE07]. Adicionalmente, Pi-Calculus for SOA [PTF08] es una herramienta que soporta coreografía de Servicios Web con Pi-Calculus, permitiendo generar Servicios Web basados en Java a partir de las descripciones de coreografía expresadas en el lenguaje WS-CDL, cuya semántica formal fue definida en Pi-Calculus.

3.5 Evaluación de Soluciones

Este capítulo introdujo varias soluciones organizacionales, tecnológicas y fundacionales que pueden soportar el desarrollo de Gobierno Integrado. El propósito de esta sección es comparar esas soluciones dentro de cada categoría, siguiendo el marco de evaluación introducido en la Sección 3.1.

Se describieron seis soluciones organizacionales en la Sección 3.2: tres Marcos de Interoperabilidad – e-GIF, NZ e-GIF y EIF, y tres Arquitecturas Empresariales – de Zachman, SAGA y FEA. Todos ellos, excepto la Arquitectura de Zachman fueron definidos para tratar necesidades particulares de Gobierno Electrónico. Comparando los Marcos de Interoperabilidad y las Arquitecturas Empresariales, los últimos pueden ser aplicados de manera más amplia, proveyendo un contexto organizacional para las soluciones de interoperabilidad y ofreciendo una guía estratégica para manejar recursos organizacionales.

En cuanto al soporte para modelar las perspectivas y para interoperabilidad, estas soluciones se comparan como sigue:

- *Vistas* – Todas las soluciones proveen herramientas para modelar la Vista de la Información, Vistas del Proceso y de la Tecnología, dedicando menos atención a la Vista de la Organización; e-GIF no la considera en absoluto.

- *Interoperabilidad* – La interoperabilidad técnica es soportada por todas las soluciones excepto la arquitectura de Zachman. La interoperabilidad semántica es tratada parcialmente por e-GIF y NZ e-GIF, y completamente por EIF. La interoperabilidad semántica es imperativa para que EIF consiga su objetivo. Las soluciones de arquitectura empresarial tratan la interoperabilidad semántica mayormente para clasificar entidades. La interoperabilidad organizacional es tratada parcialmente por todas las soluciones excepto NZ e-GIF.

Una comparación detallada de soluciones organizacionales se incluye en el Capítulo 6.

En la Sección 3.3 se introdujeron siete soluciones tecnológicas: tres soluciones de propósito general – BEA MessageQ, WebSphere MQ y MSMQ, dos soluciones específicas para Gobierno Electrónico – IAMS y Hermes2, una arquitectura de software – SOA, y un framework de desarrollo – JADE. Entre ellas, dos soluciones están basadas en arquitecturas centralizadas – BEA MessageQ y IAMS, mientras que las otras se basan en arquitecturas distribuidas.

En cuanto a atributos funcionales y no funcionales, las soluciones se comparan como sigue:

- *Atributos No Funcionales*– (1) Confiabilidad – Todas las soluciones aseguran que los mensajes no se pierdan; en BEA MessageQ y MSMQ, el modo recuperable debe ser usado para asegurar esta propiedad. WebSphere MQ, Hermes2 y JADE garantizan que los mensajes no se duplican; esta propiedad es asegurada por MSMQ cuando los mensajes son intercambiados dentro de una transacción y para SOA cuando se aplica WS-Reliability. (2) Portabilidad – BEA MessageQ y WebSphere MQ están disponibles para varias plataformas de TI, MSQM y IAMS corren bajo Windows, y SOA y JADE no dependen de la plataforma. (3) Mensajes Auto-descriptos – MSMQ, IAMS y SOA soportan XML como sintaxis para escribir mensajes, Hermes2 soporta ebXML (vocabulario XML) y AS2, mientras las otras utilizan sus propias sintaxis: BEA MessageQ - Field Manipulation Language (FML), WebSphere MQ - Message Descriptor, y JADE - ACL Message Structure Specification. (4) Paradigma de Comunicaciones – Las comunicaciones asincrónicas son soportadas nativamente por todas las soluciones, en tanto que WebSphere MQ, SOA y JADE también soportan comunicaciones sincrónicas. MSMQ puede también leer mensajes sincrónicamente – después de requerir la lectura de un mensaje, la aplicación espera hasta que el mensaje esté disponible en la cola. (5) Mecanismo de Distribución – Hermes2 y JADE soportan sólo envío de mensajes punto-a-punto, mientras los demás soportan envío punto-a-punto y publicar-suscribir. (6) Limitaciones – BEA MessageQ y MSMQ manejan mensajes de no más de 4MB, y WebSphere MQ de no más de 100MB. No se provee información acerca de tamaño máximo de mensajes para IAMS y Hermes2. SOA y JADE no imponen restricciones acerca del tamaño del mensaje, pero dependen de las restricciones impuestas por las tecnologías subyacentes.
- *Atributos Funcionales* – (a) Servicio de Recupero – MSMQ no permite recuperar mensajes en un orden diferente a First-In-First-Out (FIFO). No hay información de esta facilidad para IAMS. Todas las demás soluciones excepto SOA proveen este servicio como una funcionalidad incorporada. Para SOA, un servicio Web intermediario puede ser implementado para proveer esta funcionalidad. (b) Servicio de Auditoría – Sólo MessageQ, MSMQ, IAMS y JADE permiten la auditoría de mensajes por medio de funciones incorporadas; SOA soporta este servicio por medio de un servicio Web intermediario. (c) Servicio de Validación – Sólo IAMS, JADE y SOA (a través de un Servicio Web intermediario) permiten validación de mensajes; no hay información disponible para Hermes2. (d) Servicio de Transformación – Sólo Hermes2 y SOA (a través de un Servicio Web intermediario) son capaces de transformar mensajes; no hay información disponible para IAMS. (e) Servicio de Autenticación – Sólo MessageQ y Hermes2 no proveen este servicio. (f) Servicio de Nombrado – Todas las soluciones proveen este servicio. (g) Servicio de Localización – Todas las soluciones proveen este servicio. (h) Servicio de Transacción – Sólo MSMQ y SOA implementan semánticas de transacción; la implementación de este servicio en SOA requiere que se cumplan las especificaciones de WS-Coordination y WS-AtomicTransaction. No hay información disponible para IAMS y

Hermes2. (i) Servicio de Composición – Sólo SOA y JADE permiten este servicio. (j) Servicio de Desarrollo – Sólo JADE y SOA permiten el desarrollo de nuevos servicios; JADE provee un entorno y una metodología de desarrollo.

Una comparación detallada de soluciones tecnológicas se incluye en el Capítulo 6.

Finalmente, se introdujeron cuatro soluciones fundacionales – RSL, CSP, Reo y Pi-Calculus. RSL es un lenguaje de amplio espectro para especificar sistemas de uso intensivo de datos a diferentes niveles de abstracción. CSP y Pi-Calculus son lenguajes y cálculos de procesos para modelado y análisis de sistemas distribuidos, ambos asumen estructuras de comunicaciones, el primero estáticas, el segundo dinámicas. Reo es un lenguaje para coordinar sistemas basados en componentes.

Las características de las soluciones fundacionales se comparan como sigue:

- *Estilo de Especificación* – Todas las soluciones fundacionales presentadas soportan especificaciones basadas en acciones. Adicionalmente, RSL también soporta especificaciones basadas en estados.

- *Semánticas* – Tres tipos de semánticas formales – operacional, denotacional y axiomática – son provistas para RSL, CSP y Pi-Calculus. Sólo semánticas operacionales son provistas para Reo.

- *Refinamiento* – El refinamiento en RSL se basa en preservación de propiedades y sustitutividad de especificaciones. Tres tipos de refinamiento son definidos para CSP – refinamiento de traza, refinamiento de falla y refinamiento de falla/divergencia. En Reo y Pi-Calculus, las especificaciones están relacionadas a través de varias formas de equivalencia de comportamientos.

- *Herramientas* – Se dispone de varias herramientas para RSL: para edición y documentación de especificaciones, para generar condiciones de confianza, para traducir especificaciones a programas o teorías, para generar especificaciones para modelos de desarrollo. Las herramientas de CSP soportan: escritura de especificaciones, control de condiciones de correctitud, simulación de interacciones con el sistema, y generación de procesos desde protocolos. Las herramientas de Reo soportan modelado y construcción de especificaciones, y generación de código. Pi-Calculus es soportado por herramientas para modelado, verificación y desarrollo.

Una comparación detallada de soluciones fundacionales se incluye en el Capítulo 6.

Capítulo 4

Fundamentos de Mensajería Programable

Este capítulo presenta a Government-Enterprise Ecosystem Gateway (G-EEG) como una realización concreta del concepto de Mensajería Programable – un paradigma para el intercambio automatizado de mensajes entre entidades colaborativas para responder a diversas necesidades de comunicación en entornos colaborativos complejos y dinámicos, tales como los entornos que caracterizan al Gobierno Integrado. A continuación de este capítulo, el Capítulo 5 describe la implementación de G-EEG, mientras que el Capítulo 6 realiza la evaluación de G-EEG con respecto al problema enunciado y al caso de estudio del Capítulo 1, los desafíos al Gobierno Integrado explicados en el Capítulo 2 y las soluciones relacionadas presentadas en el Capítulo 3.

La presentación de G-EEG comienza con el grupo de conceptos interrelacionados que en conjunto definen al Gateway, en el contexto del modelo de dominio para Gobierno Integrado de la Sección 2. El diagrama de clases conceptual resultante es usado para representar, formalizar e implementar G-EEG. A fin de visualizar las estructuras de comunicaciones basadas en G-EEG, la manera en la que permiten el intercambio de mensajes y cómo evolucionan a través de dicho intercambio, se introduce una notación gráfica. La misma es usada para explicar la arquitectura, comportamiento y extensiones de G-EEG. La arquitectura comprende un framework de mensajería de tiempo de ejecución, un repositorio de extensiones horizontales (independientes del proceso) y verticales (dependientes del proceso) habilitadas por encima del framework de ejecución y un framework de desarrollo para construir nuevas extensiones. El comportamiento de G-EEG es explicado usando tres escenarios de ejemplo para intercambio de mensajes: sobre un canal simple, sobre un canal extendido horizontalmente y sobre dos canales coordinados por una extensión vertical. El segundo escenario aplica dos extensiones – Validación y Trasformación, mientras que el tercero aplica la extensión de Orden. A continuación, se introduce un conjunto representativo de extensiones horizontales y verticales, incluyendo las estructuras de comunicación creadas y el comportamiento de la mensajería. Finalmente, el capítulo presenta la formalización de G-EEG, comenzando con XML como la estructura de datos subyacente para mensajes y variables; siguiendo con una familia de lenguajes XML para representar varios tipos de expresiones sobre XML, cada una con sintaxis y semántica formalmente definidas; y hasta la mensajería programable formalmente descripta con tales lenguajes XML y llevada a cabo por G-EEG. Una discusión sobre los resultados es realizada al final.

El capítulo está organizado de la manera siguiente. La Sección 4.1 explica los conceptos de G-EEG, seguidos por una notación gráfica en la Sección 4.2 para visualizar tales conceptos y las estructuras de comunicación de G-EEG construidas con tales conceptos. La Sección 4.3 introduce la arquitecturas de G-EEG, comprendiendo los componentes G-EEG-CORE, G-EEG-EXTEND y G-EEG-DEVELOP. La Sección 4.4 explica el comportamiento de G-EEG habilitado por esta estructura, cubriendo el establecimiento de estructuras de comunicación, intercambio simple de mensajes sobre tales estructuras e intercambio de mensajes mejorado basado en estructuras de comunicación extendidas (horizontalmente y verticalmente). La Sección 4.5 describe varias extensiones horizontales y verticales que pueden ser soportadas por G-EEG. La Sección 4.6 incluye la formalización de G-EEG en RSL – RAISE Specification Language, desde estructuras de datos de XML, a través de lenguajes XML sobre tales estructuras, hasta mensajería basada en XML expresada en tales lenguajes. La Sección 4.7 incluye una discusión.

4.1 Conceptos

G-EEG es una plataforma de comunicación de alto nivel que soporta actividades realizadas por redes de trabajo inter-organizacionales. Permite la construcción, aplicación y evolución de estructuras de comunicación complejas para intercambiar mensajes asincrónicamente en varios contextos de administración, por ejemplo entrega colaborativa de servicios públicos.

Esta sección presenta cuatro conceptos principales de G-EEG - Miembro, Canal, Mensaje y Extensión. G-EEG permite a Miembros registrados, que actúan en nombre de entidades externas como software, personas u organizaciones, enviar Mensajes entre sí por medio de Canales de comunicación definidos y suscriptos por los usuarios, con una rica funcionalidad de mensajería disponible a pedido a través de varias Extensiones horizontales (independientes del proceso) y verticales (dependientes del proceso).

Estos cuatro conceptos y otros relacionados a ellos son descriptos en la Figura 25 y explicados a continuación:

- *Miembro* – Un Miembro es un usuario registrado de G-EEG. Actuando en nombre de una entidad externa – software, persona o agencia, se le asigna un nombre, un identificador único y ciertos recursos de computación y comunicación y es capaz de usar esos recursos para crear, configurar y usar estructuras de comunicación de G-EEG. Se definen tres tipos de miembros: (1) Administrador – un miembro distinguido de G-EEG responsable de autorizar cualquier cambio en las estructuras de comunicación, tales como el registro de un nuevo miembro o creación de nuevos canales; (2) Visitante – un miembro que sólo puede solicitar al administrador el registro de un nuevo miembro; y (3) Usuario – un miembro regular que puede intercambiar mensajes con otros miembros por medio de canales definidos y suscriptos por usuarios, incluyendo mensajes para crear y destruir canales, suscribir o remover a/de canales y habilitar/deshabilitar extensiones. Mientras que los miembros visitante y administrador son predefinidos y deben estar siempre presentes en el sistema (no pueden ser removidos), otros miembros (usuarios) deben ser registrados explícitamente y pueden ser removidos más tarde.

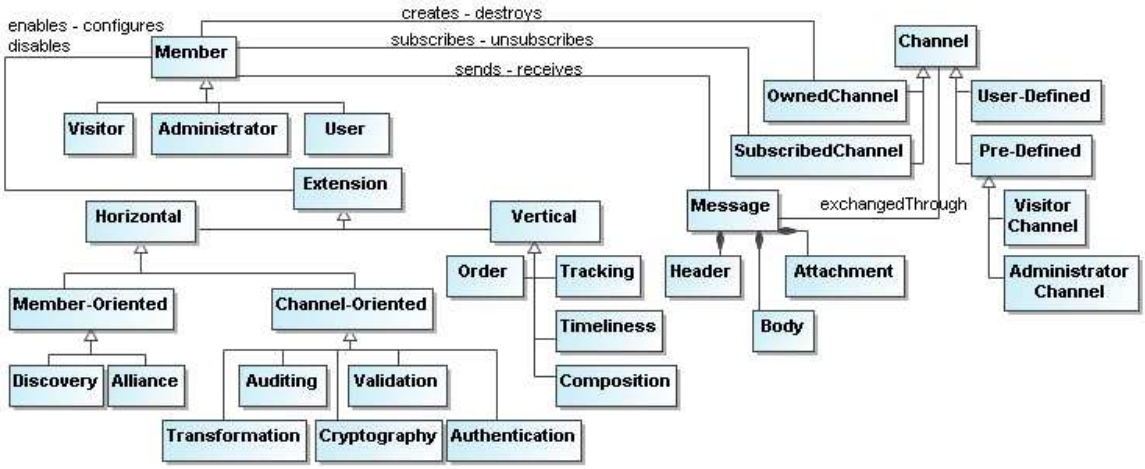
- *Canal* – Un Canal es un medio de comunicación entre miembros autorizados (suscriptores). Al permitir suscriptores múltiples, los canales soportan todo tipo de comunicaciones uno-a-uno, uno-a-muchos, muchos-a-uno y muchos-a-muchos. Cada canal tiene un nombre, un identificador único, un propietario – el miembro que creó el canal, y suscriptores – miembros autorizados a enviar y recibir mensajes por el canal. Los Canales pueden ser simples – carentes de estructura interna y capaces de transferir mensajes de un extremo a otro en una operación atómica simple, o complejos – obtenidos por la composición de canales existentes y realizando varias operaciones para transferir mensajes. Además, los canales pueden ser: (1) predefinidos – definidos por G-EEG con el propósito de permitir comunicaciones internas entre miembros, incluyendo al visitante y al administrador; o (2) definidos por el usuario – creados, suscriptos y usados por miembros para enviar y recibir mensajes. Un canal predefinido es creado automáticamente por G-EEG, es propiedad del administrador, suscripto por sólo un miembro y existe en tanto exista el suscriptor (no pueden ser destruidos por los propietarios). Por otro lado, un canal definido por el usuario, es creado dinámicamente por el administrador a requerimiento de su propietario, tiene un conjunto variable de miembros suscriptores autorizados por el propietario y puede ser destruido por el administrador a requerimiento de su propietario. En cuanto a las relaciones entre miembros y canales, los canales pueden ser propiedad de o suscriptos por miembros. En el primer caso, el miembro mantiene información acerca de todos los suscriptores y las posibles extensiones habilitadas para el canal. En el segundo caso, el miembro sólo mantiene la información relacionada al propietario del canal.

- *Mensaje* – Un Mensaje es una unidad de comunicación entre miembros, enviado por un canal para ser distribuido a un grupo particular de miembros. Todas las comunicaciones entre miembros, incluyendo los requerimientos al administrador para crear, modificar, y destruir estructuras de comunicación, se llevan a cabo por medio de mensajes y canales. Los mensajes de G-EEG son escritos usando XML y su estructura comprende un Encabezado, un Cuerpo y cero o varios Anexos. El Encabezado contiene un identificador, tipo y emisor del mensaje, dirección de todos los miembros receptores, y otra información de control. El tipo del mensaje determina su propósito y procesamiento en ambos extremos de emisión y recepción. Los tipos de mensaje estándar incluyen requerimientos para registrar o remover miembros, crear y destruir canales, suscribir y remover miembros a/de canales y otros. El Cuerpo del Mensaje tiene el contenido del Mensaje, el cual es de mayor interés para las aplicaciones que se comunican por medio de G-EEG que para G-EEG en sí mismo. La excepción son los mensajes estándar mencionados anteriormente, cuyo cuerpo es predeterminado por G-EEG. Por ejemplo, el cuerpo de un requerimiento de registro contiene el nombre del nuevo miembro, mientras que el cuerpo de la respuesta enviada por el administrador contiene el identificador asignado al mismo. Finalmente, los Anexos son documentos y datos voluminosos enviados junto con los mensajes.

- *Extensión* – Para proveer servicios de mensajería mejorados, G-EEG puede implementar y distribuir nuevas funcionalidades por medio de extensiones. Considerando el tipo de funcionalidad provista, las extensiones se clasifican en Horizontales (independientes del proceso) y Verticales (dependientes del proceso). Las extensiones Horizontales se basan en mensajes transitando por un canal simple, independientemente de cualquier proceso de

negocio dentro del cual tenga lugar el intercambio. Como ejemplo incluimos: Auditoría, Validación, Transformación y Criptografía de mensajes. Las extensiones Verticales coordinan el intercambio de mensajes en tránsito por varios canales como parte de un proceso de negocio concreto. Como ejemplo incluimos: Orden, Puntualidad y Seguimiento. Las extensiones Horizontales a su vez se clasifican en: orientadas a canal – proveen un servicio específico a todos los mensajes intercambiados a través de un canal dado, y orientadas al miembro – un conjunto de servicios relacionados con la extensión que es provisto por un miembro dedicado luego de recibir un requerimiento. Por ejemplo, Auditoría es una extensión orientada a canal – todos los mensajes intercambiados por un canal auditado son grabados en un log, mientras que Localización es una extensión orientada a miembro – un miembro puede solicitar servicios de localización enviando un mensaje al administrador de la extensión, después que la extensión es habilitada.

Figura 25: Modelo Conceptual de G-EEG



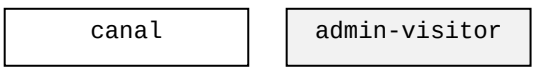
4.2 Notación

El objetivo de esta sección es introducir una notación gráfica simple para describir las estructuras de comunicación creadas por G-EEG, cómo esas estructuras soportan el intercambio de mensajes entre miembros y cómo evolucionan como resultado de ello. La notación permite la representación de los conceptos principales de G-EEG – miembros, canales, mensajes y extensiones, las relaciones entre ellos – propietario de y suscriptor a canal, mensajes enviados y recibidos por miembros a través de canales y las extensiones habilitadas y configuradas.

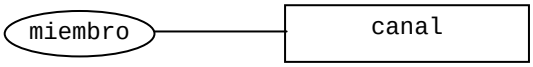
La notación se define de la siguiente manera:

1) Un miembro es descripto por una elipse con el nombre en su interior, con miembros pre-definidos como `admin` o `visitor` dibujados dentro de una elipse con relleno gris y los miembros definidos por los usuarios dento de una elipse sin relleno, por ejemplo los miembros `miembro` y `admin`:

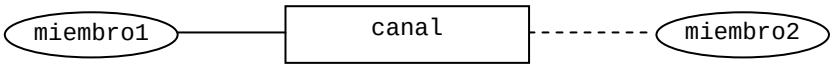
2) Un canal es descripto por un rectángulo con el nombre en su interior, con relleno gris para los canales de administración pre-definidos y sin relleno para los canales definidos por los usuarios, por ejemplo los canales `canal` y `admin-visitor`:



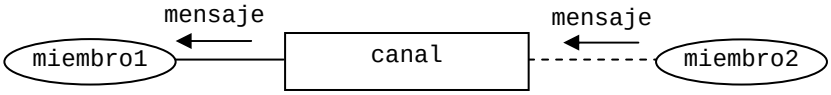
3) Una línea sólida conecta al canal con su propietario, por ejemplo. `miembro` es el propietario de `canal`:



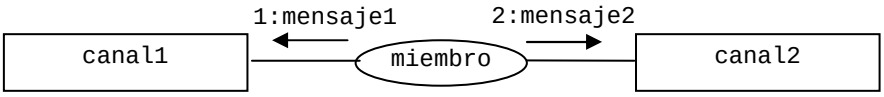
4) Una línea punteada conecta al canal con un suscriptor, por ejemplo `miembro1` es el propietario y `miembro2` el suscriptor de `canal`:



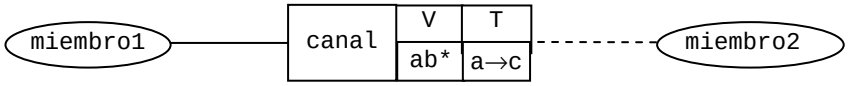
5) El acto de enviar o recibir un mensaje es representado por una flecha con nombre, dibujada sobre el canal. La flecha apunta del miembro al canal, si el mensaje es enviado por el miembro, o del canal al miembro, si el mensaje es recibido. Por ejemplo, en la figura debajo, `mensaje` es enviado por `miembro2` a `canal`, y recibido por `miembro1` desde `canal` (en general, el mensaje enviado y recibido puede o no ser el mismo):



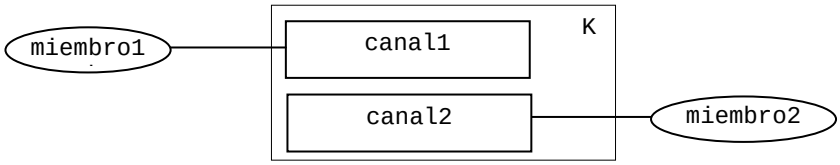
6) Cuando una serie de extensiones es enviada y recibida entre miembros, un número de secuencia es usado como prefijo en cada mensaje. El número determina el orden en el cual el mensaje es enviado o recibido. Por ejemplo, en la figura debajo `miembro` primero envía `mensaje1` al `canal1` y luego envía `mensaje2` a `canal2`:



7) La habilitación de extensiones horizontales en un canal se describe dividiendo el rectángulo del canal en columnas, la primera columna contiene el nombre del canal y las siguientes contienen el símbolo de la extensión en la celda superior y la configuración vigente de la extensión en la celda inferior. Por ejemplo, el canal mostrado debajo, tiene habilitada la extensión de validación (V) y configurada como `ab*` - los mensajes válidos deberían consistir de una letra a seguida por cualquier número de b's; y la extensión de transformación (T) habilitada y configurada como a→c – los mensajes en tránsito a través del canal son transformados reemplazando todas las ocurrencias de la letra a en letras c.

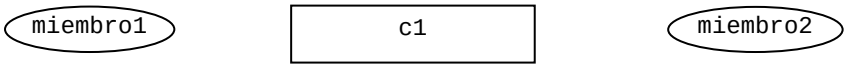


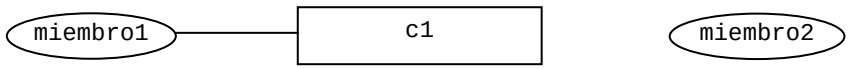
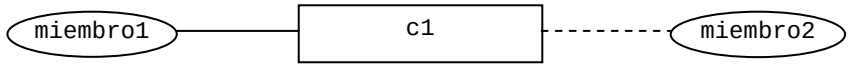
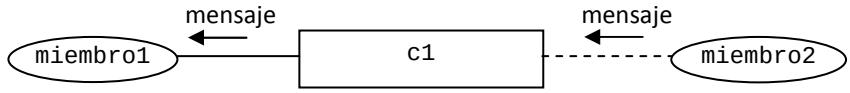
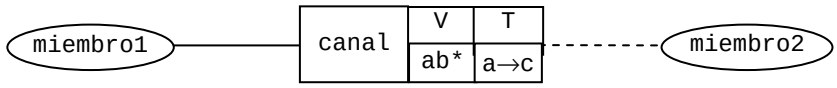
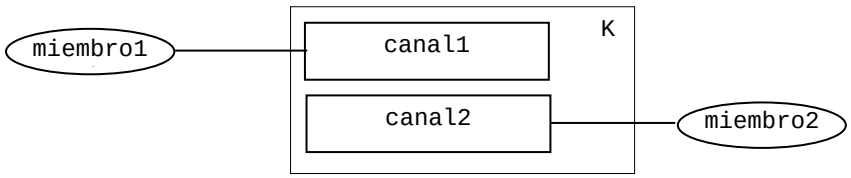
8) Finalmente, una extensión vertical se describe como un rectángulo que encapsula varios canales. El rectángulo es etiquetado con un carácter que identifica el tipo de la extensión. Por ejemplo, el tráfico de mensajes a través de `canal1` y `canal2` debajo es coordinado por la extensión de Seguimiento (K).



La notación es resumida en la Tabla 7.

Tabla 7: Notación Gráfica de G-EEG

Miembro	
Canal	

Propietario	
Suscriptor	
Mensaje	
Extensión Horizontal	
Extensión Vertical	

4.3 Arquitectura

La arquitectura de G-EEG está construida en base a un framework mínimo de tiempo de ejecución que provee servicios de mensajería básicos a sus miembros (G-EEG-CORE), un repositorio de extensiones para proveer funcionalidad de mensajería enriquecida por encima del framework básico (G-EEG-EXTEND) y un framework de desarrollo para construir nuevas extensions de manera rigurosa (G-EEG-DEVELOP). Estos componentes se describen en las siguientes tres secciones.

4.3.1 G-EEG-CORE

G-EEG-CORE es un framework de mensajería de tiempo de ejecución que permite el intercambio asincrónico de mensajes entre miembros registrados a través de canales creados y suscriptos dinámicamente. Usando G-EEG-CORE, los miembros pueden no solamente intercambiar mensajes, sino también armar, evolucionar y desarmar estructuras de comunicación complejas que sustentan y permiten el intercambio de mensajes. A este efecto, G-EEG-CORE ofrece nueve tipos de servicios:

1) Registrar un miembro,
2) Des-registrar un miembro,
3) Crear un cananl,
4) Destruir un canal,
5) Suscribir un miembro a un canal,
6) Des-suscribir un miembro de un canal,
7) Enviar un mensaje,
8) Recibir un mensaje y
9) Reenviar un mensaje entre miembros.

Todos estos servicios son ejecutados mediante el envío y la recepción de mensajes entre miembros autorizados: un visitante puede requerir al administrador registrar un nuevo miembro y una vez registrado, el miembro puede requerir al administrador des-registrarse; un miembro puede requerir al administrador crear un canal y una vez creado, el propietario puede requerir al administrador destruir el canal; un miembro puede requerir al propietario de un canal suscribirse al canal y una vez suscripto, puede requerir al propietario des-suscribirse; una vez suscripto a un canal, un miembro puede enviar un mensaje y recibir mensaje enviados por otros suscriptores a través del canal. Basándose en tres conceptos simples – miembro, canal y mensaje, G-EEG-CORE es el componente fundacional de G-EEG, con todas las otras funcionalidades de mensajería construidas sobre él.

4.3.2 G-EEG-EXTEND

G-EEG-EXTEND es un repositorio de extensiones listas para usar que proveen funcionalidad de mensajería mejorada requerida por las aplicaciones que se comunican, y un mecanismo para entregar, habilitar y configurar dinámicamente tales extensiones sobre las estructuras de comunicación creadas por G-EEG-CORE.

Las extensiones son clasificadas en horizontales y verticales:

- *Extensiones Horizontales* – Proveen la funcionalidad comúnmente requerida por las aplicaciones que se comunican, independiente de toda aplicación o contexto de proceso dentro del cual sucede el intercambio de mensajes, por ejemplo: auditoría, validación, transformación, criptografía de mensajes, autenticación de miembros, alianza y localización. Las extensiones horizontales son a su vez clasificadas en: orientadas a canales y orientadas a miembros. Asociadas a canales específicos, las extensiones orientadas a canal proveen funcionalidad sobre el flujo de mensajes pasando a través del canal, tal como: auditoría, validación, transformación, criptografía y autenticación. Por ejemplo, todos los mensajes enviados a través de un canal auditado son grabados en una base de datos y se hacen disponibles a todos los suscriptores del canal para una futura recuperación. Las extensiones orientadas a miembro ofrecen funcionalidad específica para miembros a través de un miembro dedicado, responsable por la extensión (administrador de la extensión). Los miembros pueden requerir esta funcionalidad emitiendo requerimientos al administrador de la extensión, quién subsiguientemente responde a tales requerimientos. Un ejemplo es la extensión de localización que permite a los miembros buscar información sobre otros miembros y canales a través del administrador de la extensión.

- *Extensiones Verticales* – Proveen la funcionalidad que depende del contexto de aplicación del intercambio de mensajes, por ejemplo la entrega de un servicio integrado. Tales extensiones son usualmente habilitadas sobre varios canales y tienen como objetivo coordinar la ejecución de un proceso de negocios. Ejemplos incluyen: orden – asegurando que el flujo de mensajes a través de los canales cumple con un proceso de negocios pre-definido; puntualidad – controlando la conformidad con una política que gobierna el intercambio de información dentro de un proceso de negocios; composición – construyendo canales complejos por medio de vincular, separar, redireccionar, etc. canales existentes; y seguimiento – determinando la ubicación de mensajes en tránsito a través de canales complejos.

El conjunto inicial de extensiones en G-EEG-EXTEND, como se describe en la Sección 4.5, incluye: (1) Auditoría, (2) Validación, (3) Transformación, (4) Critografía, (5) Autenticación, (6) Localización, (7) Alianza, (8) Orden, (9) Puntualidad, (10) Composición y (11) Seguimiento. El conjunto comprende extensiones horizontales, incluyendo extensiones orientadas a canales y a miembros, y extensiones verticales.

G-EEG-EXTEND administra tales extensiones y las hace disponibles a miembros ofreciendo cuatro tipos de servicios:

1) Entregar una extensión – un nuevo tipo de extensión se hace disponible a los miembros de G-EEG;
2) Habilitar una extensión – un extensión entregada se habilita en un canal, miembro o conjunto de canales;
3) Configurar una extensión – una extensión habilitada es configurada con un nuevo conjunto de parámetros; y
4) Deshabilitar una extensión – una extensión habilitada es removida del canal, miembro o conjunto de canales.

4.3.3 G-EEG-DEVELOP

Mientras que G-EEG-EXTEND ofrece un conjunto fijo de extensiones para que los miembros de G-EEG utilicen, ningún conjunto fijo de extensiones puede satisfacer las necesidades complejas y cambiantes de comunicación del tipo de entornos colaborativos característicos de Gobierno Integrado. A fin de resolver esto, más extensiones pueden ser construidas a través del framework de G-EEG-DEVELOP y se pueden hacer disponibles a los miembros a través del servicio de entrega de G-EEG-EXTEND.

G-EEG-DEVELOP es un framework de desarrollo que permite la especificación rigurosa, el diseño y la verificación de extensiones de mensajería, todo basado en los servicios de mensajería básicos provistos por G-EEG-CORE. Una vez desarrolladas, tales extensiones se convierten en parte de G-EEG-EXTEND. En esta etapa, G-EEG-DEVELOP comprende:

1) Un modelo formal de las estructuras de datos XML que capturan los contenidos de las variables poseídas por los miembros – como variables que representan el inbox y outbox de los miembros, parte del estado local de los miembros, y los mensajes intercambiados por ellos – tal intercambio produce que los estados locales y globales de G-EEG cambien;

2) Una familia de lenguajes XML interrelacionados, todos con sintaxis y semántica formal asignada, para capturar los varios tipos de expresiones – Booleanas, numéricas, de textos, nodos, listas de nodos, árboles, etc. sobre las estructuras de XML subyacentes, incluyendo expresiones Booleanas para capturar propiedades lógicas o expresiones de árbol para representar varias formas en las cuales las estructuras pueden cambiar, por ejemplo la recepción de un mensaje en el outbox de un miembro; y

3) Un modelo para representar las estructuras y operaciones de G-EEG-CORE y G-EEG-EXTEND, incluyendo las descripciones formales de los nueve servicios de mensajería básicos, cuatro servicios para manejar extensiones de mensajería, y un conjunto inicial de once extensiones horizontales y verticales, todas escritas como expresiones en la familia de lenguaje XML. El método permite especificar extensiones a nivel abstracto (a ser) y concreto (como es). Nuevas extensiones pueden ser especificadas e implementadas en la misma forma como los servicios de mensajería básicos y las once extensiones de ejemplo.

Basándose en el modelo formal subyacente y en una familia de lenguajes definidos formalmente, G-EEG-DEVELOP hace posible especificar las extensiones de mensajería a diferentes niveles de abstracción. En el nivel abstracto, el modelo permite especificar el modelo de comportamiento esperado de una extensión de mensajería – qué efectos observables debería producir. A nivel intermedio, permite diseñar estructuras de comunicación que son requeridas para permitir tales efectos. A nivel más bajo, permite describir el comportamiento de mensajería concreto de la extensión, que se ejecuta a través de las estructuras de comunicación diseñadas, para producir los resultados esperados. Como el modelo formal abarca todas las estructuras de datos y lenguajes usados, es posible relacionar descripciones expresadas en diferentes niveles de abstracción, para verificar que las descripciones de menor-nivel correctamente implementan las de mayor-nivel. Sin embargo, en el estado actual de G-EEG-DEVELOP, esta posibilidad aún no se ha explorado.

4.4 Comportamiento

Siguiendo los conceptos de G-EEG explicados en la Sección 4.1, la notación gráfica introducida en la Sección 4.2 y la arquitectura presentada en la Sección 4.3, esta sección describe el comportamiento de G-EEG – cómo se registran los miembros; cómo se crean, modifican y destruyen las estructuras que conectan a los miembros y cómo se utilizan tales estructuras para el intercambio de mensajes entre miembros. Se explica por separado el comportamiento básico de mensajería (Sección 4.4.1) y la mensajería extendida usando extensiones horizontales (Sección 4.4.2) y verticales (Sección 4.4.3). El comportamiento es ilustrado por medio de la notación gráfica mediante una secuencia de diagramas de comunicación que muestran paso a paso de qué manera las estructuras de comunicación permiten el intercambio de mensajes entre miembros y cómo evolucionan como resultado de dicho intercambio, proveyendo bloques de construcción básicos para expresar comportamientos de mensajería de complejidad creciente.

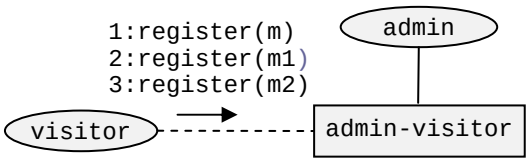
4.4.1 Servicios de Mensajería Básicos

G-EEG tiene dos miembros pre-definidos – administrador (`admin`) y visitante (`visitor`), y un número arbitrario de miembros definidos por el usuario. El administrador lleva a cabo las funciones administrativas tales como registro de miembros, creación de canales, o entrega de extensiones. Posicionado únicamente en G-EEG, `admin` mantiene conexiones uno-a-uno con cada miembro `m` por medio de la familia de canales de administración `admin-m`, con `admin` como propietario y `m` como único suscriptor. Tales canales son creados automáticamente como parte de la registración del miembro y existen en tanto exista el miembro. Para registrar un nuevo miembro `m`, el miembro `visitor` puede ser usado para enviar el mensaje de requerimiento de registro `register(m)` a `admin` usando el canal `admin-visitor`; la registración es el único servicio que ofrece `visitor`.

La Figura 26 debajo ilustra la configuración inicial de G-EEG, con los miembros `admin` y `visitor` conectados por medio del canal `admin-visitor`. El canal es propiedad de `admin` y es suscripto por `visitor`. Supongamos que

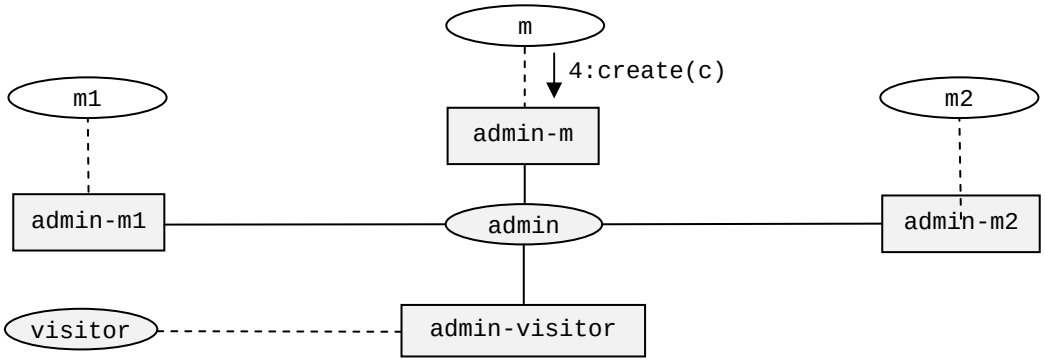
visitor requiere el registro de tres nuevos miembros m, m1 y m2 enviando los mensajes 1:register(m), 2:register(m1) y 3:register(m2) a admin por medio del canal admin-visitor.

Figura 26: Mensajería Básica – Registro de Miembros



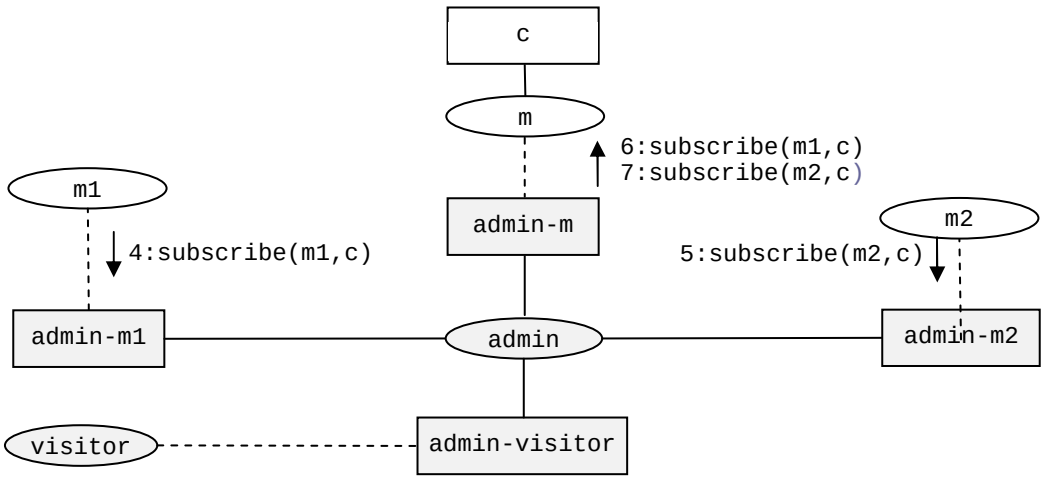
Como resultado, admin registra los tres miembros m, m1 y m2 y crea los canales de administración admin-m, admin-m1 y admin-m2 para establecer conexiones uno-a-uno con ellos (Figura 27). Como paso siguiente supongamos que el nuevo miembro registrado m requiere la creación de un nuevo canal c enviando el requerimiento 4:create(c) a admin.

Figura 27: Mensajería Básica – Creación de Canal



Como respuesta, se crea c con m como propietario. Siguiendo, supongamos que m1 y m2 quieren suscribirse a c. Para este fin, se envían los requerimientos 4:subscribe(m1,c) y 5:subscribe(m2,c) al propietario de m reenviándolos primeramente a admin a través de los canales admin-m1 y admin-m2. Al recibir los mensajes, admin los reenvía a m por medio del canal admin-m - 6:subscribe(m1,c) y 7:subscribe(m2,c). Ver Figura 28.

Figura 28: Mensajería Básica – Subscripción de Miembros a Canal



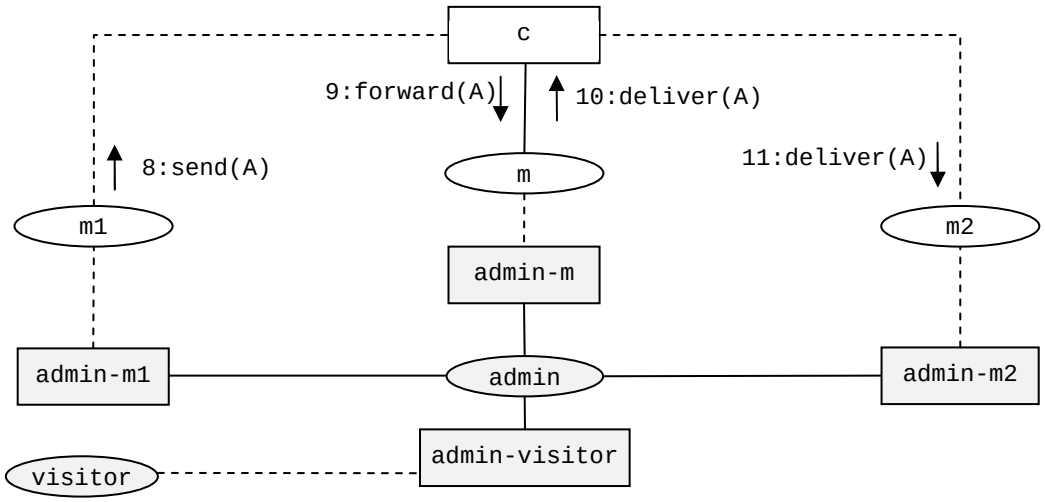
Como resultado de esto c tiene a m como propietario y a m1 y m2 como suscriptores. Luego supongamos que el suscriptor m1 envía un mensaje A al canal c – 8:send(A). Supongamos que el mensaje está escrito en XML como se muestra en XML 1.

XML 1: Mensaje Enviado por Miembro Suscripto a un Canal

```
<?xml version="1.0"?>
<message>
  <to>c</to>
  <from>m1</from>
  <subject>prueba</subject>
  <body>mensaje del suscriptor</body>
</message>
```

El mensaje es reenviado primero al propietario del canal, m – 9:forward(A), y luego distribuido a todos los suscriptores – 10:deliver(A) excepto al emisor y al propietario. Como resultado, es enviado a m2 – 11:deliver(A). Ver Figura 29.

Figura 29: Mensajería Básica – Envío de Mensaje como Suscriptor de Canal



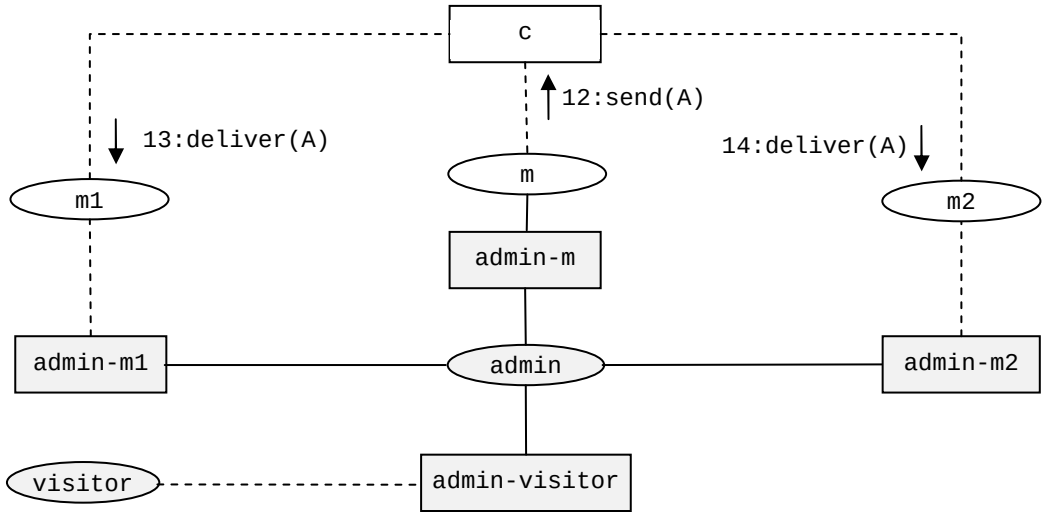
Supongamos ahora que el propietario m quiere enviar por sí mismo un mensaje B al canal c – 12:send(B). El mensaje sería como se muestra en XML 2.

XML 2: Mensaje Enviado por Miembro Propietario de un Canal

```
<?xml version="1.0"?>
<message>
  <to>c</to>
  <from>m</from>
  <subject>prueba</subject>
  <body>mensaje del propietario</body>
</message>
```

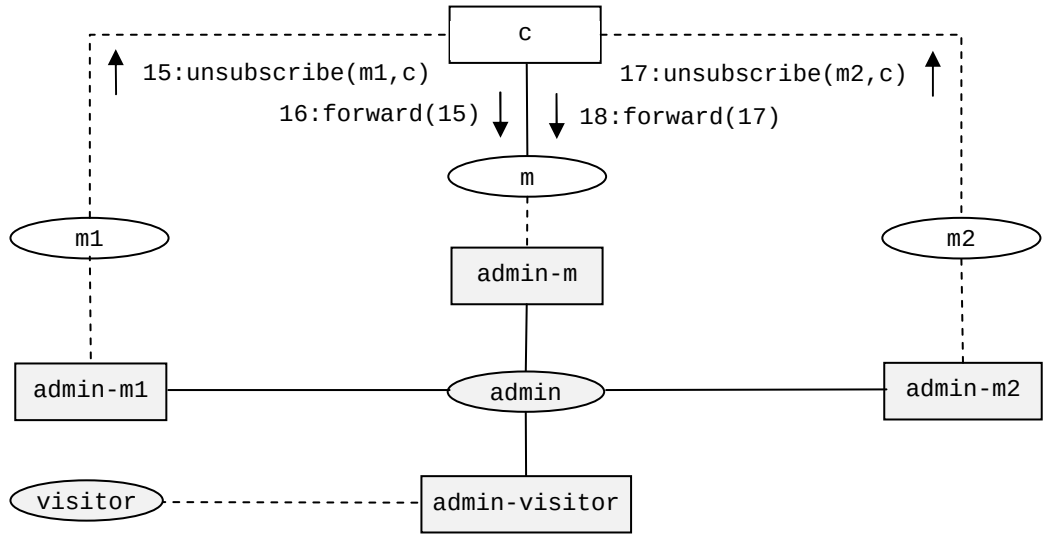
Esta vez, el procedimiento se ve simplificado ya que el mensaje es enviado a todos los suscriptores de c excepto el emisor. Como resultado, m1 recibe el mensaje – 13:deliver(B) y m2 recibe el mismo mensaje – 14:deliver(A). Ver Figura 30.

Figura 30: Mensajería Básica – Envío de Mensaje como Propietario de Canal



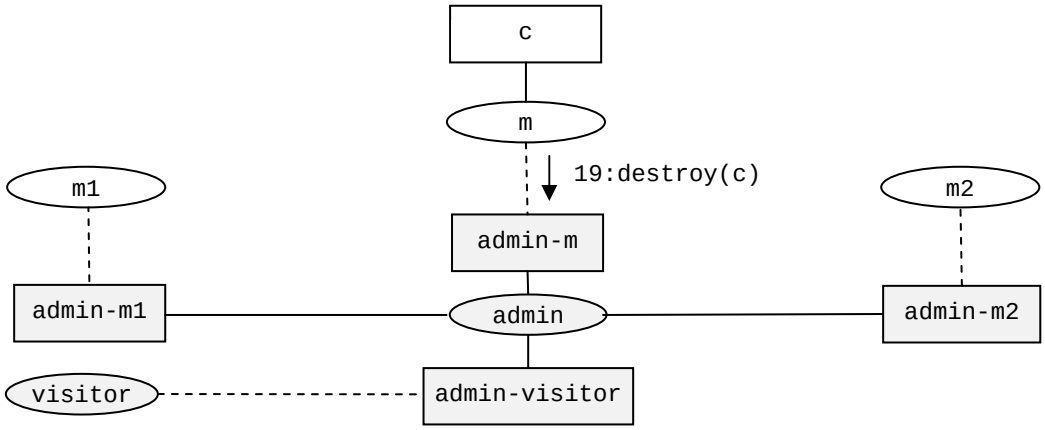
Una vez que ha tenido lugar el intercambio de mensajes, los miembros podrían necesitar modificar las estructuras de comunicación, por ejemplo des-suscribiéndose de los canales. Supongamos que `m1` y `m2` quieren des-suscribirse del canal `c`. Esto se lleva a cabo enviando los requerimientos `15:unsubscribe(m1,c)` y `17:unsubscribe(m2,c)` al canal `c`. Así como cualquier otro mensaje regular enviado por un suscriptor, ambos son reenviados al propietario – `16:forward(15)` y `18:forward(17)`; se usa una abreviatura para referirse al mensaje a través de su número de secuencia. Ver Figura 31.

Figura 31: Mensajería Básica – Des-suscripción de Miembro a Canal



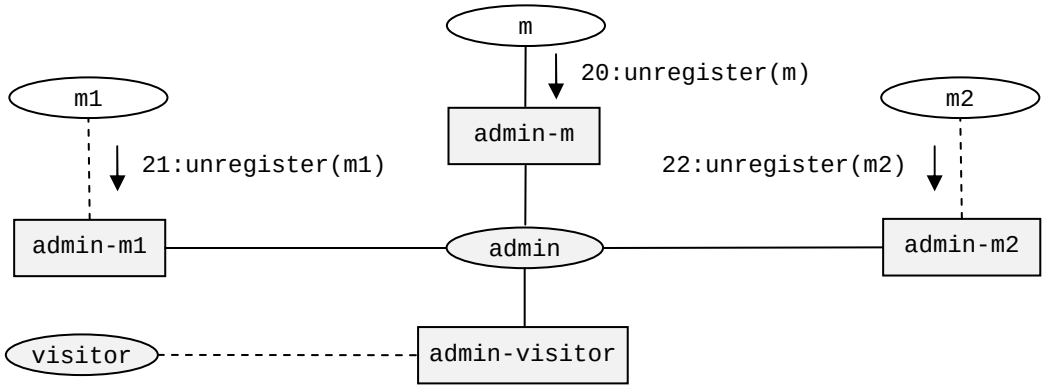
Sin embargo, al recibir ambos mensajes, el propietario no los envía a los suscriptores sino que remueve a `m1` y `m2` como suscriptores del canal. La estructura resultante se muestra en Figura 32. Como paso siguiente, supongamos que `m` desea destruir `c`. Para este fin, se envía un requerimiento `19:destroy(c)` a `admin` por medio del canal `admin-m`.

Figura 32: Mensajería Básica – Destrucción de Canal



Una vez que el mensaje es recibido, `admin` remueve el canal `c`, produciendo la estructura de comunicación mostrada en la Figura 33. Finalmente, supongamos que todos los miembros `m`, `m1` y `m2` desean eliminar sus registros. Esto es llevado a cabo por cada uno enviando un requerimiento de remoción a `admin` – `20:unregister(m)`, `21:unregister(m1)` y `22:unregister(m2)` por medio del canal de administración correspondiente (Figura 33).

Figura 33: Mensajería Básica – Remoción de Miembros



Al recibir los requerimientos, `admin` remueve los miembros y los canales de administración por los cuales se conectaba a ellos. La estructura de comunicación resultante muestra el estado inicial de G-EEG, mostrado en la Figura 34, con `admin` y `visitor` conectados a través del canal `admin-visitor`.

Figura 34: Mensajería Básica – Retorno al Estado Inicial

4.4.2 Extensiones Horizontales

El comportamiento de la mensajería descripto en la Sección 4.4.1 provee la funcionalidad básica para el framework de mensajería subyacente, con el cual se pueden construir comportamientos de mensajería más complejos. Patrones típicos de comportamientos complejos son provistos en G-EEG como extensiones, con G-EEG-EXTEND proveyendo un repositorio de extensiones horizontales y verticales más un mecanismo para habilitarlas y configurarlas

dinámicamente por encima de G-EEG-CORE. Esta sección se enfoca en las extensiones horizontales, que proveen servicios de mensajería independientes del contexto (procesos) usado por los miembros para el intercambio de mensajes. Ejemplos típicos son validación y transformación de los mensajes que circulan por los canales. Ambas extensiones serán usadas en esta sección para explicar el comportamiento de las extensiones horizontales.

Considere la estructura descripta en la Figura 35, con tres miembros `m`, `m1` y `m2`, y un canal `c` propiedad de `m` y suscripto por `m1` y `m2`. La estructura contiene también los miembros predefinidos `admin` y `visitor`, y los canales predefinidos que conectan a todos los miembros con `admin`: `admin-m`, `admin-m1`, `admin-m2` y `admin-visitor`. Supongamos que queremos entregar la extensión de validación, que verifica el formato de los mensajes en tránsito a través de canales, de tal modo que esté disponible para todos los miembros. Para tal fin, `visitor` solicita la registración del miembro `validate` que implementa la extensión de validación, enviando el mensaje de registro `1:register(validate)` a `admin`. El resultado se muestra en la Figura 36.

Figura 35: Mensajería Extendida – Entrega de la Extensión de Validación

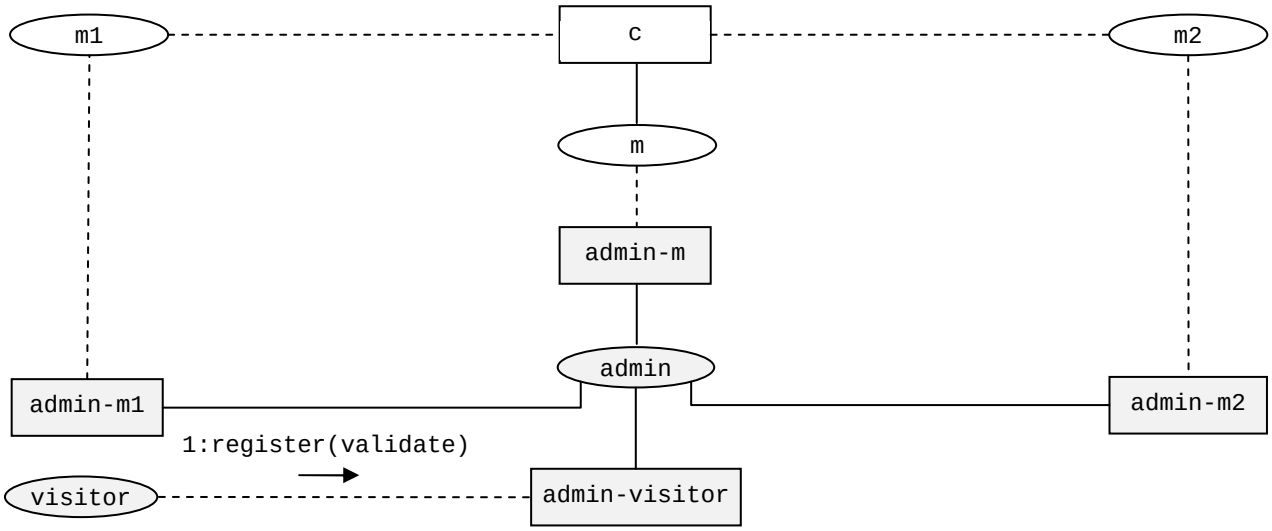
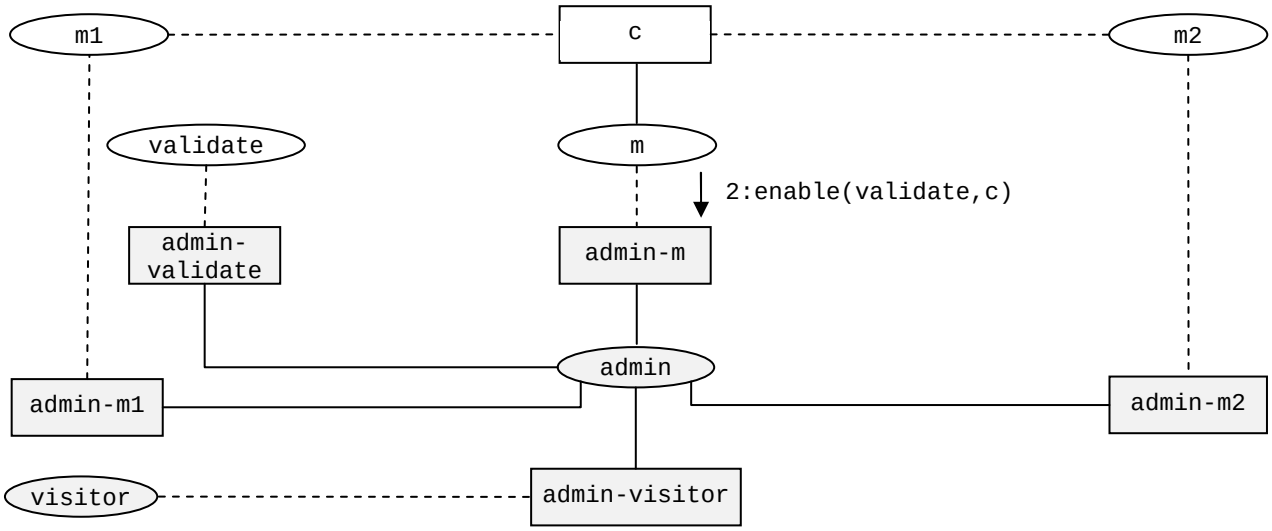


Figura 36: Mensajería Extendida – Habilitación de la Extensión de Validación en un Canal



La Figura 36 describe el nuevo miembro registrado `validate` y el canal `admin-validate` creado para conectar este miembro con `admin`. De esta manera, validación está disponible para todos los propietarios de canal en G-EEG.

Usualmente, se trata de una operación de única vez para el Gateway, a menos que algunos miembros nuevos asuman las responsabilidades de validación en el futuro.

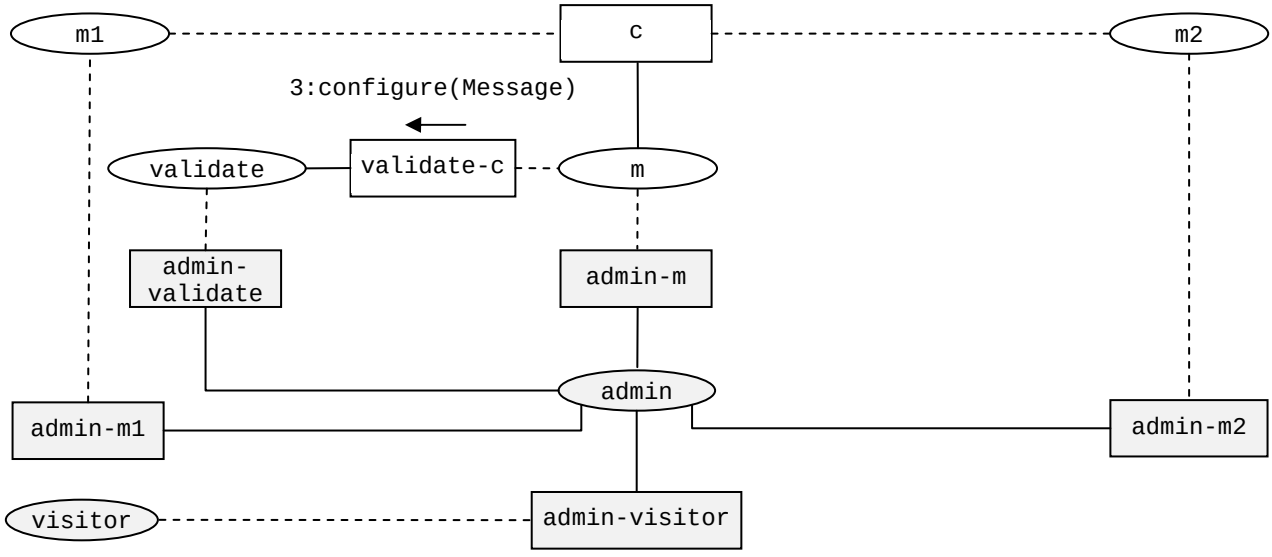
A continuación, supongamos que el propietario `m` desea habilitar la extensión de validación sobre el canal c. Para ello, `m` envía la solicitud a `admin` para habilitar la extensión de validación en `c – 2:enable(validate,c)`. El efecto, mostrado en la Figura 37, involucra a los miembros `validate` creando el canal `validate-c` y `m` suscribiéndose a dicho canal. Con el tiempo, `validate` podría poseer varios canales `validate-c` para llevar a cabo la validación sobre diferentes canales c. Sin embargo, antes de que la extensión esté operativa en c, debe ser configurada para determinar el formato válido de mensaje en tránsito por c. Supongamos que el formato es determinado por el siguiente esquema XML:

XML 3: XML Schema para Configurar la Extensión de Validación

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="message" type="Message">
  <xsd:complexType name="Message">
    <xsd:sequence>
      <xsd:element name="to" type="Channel"/>
      <xsd:element name="from" type="Member"/>
      <xsd:element name="subject" type="xsd:string"/>
      <xsd:element name="body" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  …
</xsd:schema>
```

El esquema requiere que cada mensaje válido sea una instancia del tipo complejo `Message`, que incluye: el elemento raíz `message` y cuatro elementos - `to` de tipo `Channel`, `from` de tipo `Member`, `subject` de tipo `string` y `body` de tipo `string`, donde `Channel` y `Member` son tipos simples definidos en otro lugar también como `string`.

Figura 37: Mensajería Extendida – Configuración de la Extensión de Validación en un Canal



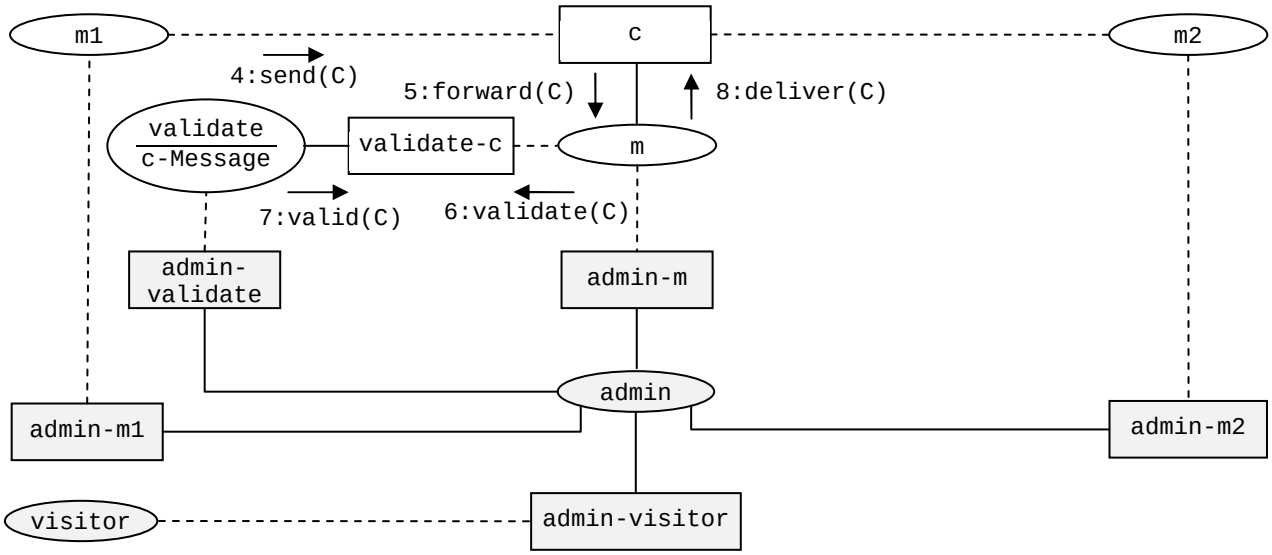
El resultado de la configuración es mostrado en la Figura 38. El diagrama muestra cómo el miembro `valídate` conecta al canal c con tipo `Message`. El canal está listo ahora para enviar mensajes validados. Supongamos que el suscriptor `m1` quiere enviar el mensaje C, descripto en XML 4, al canal c.

XML 4: Mensaje Válido Enviado a través de un Canal con Extensión de Validación

```
<?xml version="1.0"?>
<message>
  <to>c</to>
  <from>m1</from>
  <subject>prueba</subject>
  <body>mensaje simple</body>
</message>
```

Como C es claramente válido con respecto a Message, se produce el siguiente intercambio de mensajes: m1 envía C al canal c – 4:send(C); el mensaje es reenviado al propietario del canal m – 5:forward(C); reconociendo que el canal tiene la extensión de validación habilitada, m envía C para su validación a validate – 6:validate(C); como C es válido con respecto al tipo Message actualmente configurado por c, validate envía una respuesta positiva a m – 7:valid(C); sobre la respuesta positiva, m envía el mensaje al suscriptor restante m2 – 8:deliver(C).

Figura 38: Mensajería Extendida – Envío de un Mensaje Válido al Canal con Extensión de Validación



Supongamos que el suscriptor m1 envía un mensaje inválido D:

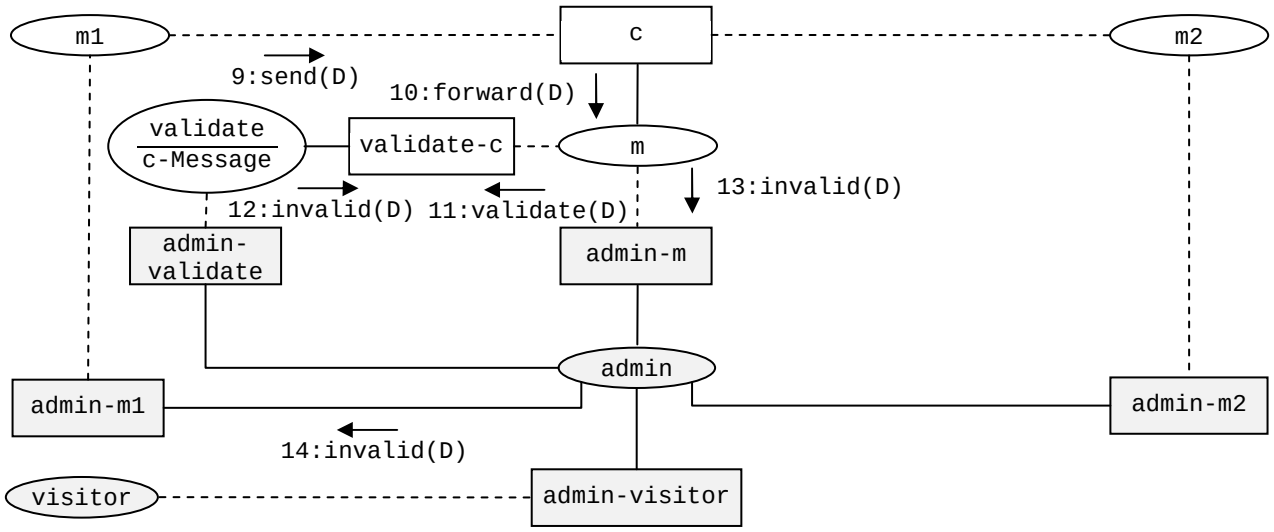
XML 5: Mensaje Inválido Enviado a través de un Canal con Extensión de Validación

```
<?xml version="1.0"?>
<message to="c" from="m1">
  <subject>prueba</subject>
  <body>mensaje del suscriptor</body>
</message>
```

En este caso se produce el siguiente intercambio de mensajes: m1 envía D al canal c – 9:send(D); D es reenviado al propietario m – 10:forward(D); m envía D para su validación a validate – 11:validate(D); validate envía una respuesta negativa a m – 12:invalid(D); y m envía la notificación del error al emisor por medio de admin - 13:invalid(D) y 14:invalid(D). El intercambio de mensajes se describe en la Figura 39.

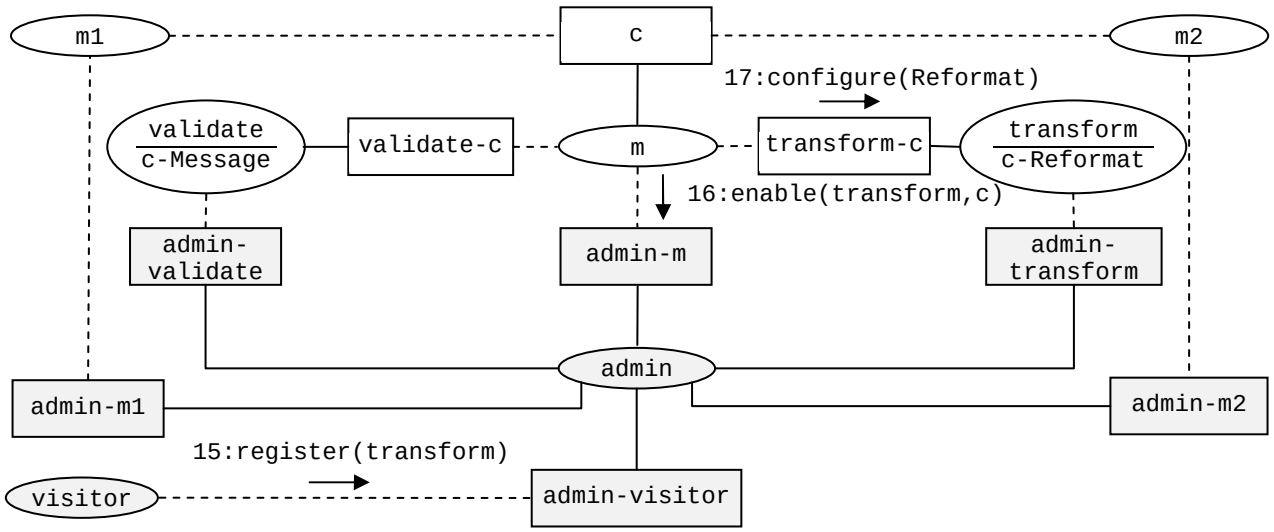
Después de haber explicado el comportamiento de la extensión de validación, supongamos que m quisiera habilitar también la extensión de transformación en c. Dicha extensión se ocupa de transformar los mensajes que circulan por el canal de un formato a otro. Así como para validación, se requiere entregar, habilitar y configurar esta nueva extensión.

Figura 39: Mensajería Extendida – Envío de un Mensaje Inválido al Canal con Extensión de Validación



Considere el escenario descripto en la Figura 40: `visitor` solicita a `admin` el registro del miembro `transform` que implementa la extensión de transformación – `15:register(transform)`; `admin` responde registrando `transform` y creando el canal `admin-transform` para conectarse a él; el propietario `m` solicita a `admin` la habilitación de la extensión en el canal `c` – `16:enable(transform,c)`; `admin` responde creando el canal `transform-c` con `transform` como propietario y `m` como suscriptor; y finalmente `m` configura la extensión de transformación en `c` de acuerdo a la plantilla `Reformat`, explicada a continuación – `17:configure(Reformat)`.

Figura 40: Mensajería Extendida – Entrega, Habilitación y Configuración de la Extensión de Transformación



`Reformat` es una plantilla escrita en lenguaje XSLT que da nuevo formato a los mensajes de tipo `Message`, tales como `C`, reemplazando los elementos `to` y `from` con los atributos `to` y `from` extraídos de `D`. Ver debajo.

XML 6: Plantilla XSLT para Transformar Mensajes

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template name="Reformat" match="message">
    <xsl:attribute name="to"><xsl:value-of select="to"/></xsl:attribute>
    <xsl:attribute name="from"><xsl:value-of select="from"/></xsl:attribute>
```

```
    <xsl:element name="subject"><xsl:value-of select="subject"/></xsl:element>
    <xsl:element name="body"><xsl:value-of select="body"/></xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

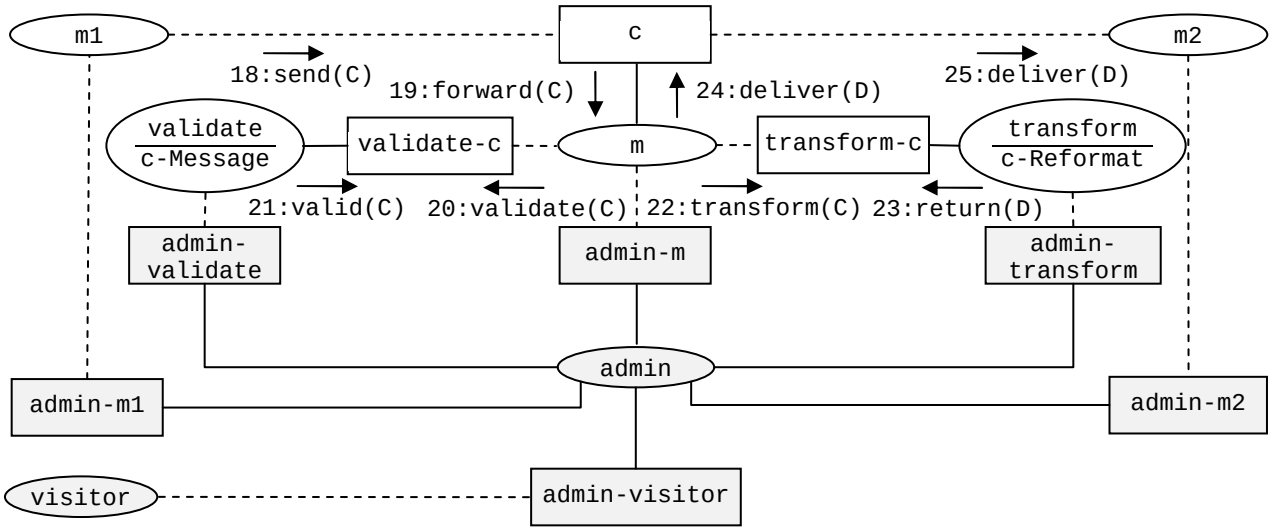
Con ambas extensiones habilitadas y configuradas en c, supongamos que m1 envía un mensaje C (mostrado en XML 7 a la izquierda) a c – 18:send(C), como se muestra en la Figura 41. El mensaje es reenviado al propietario m – 19:forward(C). Al reconocer que c tiene las extensiones de validación y transformación habilitadas (en ese orden), m reenvía primero a C al miembro de validación valídate para ser validado – 20:validate(C). En respuesta, validate envía una respuesta positiva a m – 21:valid(C). Dada la respuesta positiva, m envía el mensaje validado a transform – 22:transform(C). Como retorno, transform envía el mensaje transformado D (mostrado en XML 7 a la derecha) de vuelta a m – 23:return(D). Después de procesadas ambas extensiones, m está listo para distribuir el mensaje resultante D al suscriptor restante m2 – 24:deliver(D) y 25:deliver(D).

XML 7: Transformación de Mensajes

```
<?xml version="1.0"?>
<message>
  <to>c</to>
  <from>m1</from>
  <subject>test</subject>
  <body>simple message</body>
</message>
```

```
<?xml version="1.0"?>
<message to="c" from="m1">
  <subject>test</subject>
  <body>simple message</body>
</message>
```

Figura 41: Mensajería Extendida – Envío de Mensaje Válido al Canal con Extensión de Validación y Transformación



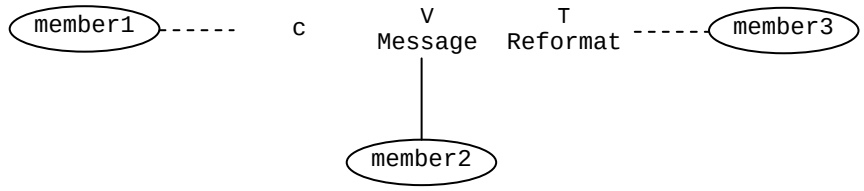
Se ha mostrado arriba cómo definir extensiones horizontales tomando como ejemplo validación y transformación. Además de las dos extensiones mencionadas previamente, la tesis define las siguientes extensiones horizontales en G-EEG-EXTEND:

- *Auditoría* – Mantiene un registro de los mensajes enviados a través de un canal, permitiendo así que mensajes históricos sean recuperados más tarde por el propietario del canal.
- *Validación* – Valida los mensajes enviados a través de un canal con respecto a un formato de mensaje dado. El formato se define al configurar la extensión, permitiendo reconfiguraciones posteriores en cualquier momento. Sólo los mensajes válidos son reenviados a los suscriptores. El emisor de un mensaje inválido es notificado por el propietario a través de admin.
- *Transformación* – Transforma los mensajes enviados a través de un canal de un formato a otro. El tipo de transformación es determinado durante la configuración y puede ser cambiado más tarde en tiempo de ejecución.

- *Criptografía* – Encripta o desencripta mensajes enviados a través de un canal. Encripción y desencripción simétricas son típicamente llevadas a cabo por medio de diferentes canales combinados en un proceso mayor.
- *Autenticación* – Asegura que solo aquellos miembros que puedan proveer alguna muestra sobre la prueba de su identidad son habilitados para enviar y recibir mensajes a través de un canal.
- *Localización* – Permite la búsqueda de miembros o canales por medio de las propiedades declaradas durante el registro o creación.
- *Alianza* – Crea un grupo de miembros que juntos llevan a cabo mensajería sobre ciertos canales, manejados por el coordinador de la alianza. Los miembros pueden pertenecer a varias alianzas y también intercambiar mensajes individualmente.

Cuando se habilitan extensiones orientadas a canal, se introduce una notación abreviada para abstraerse de las formas complejas en las que se comportan dichas extensiones. La Figura 42 debajo muestra cómo se representa un canal c con las extensiones de validación (V) y transformación (T) configuradas de acuerdo al esquema de XML tipo `Message` y la plantilla XSLT `Reformat`. Notar que los miembros `validate` y `transform` así como los canales `validate-c` y `transform-c` no son mostrados. El orden en el que las extensiones son mencionadas es relevante.

Figura 42: Mensajería Extendida – Abstracción de Extensiones Horizontales



4.4.3 Extensiones Verticales

Las extensiones horizontales son independientes del proceso. Esto significa que desconocen el contexto de aplicación para el intercambio de mensajes, usualmente proveyendo servicios sobre un canal simple o un miembro único. En contraste, las extensiones verticales dependen del contexto de aplicación y típicamente coordinan el intercambio de mensajes entre varios canales.

Consideremos el ejemplo descripto en la Figura 43 con miembros `m1`, `m2`, `m3`, `m4` y `m5`, y canales `c1` y `c2`. Supongamos que `m1` se suscribe a `c1`, el cual es propiedad de `m4`, `m3` se suscribe a `c2`, el cual es propiedad de `m5`, y `m2` se suscribe a ambos canales. Los canales `admin-m1`, `admin-m2` y `admin-m3` se omiten por simplicidad. Se desea habilitar la extensión de Orden en los canales `c1` y `c2` para coordinar el intercambio de mensajes sobre ellos. A tal fin, así como para las extensiones horizontales, la primera acción es entregar la extensión. Esto es realizado por `visitor` quien solicita a `admin` el registro del miembro `order` que implementa esta extensión – `1:register(order)`. Una vez que la extension de orden existe en el sistema, supongamos que los propietarios de canal `m4` y `m5` en conjunto solicitan a `orden`, vía `admin`, habilitar la extensión de orden en sus canales – `2:enable(c1,c2)` y `3:enable(c1,c2)`. En respuesta, `orden` crea el canal `orden-c1-c2` y `m4` y `m5` se suscriben al mismo.

El próximo paso es configurar la extensión. A tal fin, ambos propietarios de canal solicitan a `order` a través del canal `order-c1-c2` configurar la extensión – `4:configure((123)*)` y `5:configure((123)*)`; el mismo parámetro debe ser usado en ambas solicitudes. En este caso, la expresión `(123)*` determina el orden en el cual los mensajes pueden ser enviados a los canales `c1` y `c2`: primero `m1` envía un mensaje a `c1`, luego `m2` envía un mensaje a `c1` o `c2`, luego `m3` envía un mensaje a `c2`, nuevamente `m1` envía un mensaje a `c1`, etc. Notar que el envío es coordinado a través de diferentes canales. Luego de recibir ambas solicitudes, `order` es configurado para asegurar el orden `(123)*` en el cual los mensajes son enviados por los miembros `m1`, `m2` y `m3` a los canales `c1` y `c2`. El resultado se muestra en la Figura 43.

Figura 43: Mensajería Extendida – Entrega y Habilitación de la Extensión de Orden

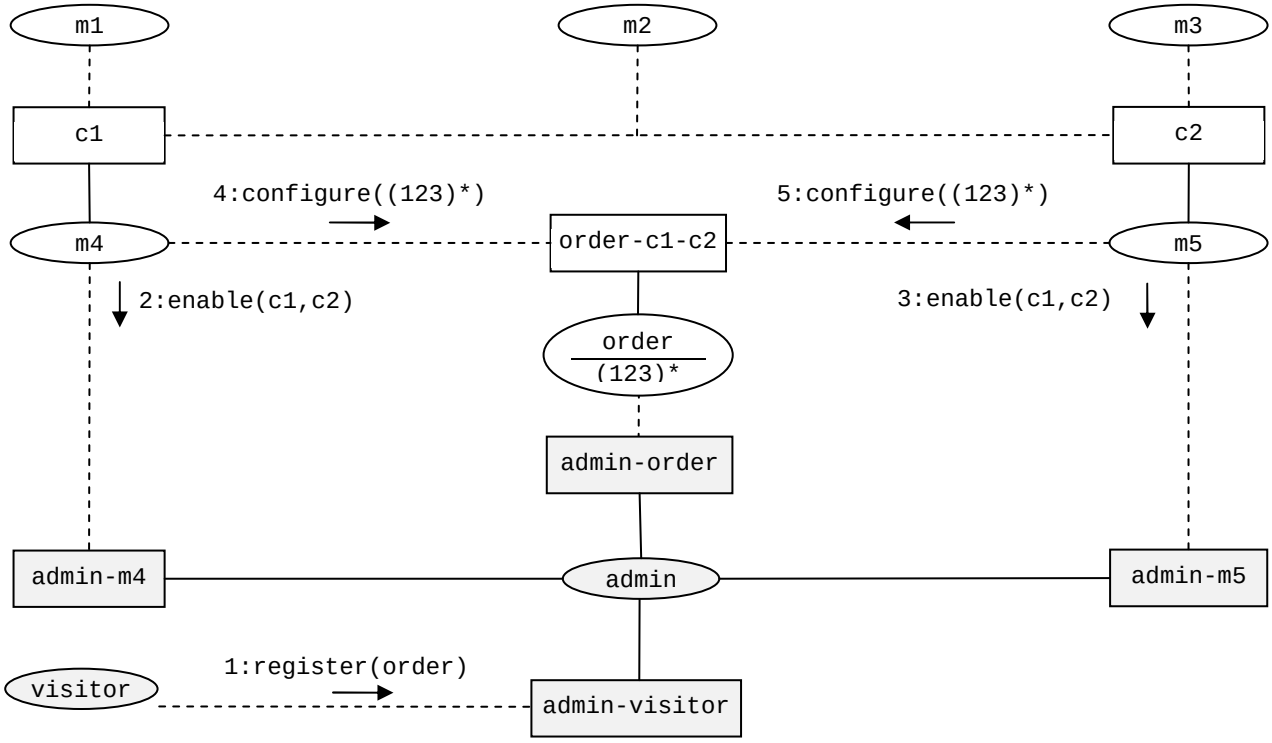
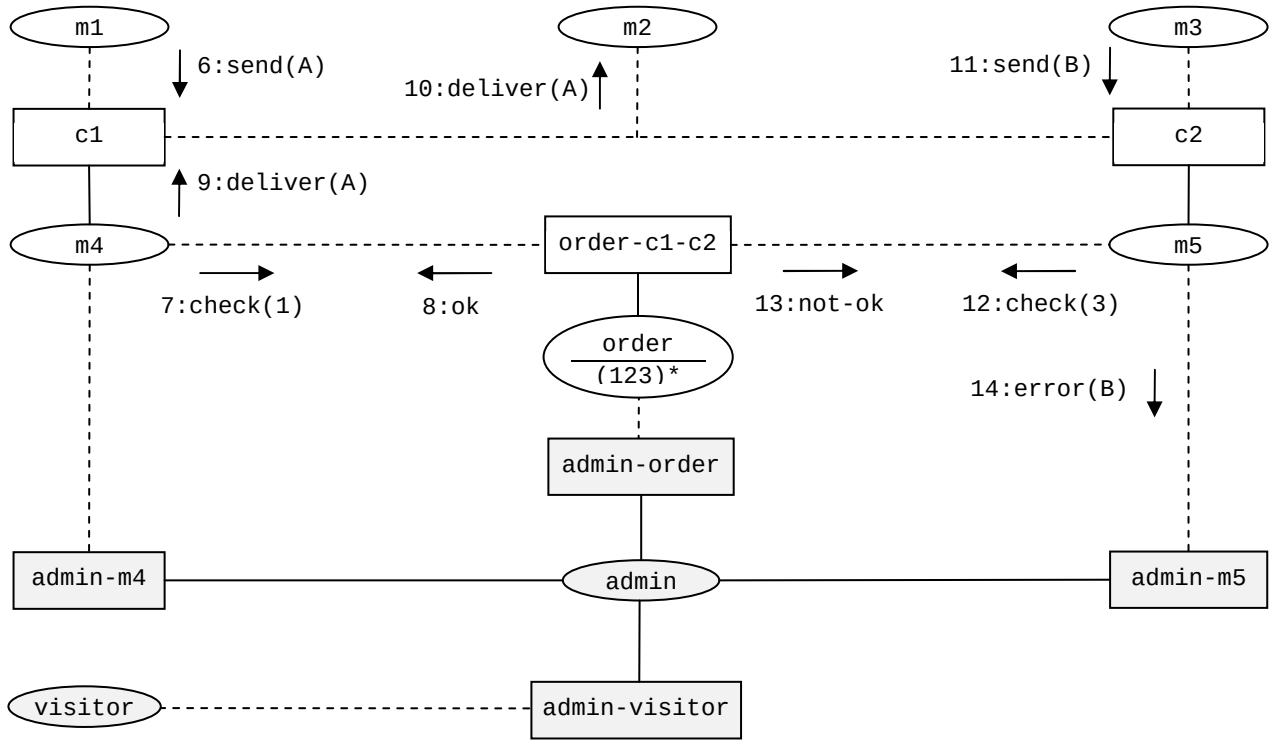


Figura 44: Mensajería Extendida – Secuencia Inválida de Mensajes en Canales con Extensión de Orden



La Figura 44 describe el comportamiento de la extensión cuando una secuencia inválida de mensajes es intercambiada sobre los canales. Supongamos que `m1` envía un mensaje `A` a `c1` – `6:send(A)`. El mensaje es reenviado al propietario

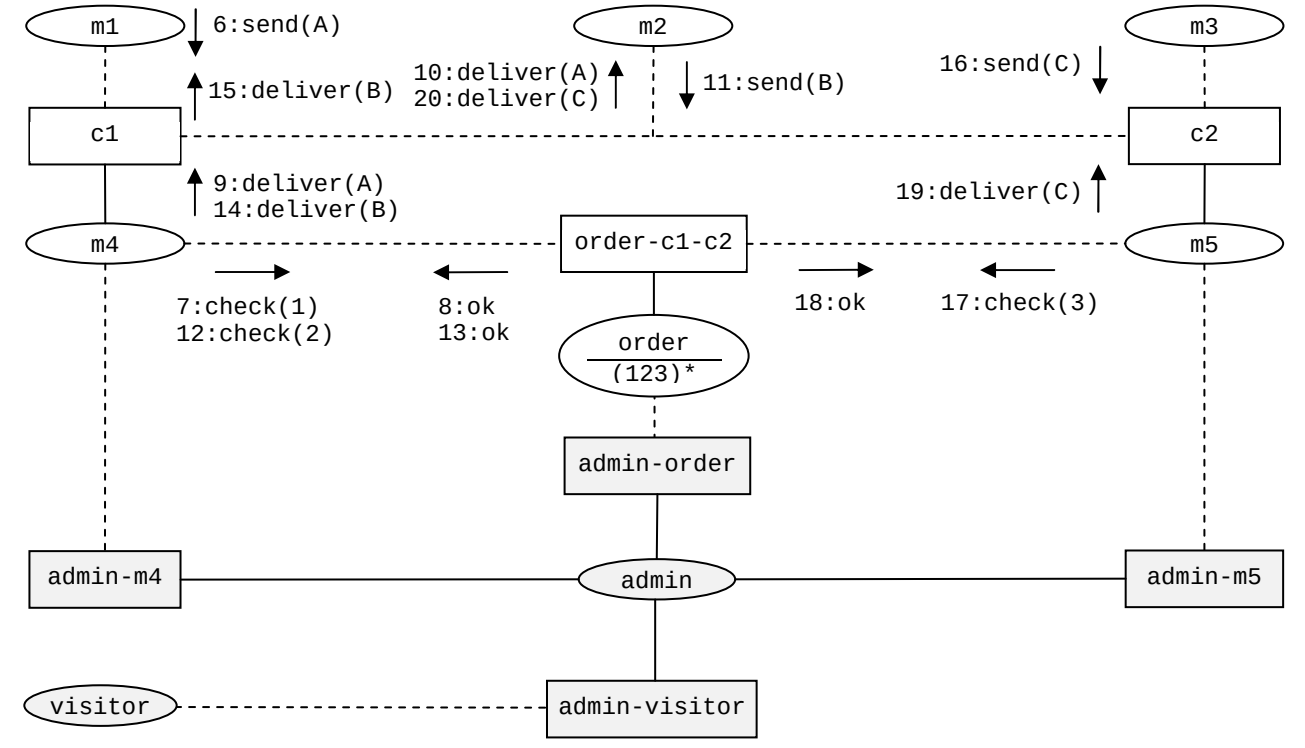
del canal m4, quien a su turno, entendiendo que el intercambio de mensajes es gobernado por la extensión de orden, emite una verificación de cumplimiento de orden a order – 7:check(1). order confirma que m1 está habilitado para enviar un mensaje en esta etapa del proceso y envía la confirmación de vuelta a m4 – 8:ok. Una vez confirmado, m4 entrega el mensaje a través del canal c1 – 9:deliver(A), el cual a su vez lo envía al suscriptor restante m2 – 10:deliver(A). En el próximo paso, m3 envía un mensaje B al canal c2 – 11:send(B), contrario al orden que sólo permite a m2 enviar después de m1. El propietario m5 emite la verificación de cumplimiento de orden a order – 12:check(3) quien determina que m3 no está habilitado para enviar mensajes en ese momento – 13:not-ok. Luego de recibir este mensaje, m5 no distribuye el mensaje original a los suscriptores de c2, sino que envía un mensaje de error al emisor m3 vía admin – 14:error(B).

La Figura 45 debajo muestra el comportamiento de la extensión configurada cuando se intercambia una secuencia válida de mensajes. El comienzo es igual al intercambio previo: m1 envía un mensaje A a c1 – 6:send(A); el propietario m4 emite una consulta a order – 7:check(1); order contesta afirmativamente – 8:ok; m4 distribuye el mensaje A a c1 – 9:deliver(A); el mensaje es entregado individualmente al suscriptor restante m2 – 10:deliver(A). En el paso siguiente, m2 envía un mensaje B a c1 – 11:send(B); el propietario m4 emite una consulta a order – 12:check(2); order confirma que m2 está habilitado para enviar un mensaje a través de c1 en este momento – 13:ok; el mensaje es entregado a c1 – 14:deliver(B); el mensaje es entregado al suscriptor restante m1 – 15:deliver(B). En el paso final, m3 envía el mensaje C a c2 – 16:send(C); el propietario m5 emite una verificación de control de orden a order – 17:check(3); order confirma que m3 está habilitado para enviar un mensaje a c2 en este momento – 18:ok; el mensaje es entregado a c2 – 19:deliver(C) y a través del mismo al suscriptor restante m2 – 20:deliver(C).

Orden es solo un tipo de extensión vertical. Otras extensiones verticales en G-EEG-EXTEND son:

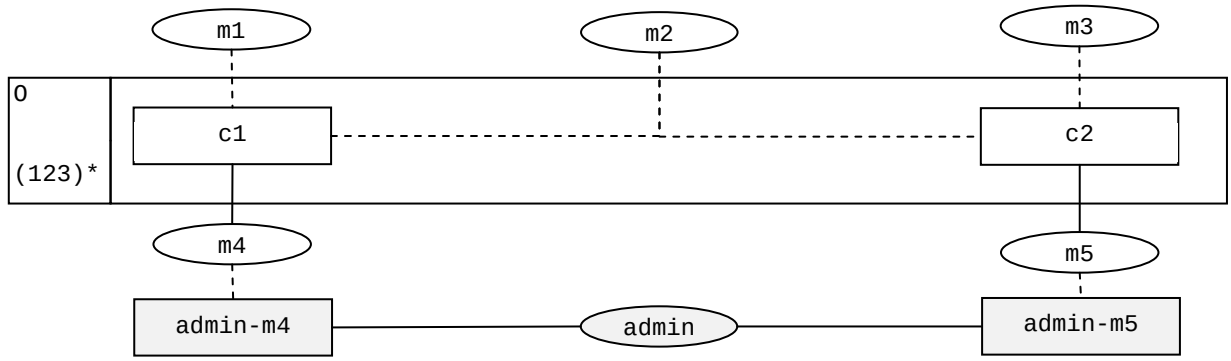
- *Puntualidad* – Monitorea si la mensajería realizada a través de varios canales satisface políticas de puntualidad definidas por el usuario;
- *Composición* – Permite la composición de canales existentes en canales más complejos, a través de operaciones como: vinculación, distribución, unión, filtrado y ruteo; y
- *Seguimiento* – Determina la ubicación de un mensaje en tránsito a través de un canal complejo.

Figura 45: Mensajería Extendida – Secuencia Válida de Mensajes en Canales con Extensión de Orden



Cuando se habilita una extensión vertical, se puede usar una notación abreviada para abstraer las formas complejas en las cuales se implementan y comportan tales extensiones. La Figura 46 muestra cómo representar la extensión de Orden (O) habilitada en los canales `c1` y `c2`, con `(123)*` como configuración actual de la extensión. Note que no se muestra ni el miembro `order`, ni el canal `order-c1-c2`.

Figura 46: Mensajería Extendida – Abstracción en Extensiones Verticales



4.5 Extensiones

La Sección 4.4 explicó el comportamiento de G-EEG, cubriendo la mensajería básica y extendida. Esta última fue explicada usando dos ejemplos de extensiones horizontales – validación y transformación, y una extensión vertical – orden. El objetivo de esta sección es introducir estas y otras extensiones representativas de mensajería para mostrar un rango de comportamientos útiles provistos por G-EEG y para proveer un conjunto inicial de extensiones para el repositorio de G-EEG-EXTEND.

Diez extensiones serán presentadas en esta sección. Este número incluye cinco extensiones horizontales, orientadas a canal: (1) Auditoria, (2) Validación, (3) Transformación, (4) Criptografía, y (5) Autenticación; dos extensiones horizontales orientadas a miembros: (6) Localización y (7) Alianza; y tres extensiones verticales: (8) Orden, (9) Puntualidad, y (10) Composición. Para cada extensión presentamos su: descripción – el objetivo y la funcionalidad provista; atributos – nombre, identificador, tipo, parámetros; y operaciones – las acciones requeridas para la entrega, habilitación, configuración y uso de la extensión. Las extensiones son presentadas en secciones individuales desde la sección 4.5.1 hasta la 4.5.10.

4.5.1 Extensión 1 - Auditoría

La extensión de Auditoria permite grabar en una base de datos todos los mensajes que transitan por un canal, y luego permite al propietario del canal recuperarlos desde la base de datos. Una extensión horizontal, orientada a canal, Auditoría tiene dos parámetros - un canal a auditar y una base de datos en la cual grabar los mensajes que circulan por dicho canal. Ver Tabla 8.

Tabla 8: Extensión 1 – Auditoría

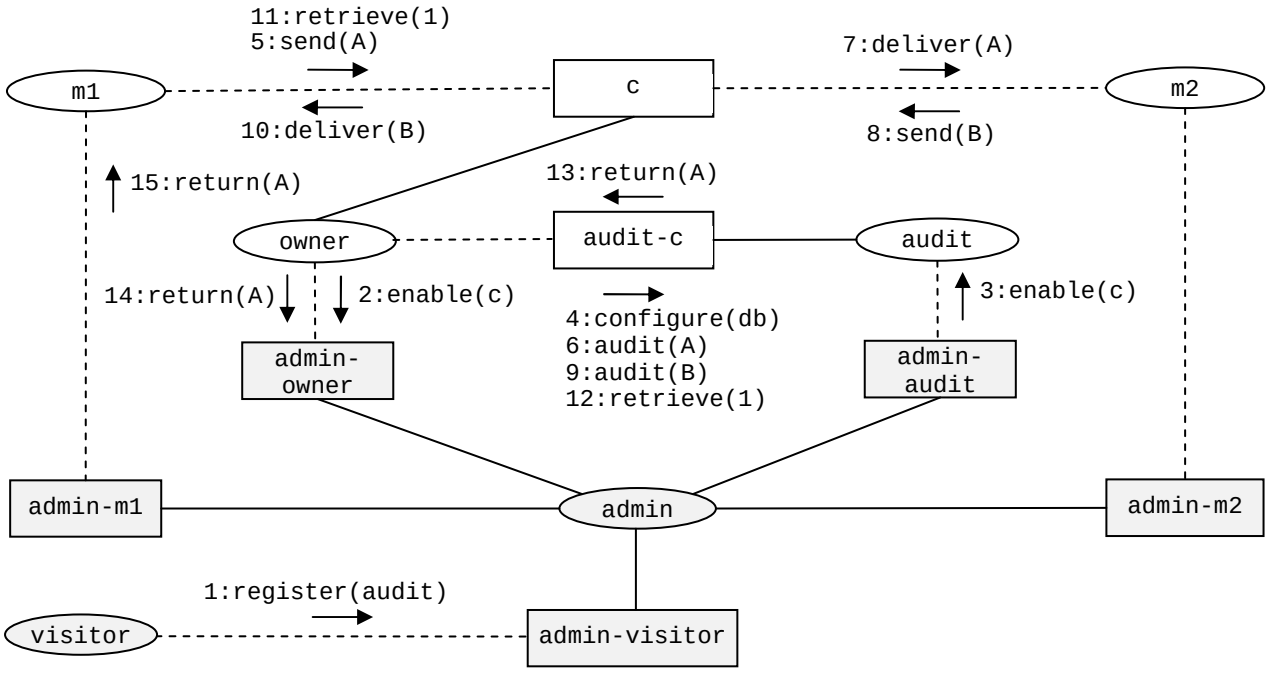
NOMBRE	Auditoría	
IDENTIFICADOR	A	
TIPO	Horizontal, Orientada a Canal	
PARÁMETROS	canal	canal a extender con Auditoría
	database	base de datos donde grabar los mensajes auditados

La entrega, habilitación, configuración y uso de la extensión de Auditoría involucra lo siguiente:

- *Entrega*: El miembro `visitor` solicita a `admin` registrar el miembro `audit` que implementa la extensión de auditoría enviando un mensaje de requerimiento `register(audit)` a través del canal `visitor-admin`. Una vez registrado, `audit` está disponible para que todos los miembros puedan habilitar la extensión de Auditoría en los canales que poseen.
- *Habilitación*: A fin de habilitar la extensión de Auditoría en un canal c, su propietario `owner` envía un mensaje de requerimiento `enable(c)` a `audit` vía el miembro `admin`. En respuesta, `audit` requiere a `admin` crear un canal `audit-c`, una vez que el canal es creado, suscribe a `owner` a este canal.
- *Configuración*: Una vez habilitada, la extensión puede ser configurada por el propietario enviando un mensaje de requerimiento `configure(database)` a `audit` especificando la base de datos a utilizar para almacenar los mensajes que circulan a través de c.
- *Uso*: Por cada mensaje enviado a través del canal c, una vez recibido por el propietario, el mensaje es enviado a `audit` vía `audit-c` para ser grabado en la base de datos en uso. Subsiguientemente, cualquier suscriptor `m` de c puede enviar un mensaje `retrieve(n)` al propietario de c para recuperar el n-ésimo mensaje enviado a c desde la última configuración de la extensión; parámetros de consulta más sofisticados son posibles. Una vez que el propietario recibe este requerimiento, reconoce su naturaleza especial y despacha el requerimiento a `audit` para grabar y recuperar el mensaje pedido, si existe en la base de datos. Luego que `audit` le responde al propietario, éste envía el mensaje al emisor `m` vía `admin`.

Una estructura y comportamiento típicos de la extensión de Auditoría se explica en el Ejemplo 33 y se muestra en la Figura 47 debajo.

Figura 47: Extensión 1 – Auditoría



Ejemplo 33: Extensión 1 – Auditoría

La Figura 47 presenta un comportamiento típico de la extensión de Auditoría, desde la entrega, a través de la habilitación y configuración, hasta el uso. La secuencia comienza con el miembro `visitor` requiriendo a `admin` registrar el miembro `audit` – `1:register(audit)`. Una vez que `audit` existe, el propietario de un canal c – `owner` solicita a `admin` habilitar la extensión de Auditoría en c – `2:enable(c)` y `3:enable(c)`. Esto resulta en la creación de un canal `audit-c`, a través del cual `owner` envía el pedido de configuración – `4:configure(db)` a `audit`. Esto completa la creación de las estructuras básicas, con el canal c configurado para auditar mensajes, su propietario `owner` y dos suscriptores `m1` y `m2`.

El resto del escenario se divide en tres pasos: (1) m1 envía un mensaje A a c, (2) m2 envía un mensaje B a c, y (3) m1 solicita la recuperación del primer mensaje que fue enviado a c, recibiendo A como respuesta. En el primer paso, m1 envía A a c – 5:send(A). Luego de recibir este mensaje, owner lo despacha para que sea grabado – 6:audit(A) y subsiguientemente lo entrega a los restantes suscriptores m2 – 7:deliver(A). El segundo paso es esencialmente el mismo: m2 envía B a c – 8:send(B), y owner despacha B para que sea grabado – 9:audit(B) y lo entrega al restante suscriptor – 10:deliver(B). El último paso comienza con m1 solicitando recuperar el primer mensaje enviado – 11:retrieve(1). Una vez recibido por owner, el requerimiento es despachado a audit – 12:retrieve(1) quien recupera el mensaje solicitado A de la base de datos y lo envía al emisor m1 a través de admin – 13:return(A) y 14:return(A).

4.5.2 Extensión 2 - Validación

La extensión de Validación permite validar todos los mensajes que circulan a través de un canal dado de acuerdo a un formato de mensaje pre-definido. Como todos los mensajes son escritos en XML, el formato es expresado en lenguaje XML Schema. Cuando se recibe un mensaje válido, la extensión entrega el mensaje a todos los suscriptores del canal excepto al emisor, como es habitual. Sin embargo, cuando se detecta un mensaje inválido, envía un mensaje de error al emisor a través del administrador. Una extensión horizontal, orientada a canal, Validación tiene dos parámetros: un canal a validar y un XML Schema especificando el formato requerido para los mensajes que circulen por el canal. Ver Tabla 9.

Tabla 9: Extensión 2 – Validación

NOMBRE	Validación	
IDENTIFICADOR	V	
TIPO	Horizontal, Orientada a Canal	
PARÁMETROS	canal	canal a extender con Validación
	sintaxis	XML Schema determinando el formato de los mensajes que pasan a través del canal

La entrega, habilitación, configuración y uso de la extensión de Validación involucra lo siguiente:

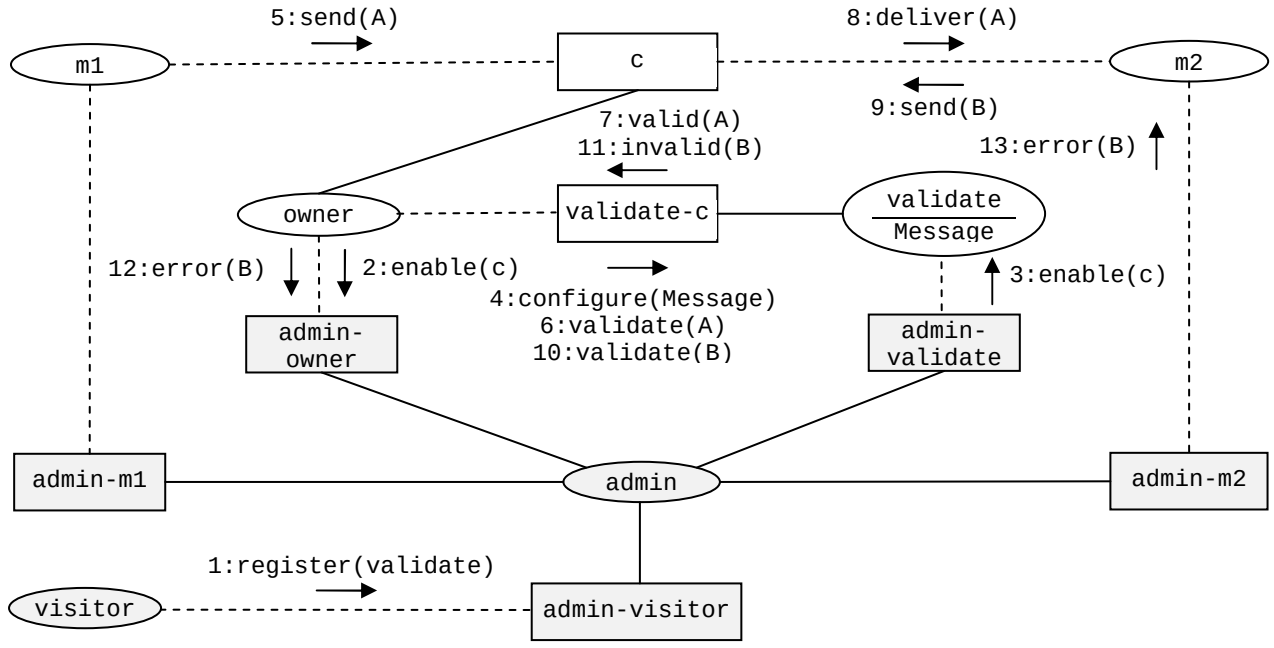
- *Entrega*: El miembro visitor solicita a admin registrar el miembro validate que implementa la extensión de validación, enviando un mensaje de requerimiento register(validate) a través del canal visitor-admin.
- *Habilitación*: A fin de habilitar la extensión de Validación en un canal c, su propietario owner envía un mensaje a validate vía admin – enable(c). En respuesta, validate requiere a admin crear un canal validate-c, una vez que el canal es creado, suscribe a owner a este canal.
- *Configuración*: Una vez que la extensión es habilitada en c, el propietario especifica la sintaxis requerida para los mensajes que circulan por c enviando un mensaje de solicitud de configuración a validate – configure(schema). Generalmente, schema nombra a un tipo complejo definido en un documento XML Schema dado. La extensión puede ser reconfigurada en cualquier momento por el propietario del canal reenviando el pedido de configuración a validate con la nueva sintaxis de mensajes. El cambio tendrá efecto para todos los mensajes subsiguientes que transiten a través de c.
- *Uso*: Por cada mensaje enviado a través del canal, una vez recibido por el propietario, el mensaje es despachado a validate quien controla el mensaje con la sintaxis de mensaje en uso y responde al propietario. Si la respuesta es positiva (el mensaje es válido), el propietario entrega el mensaje a todos los suscriptores como siempre. En caso contrario (el mensaje es inválido), el propietario envía una notificación de error al emisor del mensaje vía el administrador.

El Ejemplo 34 ilustra el comportamiento de la extensión, desde la entrega y habilitación, hasta la configuración y uso.

Ejemplo 34: Extensión 2 – Validación

La Figura 48 presenta un comportamiento típico de la extensión de Validación. La secuencia comienza cuando `visitor` solicita a `admin` registrar al miembro `validate` – `1:register(validate)`. Una vez que el miembro existe, el propietario de un canal c – `owner` requiere a `validate` vía `admin` habilitar la extensión de Validación en el canal `c` – `2:enable(c)` y `3:enable(3)`. Esto resulta en la creación del canal `validate-c`, a través del cual, luego, `owner` envía el requerimiento – `4:configure(Message)` a `validate`; `Message` es el tipo XML Schema definido en la Sección 4.4.2. Esto completa la creación de las estructuras básicas, con el canal c configurado para validar mensajes, su propietario `owner`, y dos suscriptores `m1` y `m2`. Subsiguientemente, la figura muestra dos escenarios: uno donde `m1` envía un mensaje válido A a c y otro donde `m2` envía un mensaje inválido B a c; A y B son como los definidos en la Sección 4.4.2. En el primer escenario, `m1` envía un mensaje A a c – `5:send(A)`, y el propietario solicita a `validate` validar el mensaje – `6:validate(A)`. La respuesta es positiva – `7:valid(A)` y `owner` entrega A a `m2` – `8:deliver(A)`. En el segundo escenario, `m2` envía un mensaje B a c – `9:send(B)`. El pedido de `owner` para validar el mensaje – `10:validate(B)` produce un resultado negativo – `11:invalid(B)` y `owner` pasa un mensaje de error a `m2` vía `admin` – `12:error(B)` y `13:error(B)`.

Figura 48: Extension 2 – Validación



4.5.3 Extensión 3 - Transformación

La Extensión de Transformación permite transformar todos los mensajes que circulan a través de un canal dado de un formato a otro, de acuerdo a un conjunto predefinido de reglas de transformación. Como todos los mensajes son escritos en XML, las reglas son escritas en lenguaje XSLT, especialmente diseñado para este propósito. Una extensión horizontal, orientada a canal, Transformación tiene dos parámetros: un canal sobre el cual la extensión es habilitada, y una plantilla XSLT que especifica el conjunto actual de reglas de transformación. Ver Tabla 10.

Tabla 10: Extensión 3 – Transformación

NOMBRE	Transformación
IDENTIFICADOR	T

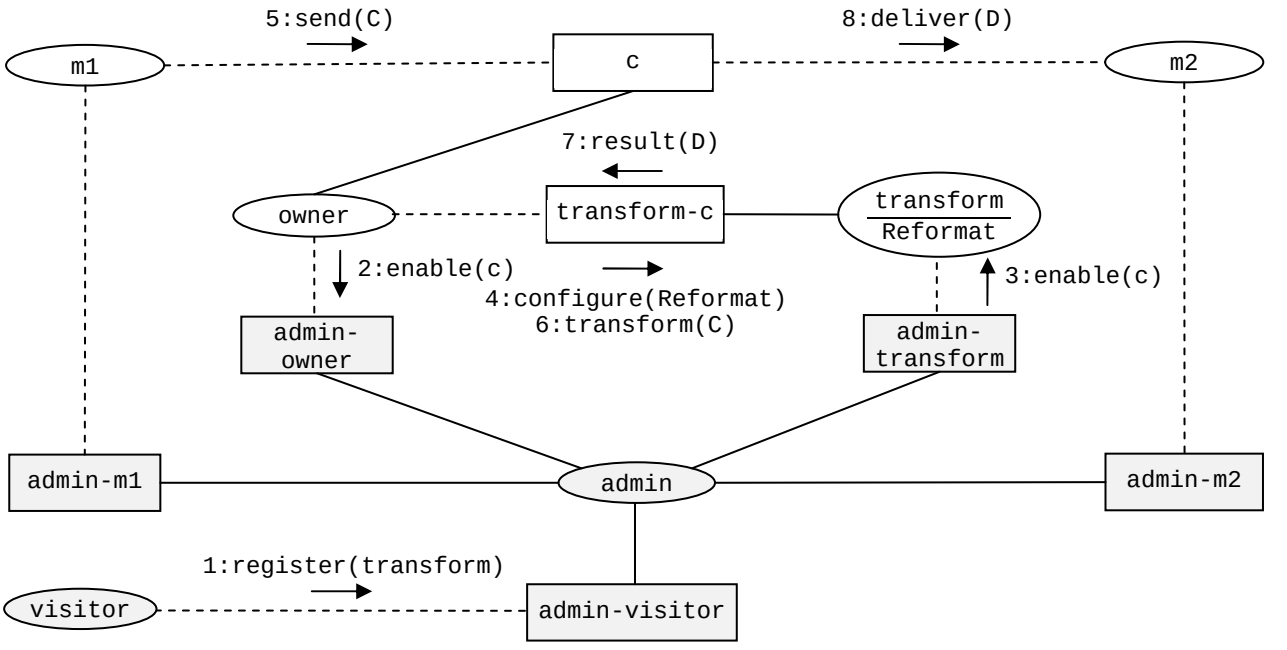
TIPO	Horizontal, Orientada a canal	
PARÁMETROS	canal	canal a extender con Transformación
	reglas	plantilla XSLT para transformar mensajes que pasan a través del canal

La entrega, habilitación, configuración y uso de la extensión de Transformación involucra lo siguiente:

- *Entrega:* El miembro `visitor` solicita a `admin` registrar el miembro `transform` que implementa la extensión de transformación, enviando el mensaje `register(transform)` a través del canal `visitor-admin`.
- *Habilitación:* A fin de habilitar la extensión de Transformación en un canal c, el propietario `owner` envía un requerimiento a `transform` vía `admin` – `enable(c)`. En respuesta, `transform` solicita a `admin` crear un canal `transform-c`, una vez que el canal es creado, suscribe a `owner` a este canal.
- *Configuración*: Una vez que la extensión es habilitada en c, el propietario especifica la transformación requerida en los mensajes que circulan por c enviando un requerimiento de configuración a `transform` – `configure(Reformat)`. Generalmente, `Reformat` nombra a una plantilla definida en un documento XSLT dado. La extensión puede ser reconfigurada en cualquier momento por el propietario del canal reenviando el pedido de configuración a `transform` con la nueva plantilla de transformación. El cambio tendrá efecto inmediatamente para todos los mensajes subsiguientes que transiten a través de c.
- *Uso*: Por cada mensaje enviado a través del canal, una vez recibido por el propietario, el mensaje es despachado a `transform` quien aplica las reglas de transformación para reformatear el mensaje. El resultado es enviado de vuelta al propietario del canal. Mientras que la ejecución de una plantilla XSLT producirá un documento XML reformateado para cualquier XML de input, tales plantillas son particularmente aconsejadas para transformar mensajes que conforman con un formato XML determinado. Por lo tanto, la extensión de Transformación es usualmente usada en conjunción con la extensión de Validación, como se mostró en la Sección 4.4.2.

Una estructura y comportamiento típicos de la extensión de Transformación se explica en el Ejemplo 35 y se muestra en la Figura 49 debajo. El comportamiento cubre todas las etapas, desde la entrega, a través de la habilitación y configuración, hasta el uso.

Figura 49: Extensión 3 – Transformación



Ejemplo 35: Extensión 3 – Transformación

La Figura 49 presenta una estructura de comunicación típica para la extensión de Transformación, con un canal c que posee la extensión de Transformación habilitada, propiedad del miembro owner y suscripto por los miembros m1 y m2, el miembro transform que implementa la extensión de transformación, el canal transform-c conectando owner y transform, miembros pre-definidos admin y visitor, y todos los canales conectando miembros con admin.

La figura también muestra un comportamiento típico de la extensión de Transformación. Comienza cuando visitor solicita a admin registrar al miembro transform – 1:register(transform). Una vez que el miembro existe, owner requiere a transform habilitar la extensión de Transformación en el canal c, enviando el pedido vía admin – 2:enable(c) y 3:enable(c). Esto resulta en la creación del canal transform-c, que es usado por owner para solicitar la configuración de la extensión para transformar mensajes de acuerdo a la plantilla Reformat – 4:configure(Reformat), definido en la Sección 4.4.2 junto con los mensajes C (input de la transformación), y D (output de la transformación) . Esto completa la creación de las estructuras básicas. A continuación, supongamos que el miembro m1 envía un mensaje C a c – 5:send(C). Una vez recibido por owner, el mensaje es despachado a transform – 6:transform(C), quien aplica a C las reglas de transformación definidas en la plantilla Reformat y devuelve el mensaje D – 7:result(D). A continuación, entrega el mensaje transformado D a los restantes suscriptores m2 – 8:deliver(D).

4.5.4 Extensión 4 – Criptografía

La extensión de Criptografía permite el encriptado de todos los mensajes que pasan por un canal dado, para ser transferidos por el canal en forma cifrada, y la desencriptación a su forma original antes de que sean recibidos. Una extensión horizontal, orientada a canal, Criptografía tiene dos parámetros: el canal sobre el que se habilita la extensión y el par de claves pública y privada usadas para encriptar y desencriptar los mensajes. Ver Tabla 11.

Tabla 11: Extensión 4 – Criptografía

NOMBRE	Criptografía	
IDENTIFICADOR	C	
TIPO	Horizontal, Orientada a canal	
PARÁMETROS	canal	canal a extender con Criptografía
	reglas	par de claves pública y privada usadas para encriptar y desencriptar mensajes

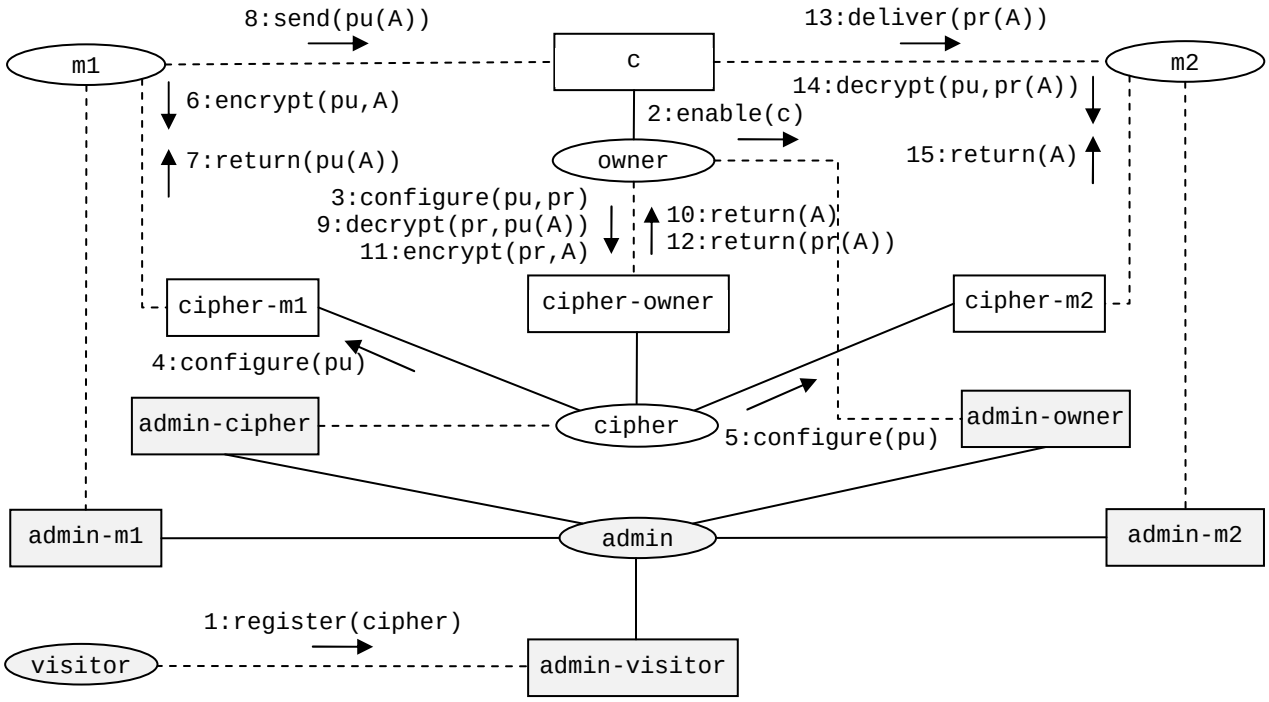
La entrega, habilitación, configuración y uso de la extensión de Criptografía involucra lo siguiente:

- *Entrega*: El miembro visitor solicita a admin el registro del miembro cipher que implementa la extensión de Criptografía, enviando el mensaje register(cipher) a admin.
- *Habilitación*: A fin de habilitar la extensión de Criptografía sobre un canal c, el propietario owner envía un requerimiento a cipher vía admin – enable(c). En respuesta, cipher solicita a admin la creación de los canales cipher-owner, cipher-m1 y cipher-m2 y suscribe a owner, m1 y m2 al canal correspondiente.
- *Configuración*: Después que la extensión es habilitada en c, es configurada por owner quien envía un par de claves pública y privada a cipher vía el canal cipher-owner, quien a su turno despacha las claves públicas a todos los suscriptores a lo largo de los canales cipher-m1 y cipher-m2. La extensión puede ser reconfigurada por owner en cualquier momento reenviando la solicitud de configuración a cipher con un nuevo par de claves pública y privada.
- *Uso*: El uso es descripto en dos versiones – cuando envía un mensaje el propietario del canal y cuando lo envía un suscriptor. Si es enviado por el propietario, el mensaje es reenviado primero a cipher vía el canal cipher-owner. Al recibir el mensaje, cipher encripta el mensaje con la clave privada y lo devuelve a owner vía cipher-owner quien, a su turno, despacha el mensaje encriptado a todos los suscriptores. Al recibir el mensaje, cada suscriptor reenvía el mensaje a cipher vía cipher-m1 o cipher-m2 quien lo desencripta con la clave pública y envía de vuelta el mensaje original. Si el mensaje es enviado por un suscriptor, es reenviado primero a cipher quien encripta el mensaje con la clave pública y lo devuelve. El mensaje encriptado es enviado al canal. Al recibir

el mensaje, owner reenvía el mensaje a cipher quien lo desencripta con la clave privada y devuelve el mensaje original. El propietario envía nuevamente el resultado a cipher, esta vez para encriptar el mensaje con la clave privada y para que lo devuelva. El mensaje encriptado resultante es enviado a todos los suscriptores. Sobre el recibo, cada suscriptor envía el mensaje encriptado a cipher quien lo desencripta con la clave pública y devuelve el mensaje original. Ver Ejemplo 36 debajo.

Una estructura y comportamiento típicos de la extensión de Criptografía se explica en el Ejemplo 36 y se muestra en la Figura 50 debajo. El comportamiento cubre todas las etapas, desde la entrega, a través de la habilitación y configuración, hasta el uso.

Figura 50: Extensión 4 – Criptografía



Ejemplo 36: Extensión 4 – Criptografía

La Figura 50 debajo describe una estructura típica de comunicación para la extensión de Criptografía. La estructura muestra un canal c extendido con Criptografía con propietario owner y miembros m1 y m2 como suscriptores. La estructura muestra también el miembro cipher que implementa la extensión, junto con los canales cipher-owner, cipher-m1 y cipher-m2 conectando cipher a owner, m1 y m2 respectivamente. Finalmente, se muestran también los miembros predefinidos admin y visitor y todos los canales predefinidos que conectan admin con otros miembros.

Basada en esta estructura, la Figura 50 muestra el comportamiento típico de la extensión de Criptografía. La secuencia comienza con visitor solicitando a admin el registro de cipher – 1:register(cipher). Después que el miembro es creado, owner solicita a cipher vía admin que habilite la extensión sobre c – 2:enable(c). De esto resultan los canales cipher-owner, cipher-m1 y cipher-m2 conectando a cipher con los miembros. Luego, owner solicita a cipher configurar la extensión con un par de claves pública y privada pu y pr – 3:configure(pu,pr) y cipher reenvía la clave pública pu a todos los suscriptores vía los canales cipher-m1 y cipher-m2 – 4:configure(pu) y 5:configure(pu). Esto completa las estructuras básicas para la extensión de Criptografía sobre el canal c.

Supongamos que m1 quisiera enviar un mensaje A a c. El mensaje es enviado primero a cipher para encripción con

clave pública – `6:encrypt(pu,A)` y el mensaje retornado – `7:return(pu(A))` es enviado a c – `8:send(pu(A))`. Sobre el recibo, `owner` envía el mensaje a `cipher` para desencripción con clave privada – `9:decrypt(pr,pu(A))` recibiendo de nuevo el mensaje original – `10:return(A)`. Una vez más, `owner` envía `A` a `cipher` para encripción con clave privada – `11:encrypt(pr,A)`, recibe el mensaje encriptado – `12:return(pr(A))` y lo envía al suscriptor restante `m2` – `13:deliver(pr(A))`. Cuando el mensaje es recibido por `m2`, es enviado a `cipher` para desencripción con clave pública – `14:decrypt(pu,pr(A))`, recibiendo de vuelta el mensaje original – `15:return(A)`.

4.5.5 Extensión 5 – Autenticación

La extensión de Autenticación permite controlar que solo aquellos miembros que puedan probar su identidad hagan uso de un canal. La identidad es establecida en base a la asignación de claves a miembros. Una extensión horizontal, orientada a canal, Autenticación tiene dos parámetros: el canal sobre el cual se habilita la extensión y la clave asignada por el propietario a cada suscriptor del canal. Ver Tabla 12.

Tabla 12: Extensión 5 – Autenticación

NOMBRE	Autenticación	
IDENTIFICADOR	U	
TIPO	Horizontal, Orientada a Canal	
PARÁMETROS	canal	canal a ser extendido con Autenticación
	claves	claves usadas para autenticar los miembros que usan el canal

La entrega, habilitación, configuración y uso de la extensión de Autenticación involucra lo siguiente:

- *Entrega*: El miembro `visitor` solicita a `admin` registrar el miembro `auth` que implementa la extensión de Autenticación, enviando el mensaje `register(auth)` a `admin`.
- *Habilitación:* A fin de habilitar la extensión de Autenticación sobre un canal c, el propietario `owner` envía un requerimiento a `auth` vía `admin` – `enable(c)`. En respuesta, `auth` le solicita a `admin` crear un canal `auth-owner` y, una vez que el canal es creado, suscribe `owner` al mismo.
- *Configuración*: Después que la extensión es habilitada en c, es configurada por el propietario quien envía un conjunto de claves a `auth`, incluyendo su propia clave y las claves de los suscriptores. Luego, `auth` reenvía las claves a todos los suscriptores vía `admin`. La extensión puede ser reconfigurada por el propietario re-enviando un nuevo grupo de claves a `auth`. Cuando un miembro existente se des-susbcribe, el propietario solicita la anulación de la clave a `auth`. Cuando se suscribe un nuevo miembro, el propietario genera una nueva clave y le solicita a `auth` que la pase.
- *Uso*: A los suscriptores al canal extendido con Autenticación se les requiere que envíen sus claves junto con sus mensajes; para asegurar el intercambio de claves se podría aplicar la extensión de Criptografía. Una vez que el mensaje y la clave son enviados al canal y recibidos por el propietario, éste reenvía la clave y la identidad del emisor a `auth` quien controla que la clave recibida es la misma que la registrada, e informa al propietario. Si la clave es correcta, el propietario solo entrega el mensaje sin la clave a todos los suscriptores restantes. De lo contrario, el propietario envía un mensaje de error al emisor vía `admin`.

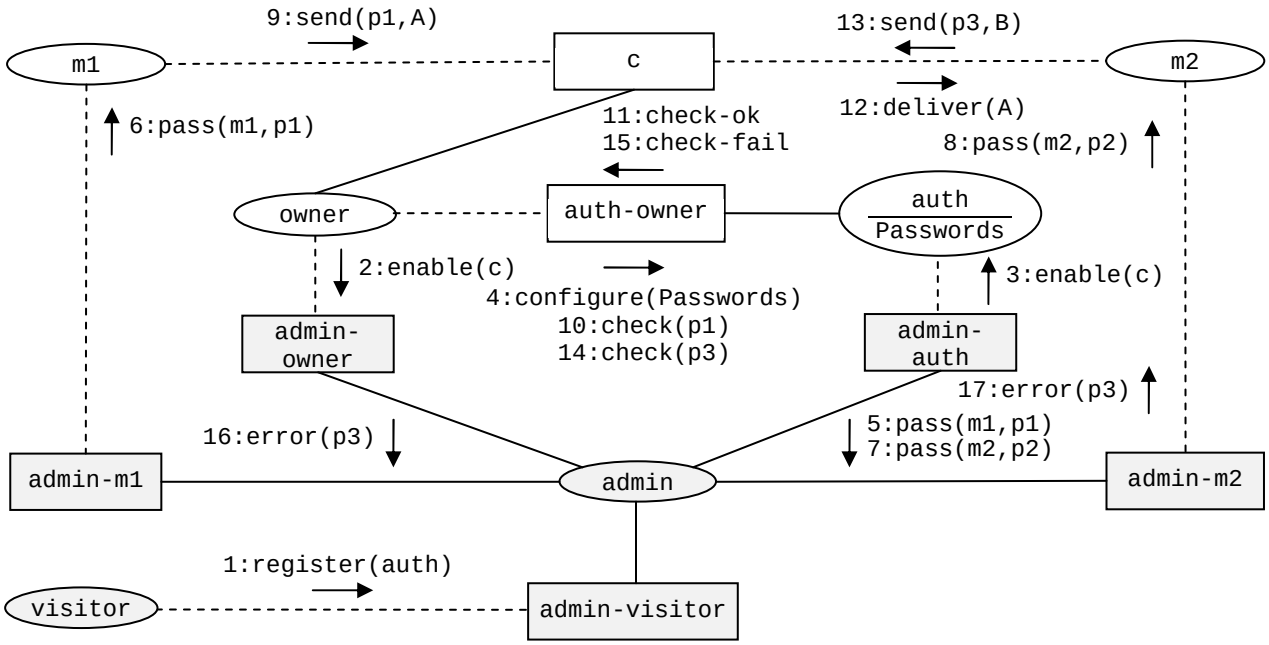
El Ejemplo 37 ilustra el comportamiento de la extensión, desde la entrega y habilitación, hasta la configuración y uso.

Ejemplo 37: Extensión 5 – Autenticación

La Figura 51 muestra una estructura típica de comunicación para la extensión de Autenticación, con el canal c extendido con Autenticación, su propietario `owner` y dos suscriptores `m1` y `m2`. La figura muestra también el miembro `auth` que implementa la extensión, conectado a `owner` por medio del canal `auth-owner`. Finalmente, se muestran también los miembros predefinidos `admin` y `visitor` y todos los canales predefinidos que conectan a los miembros con `admin`.

Dada esta estructura, la figura muestra dos escenarios típicos para el intercambio de mensajes sobre un canal c con extensión de Autenticación. La secuencia comienza con `visitor` solicitando a `admin` el registro del miembro `auth` – `1:register(auth)`. Una vez registrado, `owner` solicita a `auth` vía `admin` que habilite la extensión de autenticación en `c` – `2:enable(c)` y `3:enable(c)`, `owner` configura la extensión enviando las claves a `auth` – `4:configure(Passwords)`, las cuales son diseminadas posteriormente por `auth` a los miembros individuales vía `admin` – `5:pass(m1,p1)`, `6:pass(m1,p1)`, `7:pass(m2,p2)` y `8:pass(m2,p2)`. Esto completa las estructuras básicas. Supongamos que `m1` quiere enviar un mensaje `A` a `c`. Esto es hecho por `m1` enviando el mensaje junto con su clave `p1` – `9:send(p1,A)`. Al recibir el mensaje, `owner` solicita la verificación de clave a `auth` – `10:check(p1)` quien responde positivamente – `11:check-ok`, por lo tanto `owner` entrega el mensaje sin clave al suscriptor restante `m2` – `12:deliver(A)`. En el segundo escenario, supongamos que `m2` desea enviar un mensaje `B` a `c`, pero aplica una clave incorrecta `p3` – `13:send(p3,B)`. En esta oportunidad, cuando `owner` solicita a `auth` la verificación de la clave – `14:check(p3)`, `auth` responde en forma negativa – `15:check-fail`, y `owner` envía un mensaje de error de vuelta al emisor `m2` vía `admin` – `16:error(p3)` y `17:error(p3)`.

Figura 51: Extensión 5 – Autenticación



XML 8 debajo presenta un esquema XML que define una posible estructura de datos para los parámetros de configuración usados por la extensión de Autenticación. La estructura puede contener cualquier número de elementos `MemberPasswords`, con cada elemento especificando un miembro (`MemberId`) y su clave (`Password`).

XML 8: Extensión 5 – Esquema de Parámetros de Autenticación

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This schema defines the parameters for the G-EEG Authentication Extensión -->
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xg2g=http://iist.unu.edu/elsa/G-EEG/xmlUtil"
    targetNamespace=http://iist.unu.edu/elsa/G-EEG/xmlUtil
    elementFormDefault="qualified">

  <xs:element name="MemberPasswords" minOccurs="1" maxOccurs="unbounded" type="Member">

  <xs:complexType name="Member">
    <xs:sequence>
      <xs:element name="MemberId" type="xs:string" />
```

```
      <xs:element name="Password" type="xs:string" />
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

4.5.6 Extensión 6 – Localización

La extensión de Localización ofrece servicios para proveer información de miembros y canales a los miembros de G-EEG. Está basada en un conjunto de atributos y valores asignados a miembros y canales durante su registro y creación, permitiendo a los miembros formular consultas y descubrir miembros y canales con ciertas combinaciones de atributos y valores. Una extensión horizontal, orientada a miembro, Localización tiene dos parámetros – un conjunto de atributos válidos para descripción de miembros y otro conjunto válido para descipción de canales. Ver Tabla 13.

Tabla 13: Extensión 6 – Localización

NOMBRE	Localización	
IDENTIFICADOR	D	
TIPO	Horizontal, Orientada a Miembro	
PARÁMETROS	Atributos de miembro	conjunto de atributos para descripción de miembros
	Atributos de canal	conjunto de atributos para descripción de canales

La entrega, habilitación, configuración y uso de la extensión de Localización involucra lo siguiente:

- *Entrega*: El miembro `visitor` solicita a `admin` el registro del miembro `discover` que implementa la extensión de Localización, enviando el mensaje `register(discover)` a `admin`.
- *Configuración*: A diferencia de las extensiones orientadas a canal que son habilitadas y configuradas por propietarios de canal, una extensión orientada a miembro como Localización es configurada por `admin` para que la usen todos los miembros, y luego habilitada para su uso por miembros individuales. A fin de configurar la extensión, `admin` envía el mensaje `configure(Attrs)` a `discover` con el conjunto de atributos `Attrs` que pueden ser usados para descripción de miembros y canales.
- *Habilitación*: Para obtener acceso al servicio de Localización, un miembro `m` debe primero enviar el requerimiento `enable(m)` a `discover`. En respuesta, `discover` crea un canal `discover-m` y suscribe a `m` a dicho canal.
- *Uso*: Cada vez que se registra o des-registra un miembro, o se crea o destruye un canal, `admin` informa a `discover` sobre esto vía el canal `admin-discover`. Adicionalmente, durante el registro de un miembro, los atributos y valores del miembro registrado son especificados y pasados a `discover`, y similarmente los atributos y valores de los canales pueden ser especificados y registrados durante la creación del canal. Con `discover` manteniendo los atributos y valores de miembros y canales, un miembro `m` puede efectuar consultas vía el canal `discover-m` acerca de la existencia de otros miembros y canales, caracterizados por configuraciones específicas de atributos y valores.

El Ejemplo 38 ilustra el comportamiento de la extensión, desde la entrega y habilitación, hasta la configuración y uso.

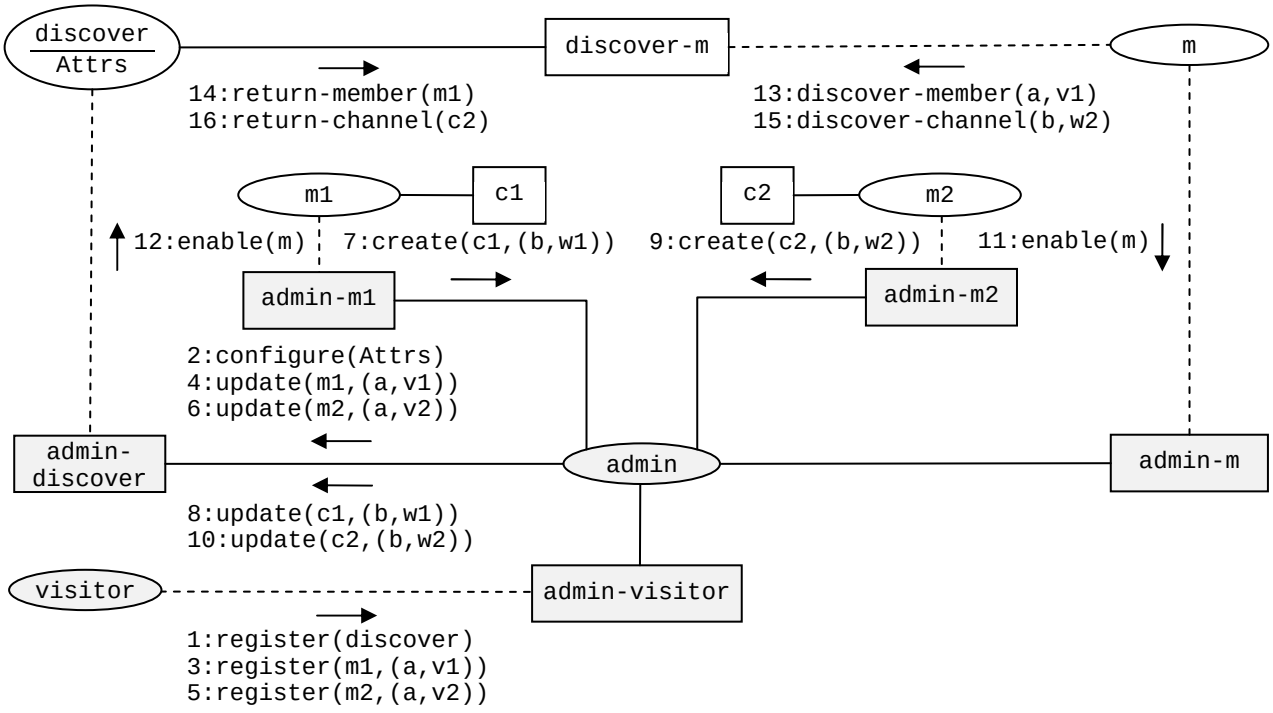
Ejemplo 38: Extensión 6 – Localización

La Figura 52 muestra una estructura de comunicación típica para la extensión de Localización, con el miembro `m` extendido con Localización. La figura muestra el miembro `discover` que implementa la extensión conectado a `m` a través del canal `discover-m`. Asimismo, se muestran los miembros `m1` y `m2` poseyendo los canales `c1` y `c2` respectivamente. Por último, también se muestran los miembros predefinidos `admin` y `visitor` y todos los canales predefinidos que conectan los miembros con `admin`.

Dada esta estructura, la figura describe el comportamiento típico de la extensión de Localización. La secuencia comienza con `visitor` solicitando a `admin` la registración del miembro `discover` – `1:register(discover)`, seguido por `admin` enviando un requerimiento de configuración a `discover` – `2:configure(Attrs)` con el conjunto `Attrs` de atributos válidos orientados a miembro y a canal. Siguiendo, `visitor` solicita a `admin` el registro

de un nuevo miembro m1 con atributo a asignado a v1, y un nuevo miembro m2 con atributo a asignado a v2 – 3:register(m1,(a,v1)) y 5:register(m2,(a,v2)). Cada vez que ocurre un registro, admin notifica a discover acerca del hecho – 4:update(m1,(a,v1)) y 6:update(m2,(a,v2)). Una vez que m1 y m2 existen, le solicitan a admin la creación de los canales c1 y c2 con atributo b asignado a w1 y w2 respectivamente – 7:create(c1,(b,w1)) y 9:create(c2,(b,w2)). Cada vez que esto ocurre, admin notifica a discover acerca de dichas operaciones – 8:update(c1,(b,w1)) y 10:update(c2,(b,w2)). Antes de que esté completa la configuración básica, el miembro m solicita a discover vía admin que habilite la extensión – 11:enable(m) y 12:enable(m). Esto crea el canal discover-m y suscribe m al canal. Para usar esta extensión, supongamos que m desea ubicar a todos los miembros registrados con atributo a asignado a v1, y todos los canales existentes con atributo b asignado a w2. Esto es hecho por m enviando el requerimiento 13:discover-member(a,v1) y 15:discover-channel(b,w2) a discover vía el canal discover-m. Después de buscar en la base de datos, discover encuentra que m1 y c2 cumplen los requisitos correspondientes y responde a m por medio de 14:return-member(m1)y 16:return-channel(c2).

Figura 52: Extensión 6 – Localización



4.5.7 Extensión 7 – Alianza

La extensión Alianza permite reunir a un grupo de miembros que deciden llevar a cabo de manera conjunta la mensajería en ciertos canales. Con esta extensión habilitada, los miembros pueden intercambiar mensajes individualmente y también como parte de una o más Alianzas. Cada alianza es administrada por un miembro designado llamado coordinador, quien envía los mensajes recibidos del exterior a todos los miembros de la alianza, y coordina también el envío de mensajes por los miembros de la alianza cuando todos coinciden en hacerlo de este modo. En nombre de la Alianza, el coordinador puede poseer y suscribir diferentes canales. Una extensión horizontal, orientada a miembro, Alianza tiene dos parámetros: el nombre de la alianza y el conjunto de miembros que la componen. Ver Tabla 14.

Tabla 14: Extensión 7 – Alianza		
NOMBRE	Alianza	
IDENTIFICADOR	L	
TIPO	Horizontal, Orientada a Miembro	
PARÁMETROS	nombre	nombre de la Alianza
	miembros	miembros que integran la Alianza

La entrega, habilitación, configuración y uso de la extensión de Alianza involucra lo siguiente:

- *Entrega*: El miembro `visitor` solicita a `admin` el registro de un miembro `alliance` que implementa la extensión Alianza enviando un mensaje `register(alliance)` a través del canal `visitor-admin`.
- *Habilitación*: A fin de habilitar una alianza `ma`, todos los miembros potenciales de la alianza envían un requerimiento `enable(ma)` a `alliance` vía `admin`. En respuesta, `alliance` solicita a `admin` la creación del coordinador `ma`.
- *Configuración*: Una vez que el coordinador existe, `alliance` lo configura con los nombres de todos los miembros fundadores enviándole el mensaje `configure(members)` vía `admin`. A su vez, el coordinador solicita a `admin` la creación de un canal interno `alliance-ma` para la alianza, y suscribe a todos los miembros fundadores.
- *Uso*: Cualquier miembro de la alianza puede enviar un mensaje a otro miembro de la alianza vía `alliance-ma` solicitando que el mensaje sea enviado en nombre de toda la alianza. Seguidamente, el coordinador debe recibir confirmaciones positivas de los otros miembros antes de decidir si envía el mensaje. Si sólo un miembro responde negativamente, el mensaje no es enviado; este comportamiento todos-o-ninguno de la alianza puede ser cambiado, a comportamiento basado-en-mayoría, por ejemplo. En esta instancia concreta, los miembros acuerdan para que la alianza cree sus propios canales o se suscriba a canales existentes. Una vez acordado, el coordinador solicita a `admin` la creación de nuevos canales, que serán propiedad de la alianza (técnicamente, del coordinador mismo) o solicita a varios miembros que suscriban a la alianza a los canales que ellos poseen (técnicamente, el coordinador es el suscriptor), con todas las decisiones alcanzadas por consenso entre todos los miembros de la alianza. Por otro lado, cuando un miembro externo envía un mensaje al canal propiedad de la alianza, el coordinador envía primero el mensaje a todos los miembros de la alianza a través del canal interno `alliance-ma`, antes de distribuirlo a los miembros externos.

El Ejemplo 39 ilustra el comportamiento de la extensión, desde la entrega y habilitación, hasta la configuración y uso.

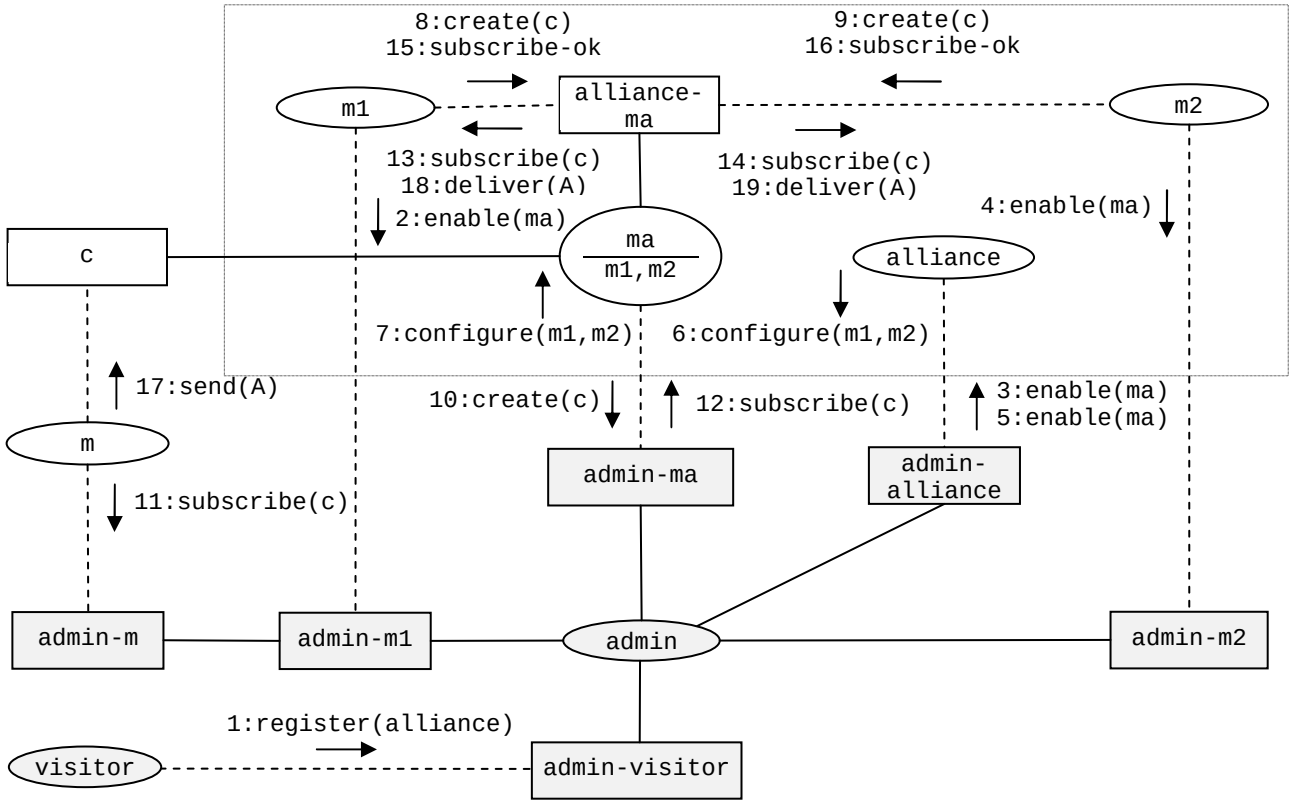
Ejemplo 39: Extensión 7 – Alianza

La Figura 53 muestra un ejemplo de estructura de comunicación para la extensión de miembro Alianza. La figura muestra el miembro `alliance` que implementa la extensión de alianza, y una alianza concreta `ma` con dos miembros `m1` y `m2`, coordinada por `ma`, y el canal `alliance-ma` propiedad de `ma` y suscripto por `m1` y `m2`. La figura muestra también el miembro externo `m` y el canal `c` propiedad de la alianza y suscripto por `m`. Por último, también se muestran los miembros predefinidos `admin` y `visitor` y todos los canales predefinidos que conectan los miembros con `admin`.

Dada esta estructura, la figura describe un escenario típico que muestra cómo la alianza es creada y cómo se comporta con respecto a miembros externos. El escenario comienza con `visitor` solicitando a `admin` la registración del miembro `alliance` – `1:register(alliance)`. Una vez registrado, los miembros `m1` y `m2` solicitan independientemente la creación de la alianza `ma` a `alliance` vía `admin` – `2:enable(ma)`, `3:enable(ma)`, `4:enable(ma)` y `5:enable(ma)`. En respuesta, `alliance` solicita a `admin` la creación del coordinador de la alianza `ma` y configura la extensión enviando las identidades de todos los miembros a `alliance` vía `admin` – `6:configure(m1,m2)` y `7:configure(m1,m2)`. A su vez, `ma` solicita a `admin` crear el canal `alliance-ma` y, una vez creado, suscribe a los miembros `m1` y `m2` al mismo. Esto completa la definición de las estructuras básicas. Supongamos que ambos miembros `m1` y `m2` quieren que la alianza cree un canal `c` y envían los mensajes `8:create(c)` y `9:create(c)`a `alliance-ma` para tal efecto. Dada la decisión por consenso, `ma` solicita a `admin` la creación de `c` – `10:create(c)` y consecuentemente, se convierte en su propietario. Seguidamente, supongamos que el miembro externo `m` envía un requerimiento a la alianza, por ejemplo a `ma` vía `admin`, para suscribirse a `c` – `11:subscribe(c)` y `12:subscribe(c)`.Como respuesta, `ma` distribuye el requerimiento a los miembros `m1` y `m2` –

13:subscribe(c) y 14:subscribe(c), recibe de vuelta las respuestas positivas – 15:subscribe-ok y 16:subscribe-ok, y suscribe m a c. Finalmente, supongamos que m envía un mensaje A a c – 17:send(A). Al recibir el mismo, ma lo envía a m1 y m2 – 18:deliver(A) y 19:deliver(A).

Figura 53: Extensión 7 – Alianza



4.5.8 Extensión 8 – Orden

La extensión Orden permite a los propietarios de canal tomar acciones conjuntas para forzar que el intercambio de mensajes sobre sus canales cumplan los requisitos de colaboración de un proceso de negocio dado, en términos del orden en el cual los mensajes son intercambiados sobre los canales y los miembros que envían tales mensajes. Una extensión Vertical, Orden tien un parámetro – descripción del proceso de negocio que hace posible determinar para cada paso de proceso el canal usado y el miembro responsable de enviar un mensaje a ese canal. Ver Tabla 15.

Tabla 15: Extensión 8 – Orden

NOMBRE	Orden	
IDENTIFICADOR	0	
TIPO	Vertical	
PARÁMETROS	proceso	descripción del proceso

La entrega, habilitación, configuración y uso de la extensión de Orden involucra lo siguiente:

- *Entrega*: El miembro visitor solicita a admin el registro del miembro order que implementa la extensión de Orden, mediante el envío del mensaje register(order).

- *Habilitación*: A fin de habilitar la extensión de Orden sobre un conjunto de canales, digamos `c1` y `c2`, sus propietarios solicitan a `order` vía `admin` la habilitación de la extensión. En respuesta, `order` solicita a `admin` crear el canal `order-c1-c2` y, una vez creado, suscribe a los propietarios de canales al mismo. El canal proporciona una conexión directa entre el coordinador de orden y los propietarios de los canales que toman parte en el proceso.
- *Configuración*: Después que la extensión es habilitada, cada propietario de canal solicita a `order` a través del canal `order-c1-c2` que configure la extensión. El parámetro contiene una descripción de proceso que, una vez configurado, permite a `order` determinar el orden en el cual los mensajes son intercambiados sobre los canales y quién es el emisor de tales mensajes; el mismo parámetro debe ser usado en todos los requerimientos.
- *Uso*: Cada vez que se envía un mensaje por un canal extendido por orden, el propietario del canal reenvía el mensaje al coordinador `order` quien verifica si el mensaje fue recibido de acuerdo con el proceso actualmente configurado, en términos del canal usado y el emisor del mensaje. Después de completar la verificación, `order` responde al propietario del canal. Si la respuesta es positiva, el propietario reenvía el mensaje a todos los suscriptores, de lo contrario notifica al emisor del mensaje vía `admin` acerca del error de orden.

El Ejemplo 40 ilustra el comportamiento de la extensión, desde la entrega y habilitación, hasta la configuración y uso.

Ejemplo 40: Extensión 8 – Orden

Las Figuras 43, 44 y 45 de la Sección 4.4.3 muestran una estructura de comunicación típica para la extensión de Orden, con los canales `c1` y `c2` extendidos por Orden, sus propietarios `m4` y `m5`, y los suscriptores `m1` y `m2` (`c1`) y `m2` y `m3` (`c2`). La figura muestra también el miembro `order` que implementa la extensión, conectado a los propietarios de los canales `m4` y `m5` por medio del canal `order-c1-c2`. Finalmente, también se muestran los miembros predefinidos `admin` y `visitor` y algunos canales predefinidos que conectan a los miembros con `admin`. La Figura 43 describe el intercambio de mensajes que conduce al desarrollo de la estructura desde la entrega, a través de la habilitación hasta la configuración. Una vez que la estructura existe, la Figura 44 muestra el intercambio de mensajes sobre el canal extendido con orden con una secuencia incorrecta de mensajes y la Figura 45 con una secuencia correcta de mensajes.

XML 9 debajo muestra un esquema XML que especifica el formato del parámetro usado por la extensión de Orden. El parámetro describe el proceso forzado por la extensión. El elemento de más alto nivel `Process` incluye el elemento `Name` de tipo `string`, una lista no vacía de elementos `Step` y el elemento `CurrentId` de tipo `integer`; el último determina el siguiente paso de proceso a ser ejecutado. Cada elemento `Step` contiene tres elementos – `Id` de tipo `integer`, y `Sender` y `Channel` de tipo `string`. Dada esta descripción de proceso, el coordinador es capaz de tomar decisiones referidas a la validez del intercambio de mensajes con respecto al proceso, reportado por los propietarios de canales. La descripción puede ser usada también por el coordinador para mantener actualizada la información acerca de los pasos actuales de proceso.

XML 9: Extensión 8 – Esquema de Parámetros de Orden

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This schema defines the parameters for G-EEG Order Extensión -->
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xg2g="http://iist.unu.edu/elsa/G-EEG/xmlUtil"
    targetNamespace="http://iist.unu.edu/elsa/G-EEG/xmlUtil"
    elementFormDefault="qualified">

  <xs:element name="Process" type="ProcessType">

  <xs:complexType name="ProcessType">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Step" minOccurs="1" maxOccurs="unbounded" type="StepType">
      <xs:element name="CurrentId" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="StepType">
    <xs:sequence>
      <xs:element name="Id" type="xs:integer" />
      <xs:element name="Sender" type="xs:string" /
      <xs:element name="Channel" type="xs:string" />
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

4.5.9 Extensión 9 – Puntualidad

La extensión de Puntualidad permite controlar si los mensajes intercambiados a través de un grupo de canales cumplen un conjunto de reglas de puntualidad que definen el tiempo máximo transcurrido entre mensajes enviados o recibidos sobre diferentes canales, y notifica a los miembros incumplidores acerca de cualquier violación. Una extensión vertical, Puntualidad tiene dos parámetros: los canales controlados y el conjunto definido de reglas de puntualidad. Ver Tabla 16.

Tabla 16: Extensión 9 – Puntualidad

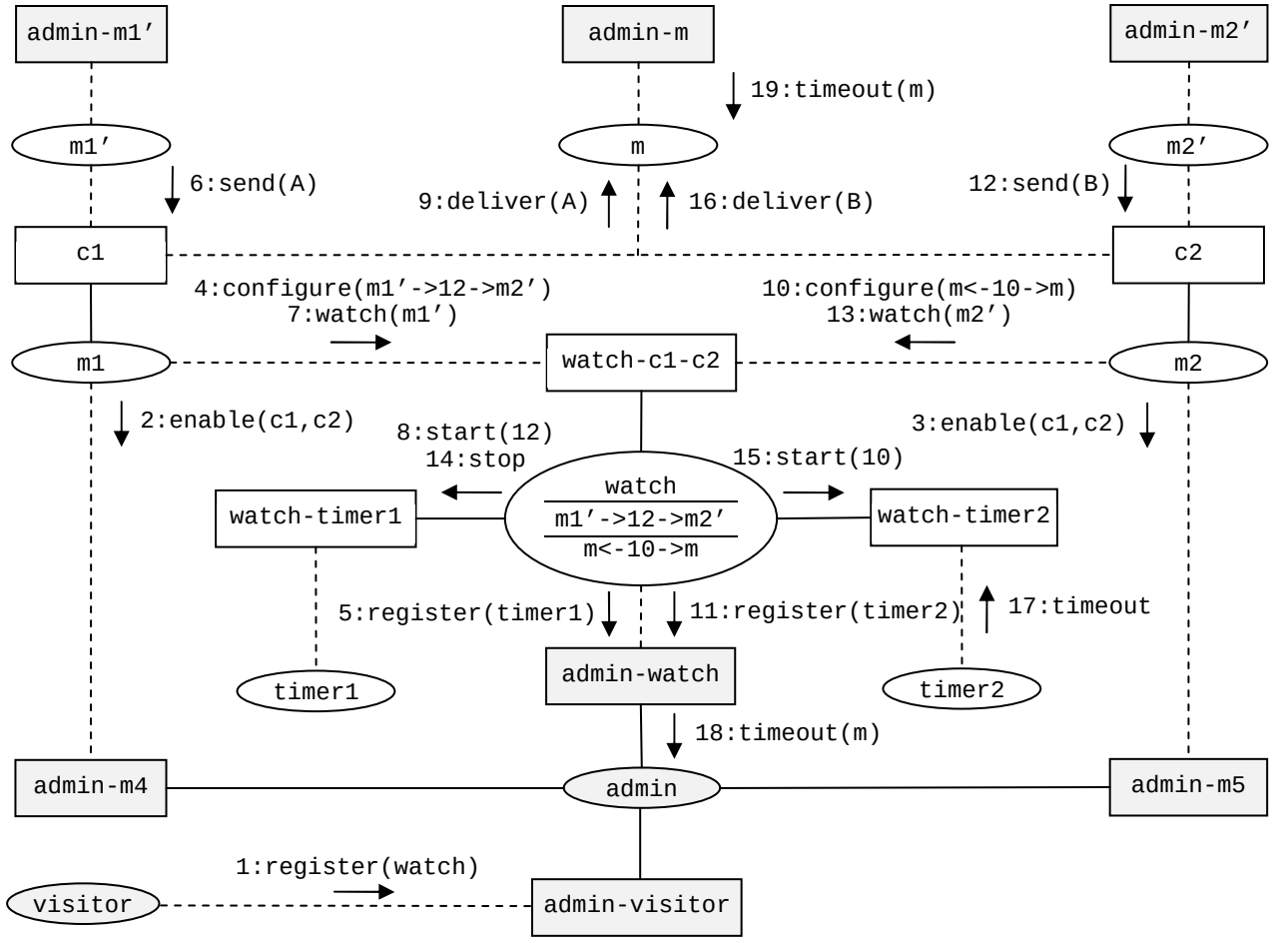
NAME	Puntualidad	
IDENTIFICADOR	I	
TIPO	Vertical	
PARÁMETROS	canales	canales a ser controlados
	estándares	conjunto de reglas de puntualidad definidas

La entrega, habilitación, configuración y uso de la extensión de Puntualidad involucra lo siguiente:

- *Entrega*: El miembro `visitor` solicita a `admin` el registro del miembro `watch` que implementa la extensión Puntualidad enviando el mensaje `register(watch)` por medio del canal `visitor-admin`.
- *Habilitación*: A fin de habilitar la extensión de Puntualidad sobre un grupo de canales, digamos `c1` y `c2`, los propietarios de tales canales deben solicitar a `watch` vía `admin` que habilite la extensión. En respuesta, `watch` solicita a `admin` la creación del canal `watch-c1-c2` y, una vez que el canal es creado, suscribe a los propietarios.
- *Configuración*: Después que la extensión es habilitada, cada propietario de canal puede enviar un conjunto de reglas de puntualidad a `watch` por medio del canal `watch-c1-c2` para determinar el tiempo máximo para que los propietarios o suscriptores de los canales controlados envien mensajes, en respuesta a mensajes previamente enviados o recibidos por el mismo. Por ejemplo, `m1'->11->m2'` especifica que después que `m1'` ha enviado un mensaje a cualquier canal controlado, `m2'` debería enviar otro mensaje a cualquiera de tales canales dentro de `11` unidades de tiempo. Como otro ejemplo, `m<-10->m` especifica que después de que `m` recibió un mensaje de cualquier canal controlado, debe enviar otro mensaje a tal canal dentro de `10` unidades de tiempo. Cada vez que tales reglas son configuradas por un propietario, `watch` solicita a `admin` el registro de un miembro `timer` y el canal `watch-time` para conectarse con él. A través de ese canal, `watch` será capaz de arrancar y parar a `timer` sobre el recibo de cualquier evento de disparo para una regla de puntualidad dada. Adicionalmente, a través de dicho canal, `timer` enviará mensajes de timeout a `watch` cuando el tiempo haya expirado antes de que sea detenido.
- *Uso*: Cada vez que se envía un mensaje a través de un canal controlado, la información es enviada a `watch`, el cual verifica la aplicabilidad de alguna regla de puntualidad. Si ninguna regla aplica, el propietario simplemente entrega el mensaje a todos los suscriptores. De lo contrario, si el evento dispara una nueva instancia de la regla de puntualidad, `watch` solicita el arranque del `timer` correspondiente para empezar la cuenta regresiva, pero si el evento aplica a una instancia existente de la regla con un `timer` en ejecución, `watch` le solicita al `timer` que se detenga. En ambos casos el propietario envía los mensajes a todos los suscriptores. Si algún `timer` en ejecución no es detenido por `watch` antes de que transcurra el tiempo especificado, significa que uno de los miembros falló en mandar un mensaje a tiempo. En este caso, `timer` envía un mensaje de time-out a `watch`, el cual a su turno reenvía el mensaje al miembro incumplidor vía `admin`.

Una estructura y comportamiento típicos de la extensión de Puntualidad se explica en el Ejemplo 41 y se muestra en la Figura 54 debajo. El comportamiento cubre todas las etapas, desde la entrega, a través de la habilitación y configuración, hasta el uso.

Figura 54: Extensión 9 – Puntualidad



Ejemplo 41: Extensión 9 – Puntualidad

La Figura 54 muestra una estructura típica de comunicación para la extensión de Puntualidad, habilitada sobre los canales `c1` y `c2`, que son propiedad de `m1` y `m2`, y suscriptos por `m1'` y `m` (`c1`) y `m2'` y `m` (`c2`). La figura muestra también al miembro `watch` que implementa la extensión, conectado a los propietarios de canal por medio del canal `watch-c1-c2`. El manejador de extensión `watch` está configurado con dos reglas de puntualidad `m1'->12->m2'` y `m<-10->m` que son controladas por los correspondientes temporizadores `timer1` y `timer2`. Los temporizadores están conectados a `watch` a través de los canales `watch-timer1` y `watch-timer2`. Asimismo, se muestran los miembros predefinidos `admin` y `visitor` y algunos canales predefinidos que conectan miembros con `admin`.

Dada esta estructura, la figura describe un escenario típico para el intercambio de mensajes sobre los canales `c1` y `c2` extendidos con Puntualidad. El escenario comienza con `visitor` solicitando a `admin` el registro del miembro `watch` – `1:register(watch)`. Una vez que el miembro es registrado, los propietarios `m1` y `m2` solicitan a `watch` vía `admin` que habilite la extensión de Puntualidad sobre los canales `c1` y `c2` – `2:enable(c1,c2)` y `3:enable(c1,c2)`. Con el canal `watch-c1-c2` ya creado, el propietario `m1` solicita a `watch` que configure la regla de puntualidad – `4:configure(m1'->12->m2')`. En respuesta, `watch` solicita a `admin` crear un temporizador para esa regla – `5:create(timer1)`, crear el canal `watch-timer1`, y suscribir `timer1` a ese canal.

Supongamos que el miembro m1’ envía un mensaje A a c1 – 6:send(A). El propietario m1 informa a watch – 7:watch(m1’) el cual, reconociendo que la regla m1’->12->m2’ aplica, solicita a timer1 que comience la cuenta regresiva – 8:start(12). Mientras tanto, el propietario m1 envía el mensaje al suscriptor restante m de c1 – 9:deliver(A). Siguiendo, supongamos que el propietario del canal c2 – m2 solicita a watch la configuración de otra regla de puntualidad – 10:configure(m<-10->m). En respuesta, watch solicita a admin que registre timer2 - 11:register(timer2), cree el canal watch-timer2, y suscriba timer2 a este canal. Con timer1 todavía contando, supongamos que el miembro m2’ envía un mensaje B a c2 antes de que expire timer1 – 12:send(B). El propietario le informa a watch – 13:watch(m2’) quien, reconociendo que ambas reglas aplican (m2’ envía un mensaje, m lo recibe), solicita a timer1 que pare – 14:stop y timer2 que arranque – 15:start(10). Mientras tanto, m5 envía el mensaje B al suscriptor m2 – 16:deliver(B). Finalmente, con timer2 contando, supongamos que m falla en cumplir el límite de tiempo para enviar un mensaje después de haber recibido uno. Con timer2 expirando, informa a watch – 17:timeout, quien a su vez envía una alarma de time-out a m vía admin – 18:timeout(m) y 19:timeout(m).

4.5.10 Extensión 10 – Composición

La Extensión de Composición permite construir canales complejos a partir de canales existentes. Cinco operaciones de composición son definidas: (1) Vinculación – contando con un canal origen y uno destino, la operación une el punto de salida del canal origen con el punto de entrada del canal destino, de tal modo los mensajes recibidos por el canal origen son reenviados de inmediato al canal destino; (2) Distribución – contando con un canal origen y varios canales destino, la operación une el punto de salida del canal origen con el punto de entrada de todos los canales destino, de tal modo los mensajes recibidos por el canal origen son reenviados a todos los canales destinos; (3) Unión – contando con varios canales de origen y un canal destino, la operación junta el punto de salida de todos los canales origen al punto de entrada del canal destino, así los mensajes recibidos por los canales orígenes son reenviados al canal destino; (4) Filtrado – contando con un canal origen y un canal destino, la operación inserta un filtro entre el punto de salida del canal origen y el punto de entrada del canal destino para decidir, basado en el contenido del mensaje y especificaciones de filtrado, si los mensajes recibidos en el canal origen deben ser enviados al canal destino o descartados; y (5) Ruteo – contando con un canal de origen y varios canales de destino, la operación inserta un ruteador entre el canal origen y los canales de destino para decidir, basado en el contenido del mensaje y especificaciones de ruteo, a qué canal de destino deben ser reenviados los mensajes recibidos por el canal de origen. Composición es un extensión vertical con un solo parámetro – la operación aplicada para componer los canales (Vinculación, Distribución, Unión, Filtrado o Ruteo). Dependiendo de la operación, el parámetro determina el conjunto de canales de origen y destino, y las especificaciones de filtrado o ruteo. Ver Tabla 17 debajo.

Tabla 17: Extensión 10 – Composición

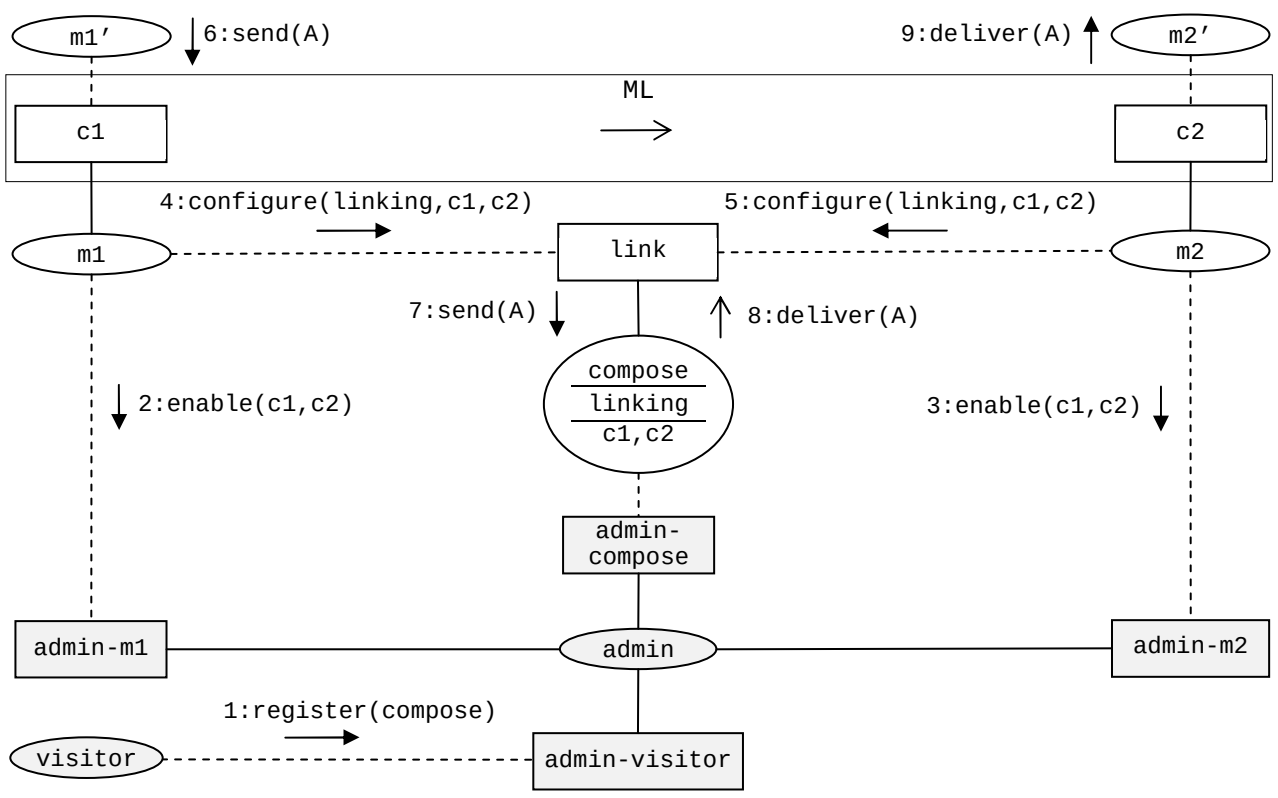
NOMBRE	Composición		
IDENTIFICADOR	M		
	ML	Vinculación	
	MS	Distribución	
	MJ	Unión	
	MF	Filtrado	
	MR	Ruteo	
TIPO	Vertical		
PARÁMETROS	operación	vinculación, distribución, unión, filtrado, ruteo	
	op.específica	vinculación	un canal origen, un canal destino
		distribución	un canal origen, varios canales destino
		unión	varios canales origen, un canal destino
		filtrado	un canal origen, un canal destino, especificación de filtrado
		ruteo	un canal origen, varios canales destino, especificación de ruteo

La entrega, habilitación, configuración y uso de la extensión de Composición involucra lo siguiente:

- *Entrega*: El miembro `visitor` solicita a `admin` el registro del miembro `compose` que implementa la extensión de Composición, enviando un mensaje `register(compose)` por medio de `visitor-admin`.
- *Habilitación*: A fin de habilitar la extensión de Composición sobre un grupo de canales, digamos `c1` y `c2`, los propietarios de todos los canales involucrados deben solicitar a `compose` vía `admin` que habilite la extensión. En respuesta, `compose` solicita a `admin` que cree un canal `link` y, una vez creado, suscribe a todos los propietarios a él.
- *Configuración*: Después de habilitada la extensión, los propietarios la configuran enviando un mensaje a `compose` a través del canal `link`, especificando la operación solicitada y los parámetros correspondientes, tales como un conjunto de canales de origen y destino, y especificaciones de filtrado o ruteo. Todos los propietarios de los canales involucrados deben enviar un parámetro idéntico a `compose` para que sea configurada la extensión.
- *Uso*: Cada vez que se recibe un mensaje por uno de los canales de origen, es enviado por el propietario al canal `link`, y recibido por `compose` para acciones posteriores dependiendo de la operación aplicada. En particular: (1) vinculación – `compose` reenvía el mensaje al propietario del canal destino; (2) distribución – `compose` reenvía el mensaje a los propietarios de los canales destinos; (3) unión – `compose` reenvía el mensaje al propietario del canal destino; (4) filtrado – `compose` decide, basado en el contenido del mensaje y las especificaciones de filtrado, si el mensaje debe ser enviado al propietario del canal destino o descartado, e implementa la decisión de manera acorde; y (5) ruteo – `compose` decide, basado en el contenido del mensaje y especificaciones de ruteo, a qué canal de destino debería ser enviado el mensaje, de haber alguno, e implementa la decisión de manera acorde.

Una estructura y comportamiento típicos de la extensión de Composición configurada con el operador de Vinculación se explica en el Ejemplo 42 y se muestra en la Figura 55 debajo. El comportamiento cubre todas las etapas, desde la entrega, a través de la habilitación y configuración, hasta el uso.

Figura 55: Extensión 10 – Composición por Vinculación



Ejemplo 42: Extensión 10 – Composición por Vinculación

La Figura 55 debajo muestra una estructura de comunicación típica para la extensión de Composición configurada con el operador de Vinculación. La estructura muestra el canal de origen `c1` propiedad de `m1` y suscripto por `m1'`, y

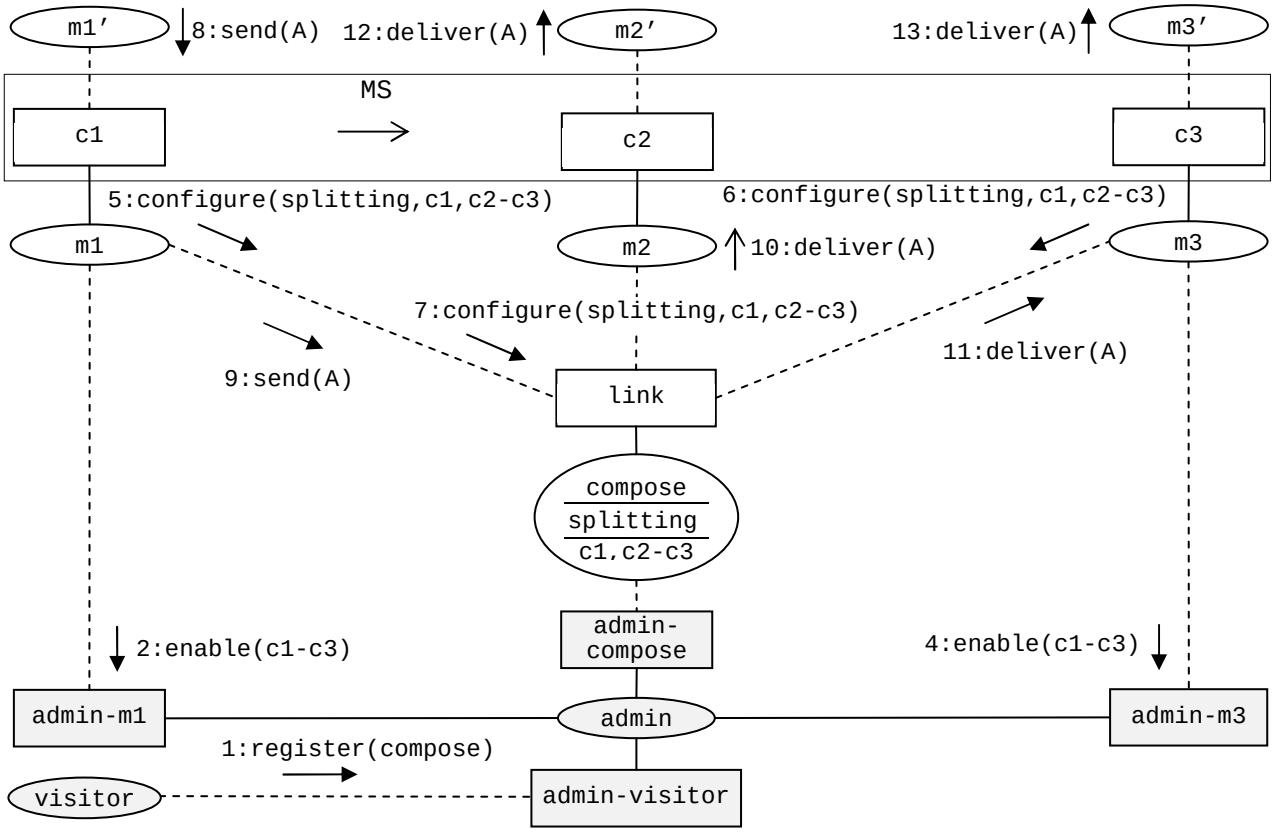
el canal destino c2 propiedad de m2 y suscripto por m2'. La figura muestra también el miembro compose que implementa la extensión, con el canal link propiedad de compose y suscripto por los propietarios de canal m1 y m2. Finalmente, se muestran también los miembros predefinidos admin y visitor y algunos canales predefinidos que conectan miembros con admin.

Dada esta estructura, la figura describe un escenario típico de intercambio de mensajes sobre los canales vinculados c1 y c2. El escenario comienza con visitor solicitando a admin la registración del miembro compose – 1:register(compose). Una vez registrado, m1 y m2 solicitan a compose vía admin la habilitación de la extensión de Composición – 2:enable(c1,c2) y 3:enable(c1,c2). Esto da como resultado el canal link con compose como propietario y m1 y m2 como suscriptores. Luego, m1 y m2 configuran la extensión solicitando a compose que vincule el canal origen c1 al canal destino c2 – 4:configure(linking,c1,c2) y 5:configure(linking, c1,c2). Esto completa las estructuras básicas.

Supongamos que m1' envía un mensaje A a c1 – 6:send(A). El mensaje es enviado por el propietario m1 al canal link – 7:send(A) y, luego de recibirlo, compose lo reenvía de vuelta a link y a al otro propietario m2 – 8:deliver(A). A su turno, m2 envía el mensaje al suscriptor del canal destino m2' – 9:deliver(A).

Una estructura y comportamiento típicos de la extensión de Composición configurada con el operador de Distribución se explica en el Ejemplo 43 y se muestra en la Figura 56 debajo. El comportamiento cubre todas las etapas, desde la entrega, a través de la habilitación y configuración, hasta el uso.

Figura 56: Extensión 10 – Composición por Separación



Ejemplo 43: Extensión 10 – Composición por Distribución

La Figura 56 debajo muestra la estructura de comunicación típica para la extensión de Composición configurada con el operador de Distribución. La estructura muestra el canal de origen c1 propiedad de m1 y suscripto por m1', y dos

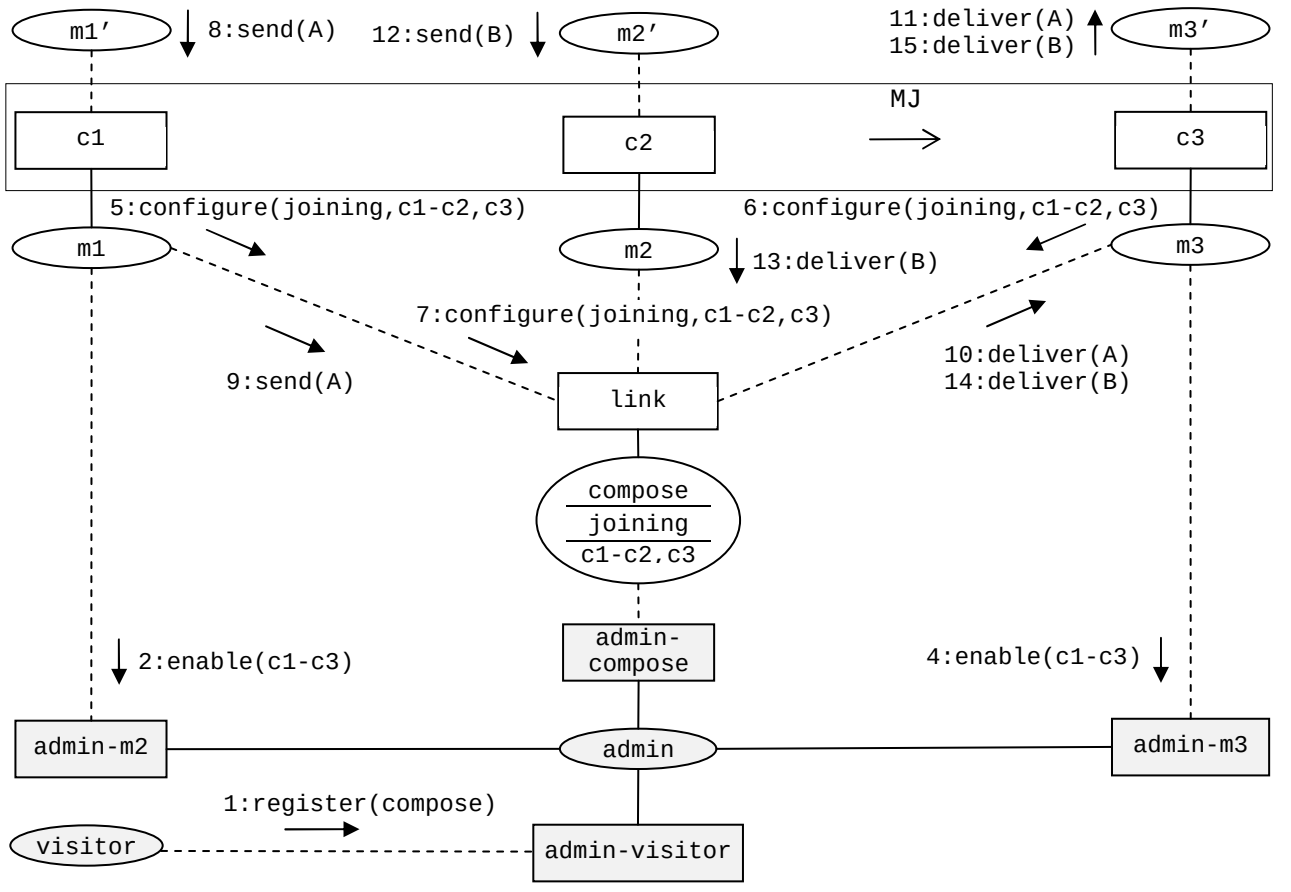
canales de destino c2 propiedad de m2 y suscriptos por m2' y c3 propiedad de m3 y suscriptos por m3'. La figura muestra también el miembro compose que implementa la extensión, con el canal link propiedad de compose y suscripto por los propietarios de los canales m1, m2 y m3. Finalmente, también se muestran los miembros predefinidos admin y visitor y algunos canales predefinidos que conectan miembros con admin.

Dada esta estructura, la figura describe un escenario típico de intercambio de mensajes sobre los canales que distribuyen mensajes c1 (origen) y c2 y c3 (destinos). El escenario comienza con visitor solicitando a admin la registración del miembro compose – 1:register(compose). Una vez registrados, m1, m2 y m3 solicitan a compose vía admin que habilite la extensión de Composición – 2:enable(c1-c3), 3:enable(c1-c3) (no mostrada) y 4:enable(c1-c3). Esto resulta en el canal link con compose como propietario y m1, m2 y m3 como suscriptores. A continuación, m1, m2 y m3 solicitan a compose que vincule el canal de origen c1 a los canales destino c2 y c3 – 5:configure(splitting,c1,c2-c3), 6:configure(splitting,c1,c2-c3) y 7:configure(splitting, c1,c2-c3). Esto completa las estructuras básicas.

Supongamos que m1' envía un mensaje A a c1 – 8:send(A). El mensaje es enviado por el propietario m1 al canal link – 9:send(A) y, a su recepción, compose lo reenvía de vuelta a través de link a m2 – 10:deliver(A) y m3 - 11:deliver(A). A su vez, m2 entrega el mensaje al suscriptor m2' de c2 – 12:deliver(A) y m3 entrega el mensaje al suscriptor m3' de c3 – 13:deliver(A).

Una estructura y comportamiento típicos de la extensión de Composición configurada con el operador de Unión se explica en el Ejemplo 44 y se muestra en la Figura 57 debajo. El comportamiento cubre todas las etapas, desde la entrega, a través de la habilitación y configuración, hasta el uso.

Figura 57: Extensión 10 – Composición por Unión



Ejemplo 44: Extensión 10 – Composición por Unión

La Figura 57 muestra la estructura de comunicación típica para la extensión de Composición configurada con el operador de Unión. La estructura muestra dos canales de origen c1 propiedad de m1 y suscripto por m1', y c2 propiedad de m2 y suscripto por m2', y el canal de destino c3 propiedad de m3 y suscripto por m3'. La figura muestra también el miembro compose que implementa la extensión, con link propiedad de compose y suscripto por m1, m2 y m3. Finalmente, se muetran también los miembros predefinidos admin y visitor y algunos canales de administración.

Excepto por el rol del canal, las estructuras para unión y distribución son idénticas. Dada esta estructura, la figura describe un escenario típico para el intercambio de mensajes sobre los canales unidos c1 y c2 (orígenes), y c3 (destino). El escenario comienza con visitor solicitando a admin la registración del miembro compose – 1:register(compose), seguido por m1, m2 y m3 solicitando a compose vía admin que habilite la extensión – 2:enable(c1-c3), 3:enable(c1-c3) (no mostrado) y 4:enable(c1-c3). Esto resulta en el canal link con compose como propietario y m1, m2 y m3 como suscriptores. Luego, m1, m2 y m3 solicitan a compose que vincule los canales origen c1 y c2 al canal destino c3 – 5:configure(joining,c1-c2,c3), 6:configure(joining,c1-c2,c3) y 7:configure(joining,c1-c2,c3). Esto completa las estructuras básicas.

Supongamos que m1' envía un mensaje A a c1 – 8:send(A). El mensaje es entregado por el propietario m1 a compose – 9:send(A), compose lo reenvía al propietario del canal destino m3 – 10:deliver(A), y m3 envía el mensaje a suscriptor m3' – 11:deliver(A). Luego, supongamos que m2' envía un mensaje B a c2 – 12:send(B). El mensaje es enviado por el propietario m2 a compose – 13:deliver(B), compose lo entrega al propietario m3 del canal destino – 14:deliver(B), y m3 entrega el mensaje al suscriptor m3' – 15:deliver(B).

El Ejemplo 45 debajo ilustra el comportamiento de la extensión de Composición para el operador de Filtrado.

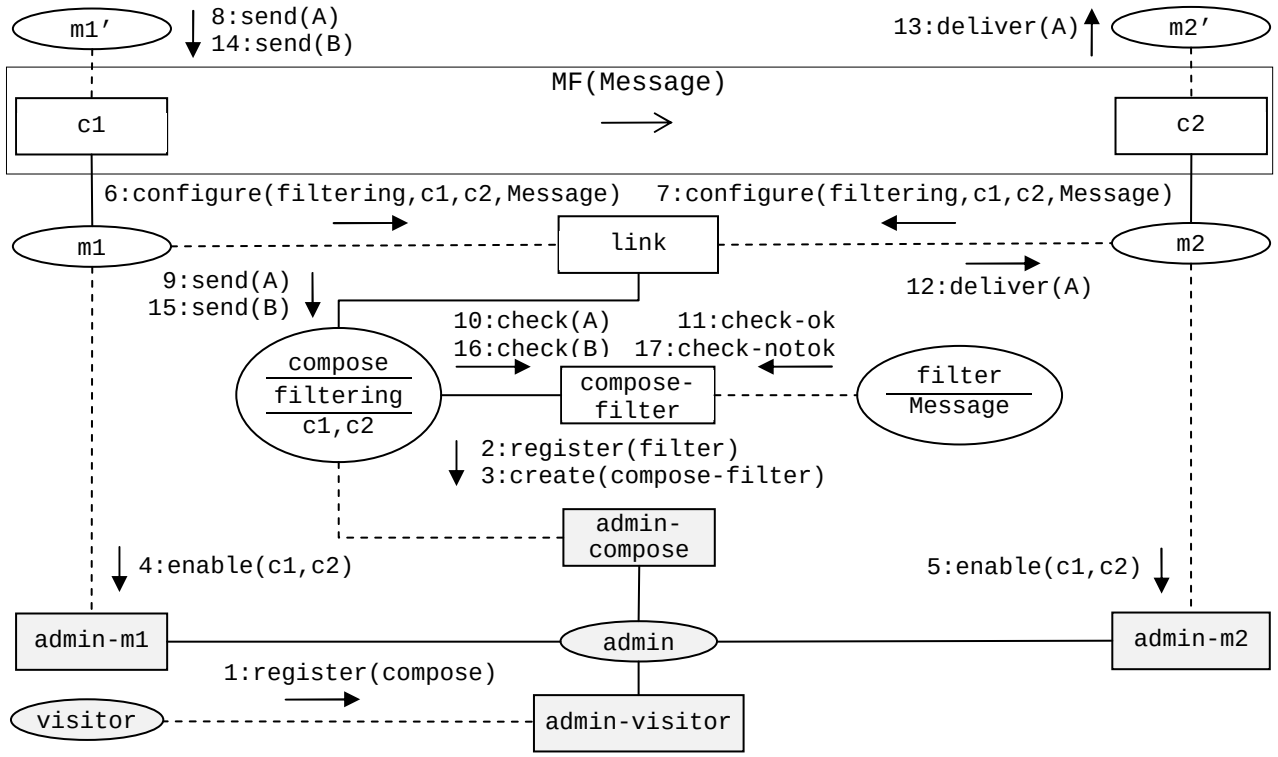
Ejemplo 45: Extensión 10 – Composición por Filtrado

La Figura 58 debajo muestra la estructura de comunicación típica para la extensión de Composición configurada con el operador de Filtrado. La estructura muestra el canal c1 propiedad de m1 y suscripto por m1', y el canal destino c2 propiedad de m2 y suscripto por m2'. La figura muestra también el miembro compose que implementa la extensión, con el canal link propiedad de compose y suscripto por los propietarios de los canales m1 y m2, y el miembro filter conectado a compose a través del canal compose-filter. Finalmente, también se muestran los miembros predefinidos admin y visitor y algunos canales predefinidos que conectan miembros con admin.

Dada esta estructura, la figura describe un escenario típico de mensajería sobre los canales filtrados c1 y c2. El escenario comienza con visitor solicitando a admin el registro del miembro compose – 1:register(compose) el cual, una vez registrado, solicita a admin el registro del miembro filter y la creación del canal compose-filter – 2:register(filter) y 3:create(compose-filter). Luego, m1 y m2 solicitan a compose vía admin que habilite la extensión de Composición – 4:enable(c1,c2) y 5:enable(c1,c2), resultando en el canal link propiedad de compose y suscripto por m1 y m2. Luego, m1 y m2 solicitan a compose la configuración de la extensión de filtrado – 6:configure(filtering,c1,c2,Message) y 7:configure(filtering,c1,c2, Message) con el esquema Message (Sección 4.4.2) aplicado a los mensajes filtrados. Esto completa la configuración básica.

Supongamos que m1' envía un mensaje A, válido con respecto al tipo Message, a c1 – 8:send(A). El mensaje es entregado por el propietario m1 al canal link – 9:send(A) y, luego de recibirlo, compose solicita a filter la verificación del formato de mensaje con respecto a Message – 10:check(A). Recibida una respuesta positiva – 11:check-ok, compose reenvía A al propietario m2 del canal destino – 12:deliver(A) el cual lo entrega al suscriptor m2' – 13:deliver(A). Supongamos que m1' envía otro mensaje B a c1 – 14:send(B), pero B es inválido con respecto a Message. El mensaje es reenviado por el propietario m1 a compose vía link – 15:send(B), el cual solicita a filter la verificación del formato de mensaje con respecto a Message – 16:check(B). Como la verificación no es satisfactoria – 17:check-notok, B no es reenviado a c2.

Figura 58: Extensión 10 – Composición por Filtrado



El Ejemplo 46 debajo ilustra el comportamiento de la extensión de Composición para el operador de Ruteo.

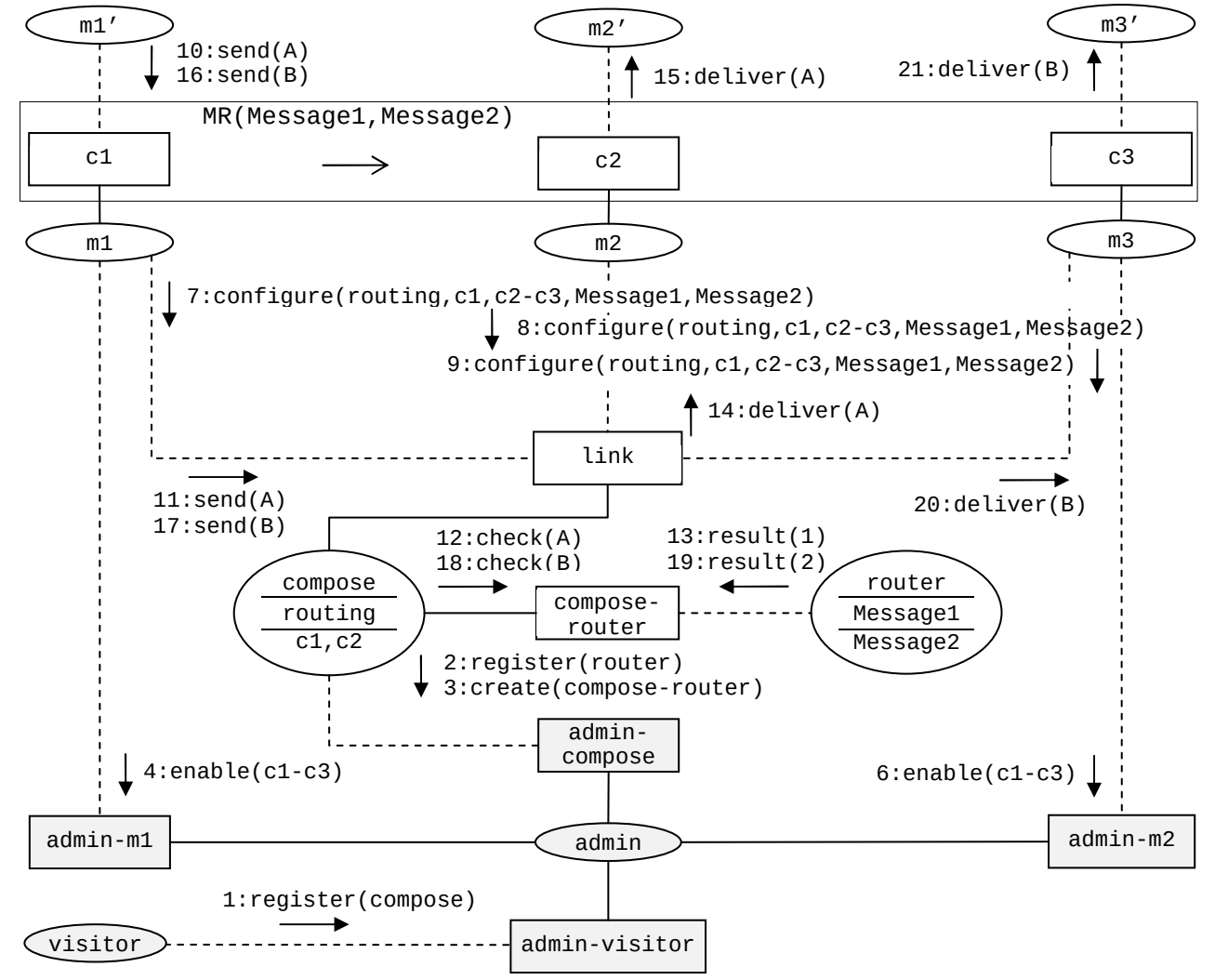
Ejemplo 46: Extensión 10 – Composición por Ruteo

La Figura 59 muestra la estructura de comunicación típica para la extensión de Composición configurada con el operador de Ruteo. La estructura muestra el canal `c1` propiedad de `m1` y suscripto por `m1'`, y dos canales destino `c2` propiedad de `m2` y suscripto por `m2'` y `c3` propiedad de `m3` y suscripto por `m3'`. La figura muestra también el miembro `compose` que implementa la extensión, con el canal `link` propiedad de `compose` y suscripto por `m1`, `m2` y `m3`, y el miembro `router` conectado a `compose` a través del canal `compose-router`. Finalmente, se muetran los miembros predefinidos `admin` y `visitor` y algunos canales predefinidos que conectan miembros con `admin`.

La figura describe un escenario típico de mensajería sobre los canales ruteados `c1` (origen) y `c2` y `c3` (destinos). Comienza con `visitor` solicitando a `admin` el registro del miembro `compose` – `1:register(compose)`, quien luego solicita a `admin` el registro del miembro `router` y la creación del canal `compose-router` – `2:register(router)` y `3:create(compose-router)`. Siguiendo, `m1`, `m2` y `m3` solicitan a `compose` que habilite la extensión de Composición – `4:enable(c1-c3)`, `5:enable(c1-c3)` (no se muestra) y `6:enable(c1-c3)`, produciendo el canal `link` con `compose` como propietario y `m1`, `m2` y `m3` como suscriptores. Luego, `m1`, `m2` y `m3` solicitan a `compose` la configuración de la extensión con `Message1` y `Message2` indicando el formato de mensajes que deben enviarse a `c2` y `c3` respectivamente - `7:configure(routing,c1,c2-c3,Message1, Message2)`, `8:configure(routing, c1,c2-c3,Message1,Message2)`, `9:configure(routing,c1,c2-c3,Message1, Message2)`.

Supongamos que `m1'` envía un mensaje `A`, valido con respecto a `Message1`, a `c1` – `10:send(A)`. El mensaje es enviado por `m1` al canal `link` – `11:send(A)` y, al recibirlo, `compose` solicita a `router` la verifiación de– `12:check(A)`. `router` responde que `A` es válido con respecto a `Message1` - `13:result(1)`, por lo tanto `compose` reenvía `A` al propietario `m2` de `c2` – `14:deliver(A)` el cual lo envía al suscriptor `m2'` – `15:deliver(A)`. Luego, supongamos que `m1'` envía un mensaje `B`, valido con respecto a `Message2`, a `c1` – `16:send(B)`. El mensaje es enviado por `m1` a `link` – `17:send(B)` y, sobre el recibo, `compose` solicita una verificación de formato a `router` – `18:check(B)`. `router` responde que `B` es válido con respecto a `Message2` - `19:result(2)`, por lo tanto `compose` reenvía `B` al propietario `m3` de `c3` – `20:deliver(B)` el cual lo envía al suscriptor `m3'` – `21:deliver(B)`.

Figura 59: Extensión 10 – Composición por Ruteo



4.6 Formalización

Una visión general de las extensiones de mensajería en la Sección 4.5 demostró que G-EEG, si bien está basado en unos pocos conceptos simples como Miembro, Mensaje, Canal y Extensión, permite la expresión de una gama de estructuras y comportamientos de mensajería útiles. La creciente complejidad de tales comportamientos, cuando se representa exclusivamente en términos de los conceptos básicos, amerita la búsqueda de abstracción, a través de la cual los comportamientos de mensajería complejos (tales como las extensiones de la Sección 4.5) pueden ser expresados, empaquetados y puestos a disponibilidad para su reuso en diferentes contextos de aplicación, sin tener que preocuparse por su complejidad interna. Asimismo, la búsqueda de abstracción y su opuesto - refinamiento, puede ser la base de un enfoque de desarrollo sistemático, a través del cual las descripciones abstractas son gradualmente refinadas en descripciones más complejas con correctitud garantizada entre los distintos niveles de abstracción basada en verificación y demostraciones. Sin embargo, la pre-condición para la abstracción, el refinamiento o un enfoque de desarrollo basado en refinamiento es la presencia de un modelo para G-EEG que pueda expresar estructuras y comportamientos de mensajería con semántica y sintaxis precisas, si es posible formales.

Esta sección introduce tal modelo. Expresado en la notación formal de RSL - RAISE Specification Language [RLG92], el modelo es desarrollado a través de una serie de refinamientos con modelos más complejos y expresivos construidos sistemáticamente sobre modelos previos. Mientras que diferentes modelos en la serie están estrechamente relacionados entre sí en términos de los conceptos definidos además de su estructura y comportamiento, no se establece ni se clama ninguna relación formal entre ellos.

La serie consta de los siguientes modelos:

1) MODELO 1 – Mensajería directa entre miembros;
2) MODELO 2 – Introducción de canales para facilitar en intercambio de mensajes;
3) MODELO 3 – Introducción de roles diferenciados para miembros y canales;
4) MODELO 4 – Transformación del estado del Gateway para ser distribuido entre los miembros;
5) MODELO 5 – Introducción de tipos de mensajes y ejecución de todas las operaciones del Gateway a través de intercambio de mensajes;
6) MODELO 6 – Reenvío de mensajes a dueños de canales para procesarlos y diseminarlos a suscriptores de canales;
7) MODELO 7 – Procesamiento de mensajes recibidos por los miembros a través de acciones programables para diferentes tipos de mensajes.

Los modelos son presentados en las Secciones 4.6.1 a 4.6.7 Cada modelo consta de un grupo de módulos RSL (esquemas) mostrados detalladamente entre los Apéndices A1 a A7.

4.6.1 MODELO 1 – Mensajería Directa

El MODELO 1 consta de seis esquemas que introducen identificadores (ID), mensajes (MESSAGE), cola de mensajes (QUEUE), miembros (MEMBER), estados (STATE), y el Gateway (GATEWAY), incluidos en el Apéndice A.1 y referidos de la siguiente manera:

```
scheme ID = ...
scheme MESSAGE(I:ID) = ...
scheme QUEUE(I:ID, M:MESSAGE(I)) = ...
scheme MEMBER(I:ID, M:MESSAGE(I)) = extend QUEUE(I, M) with ...
scheme STATE(I:ID, M:MESSAGE(I), E:MEMBER(I,M)) = ...
scheme GATEWAY(I:ID, M:MESSAGE(I), E:MEMBER(I,M), S:STATE(I,M,E)) = ...
```

Siguiendo la definición de identificadores de miembros MemberId como un tipo abstracto en el esquema ID, el esquema MESSAGE introduce el tipo Message junto con dos funciones para este tipo - sender para obtener al remitente de un mensaje (MemberId) y recipients para obtener a todos los receptores de un mensaje (conjunto de MemberId):

```
type
  MemberId
```

```
type
  Message
value
  sender: Message -> I.MemberId,
  recepients: Message -> I.MemberId-set
```

Una cola es definida en QUEUE como el tipo Queue que contiene listas de mensajes. Inicialmente vacía - init, se puede controlar si una cola está vacía - isEmpty. Los mensajes pueden ser insertados al final de la cola a través de la función enqueue y pueden ser removidos de la cabecera a través de la función dequeue, con una precondición controlando que la cola no esté vacía:

```
type
  Queue = M.Message-list

value
  init: Queue = <..>,

  isEmpty: Queue -> Bool
  isEmpty(q) is q = <..>
```

```
value
  enqueue: M.Message >< Queue -> Queue
  enqueue(m, q) is q ^ <.m.>,

  dequeue: Queue -~-> M.Message >< Queue
  dequeue(q) is (hd q, tl q) pre ~isEmpty(q)
```

El esquema MIEMBRO define el tipo abstracto Member. Cada miembro contiene dos colas o bnadeja de mensajes de entrada y salida - inbox and outbox. Inicialmente vacía**s**, ambas colas pueden ser revisadas para ver si están vacías (isEmptyInbox y isEmptyOutbox) y si tienen mensajes (isInInbox y isInOutbox). Los mensajes se insertan y remueven desde inbox y outbox a través de las funciones correspondientes enqueueInbox, enqueueOutbox, dequeueInbox y dequeueOutbox, de la siguiente manera:

```
type
  Member

value
  inbox: Member -> Queue,
  outbox: Member -> Queue

value
  init: Member :-
  inbox(init) = init /\
  outbox(init) = init
```

```
value
  isEmptyInbox: Member -> Bool
  isEmptyInbox(e) is isEmpty(inbox(e)),

  isInInbox: M.Message >< Member -> Bool
  isInInbox(m, e) is m isin elems inbox(e),

  enqueueInbox: M.Message >< Member -> Member
  enqueueInbox(m, e) as e' post
    inbox(e') = enqueue(m, inbox(e)) /\
    outbox(e') = outbox(e),

  dequeueInbox: Member -~-> (M.Message >< Member)
  dequeueInbox(e) as (m, e') post
    let (m', q) = dequeue(inbox(e)) in
      m = m' /\ inbox(e') = q /\ outbox(e') = outbox(e)
    end pre ~isEmptyInbox(e)
```

El esquema STATE define el estado del Gateway (`State`) como un mapeo bien formado desde los identificadores (`MemberId`) a los miembros (`Member`). Inicialmente vacío - `init`, el estado puede ser controlado para verificar la existencia de un miembro (`memberExists`) o un grupo de miembros (`membersExist`), mientras que `getMember` puede utilizarse para obtener un miembro identificado de un estado.

```
Type
  State' = I.MemberId -m-> E.Member,
  State = {| s: State' :- iswf(s) |}

value
  init: State' = []
```

```
value
  memberExists: I.MemberId >< State' -> Bool
  memberExists(i, s) is i isin dom s,

  membersExist: I.MemberId-set >< State' -> Bool
  membersExist(ids, s) is ids <<= dom s,

  getMember: I.MemberId >< State' -~-> E.Member
  getMember(i, s) is s(i) pre memberExists(i, s)
```

Adicionalmente, el esquema STATE contiene un grupo de funciones para verificar si varios elementos del estado están bien formados. Para que un mensaje esté bien formado en un estado determinado, su remitente y todos los receptores deben ser miembros registrados en el estado (`iswfMessage`). Una cola está bien formada (`iswfQueue`) si todos sus mensajes están bien formados. Un miembro está bien formado (`iswfMember`) siempre y cuando su colas de mensajes de entrada y de salida estén bien formadas y por cada mensaje `m` en la bandeja de entrada de un miembro `i`, el remitente de `m` debe ser `i`, y por cada mensaje `m` en la bandeja de salida de `i`, `i` debe pertenecer al grupo de receptores de `m`. Finalmente, para que un estado esté bien formado (`iswf`), todos los miembros registrados deben estar bien formados también.

```
value
  iswfMessage: M.Message >< State' -> Bool
  iswfMessage(m, s) is
    memberExists(M.sender(m), s) /\
    (all i: I.MemberId :- i isin M.recepients(m) => memberExists(i, s)),

  iswfQueue: E.Queue >< State' -> Bool
  iswfQueue(q, s) is q = <..> \/ iswfMessage(hd q, s) /\ iswfQueue(tl q, s),

  iswfMember: I.MemberId >< State' -> Bool
  iswfMember(i, s) is
    let e = getMember(i, s) in
      iswfQueue(E.inbox(e), s) /\ iswfQueue(E.outbox(e), s) /\
      (all m: M.Message :-
        (E.isInInbox(m, e) => M.sender(m) = i) /\
        (E.isInOutbox(m, e) => i isin M.recepients(m))
      )
    end,

  iswf: State' -> Bool
  iswf(s) is (all i: I.MemberId :- i isin dom s => iswfMember(i, s))
```

Para poder cambiar el estado, tres funciones parciales son introducidas a STATE para agregar, borrar y modificar miembros:

```
value
  addMember: I.MemberId >< E.Member >< State -~-> State
  addMember(i, e, s) is s union [ i +> e ] pre ~memberExists(i, s),

  deleteMember: I.MemberId >< State -~-> State
  deleteMember(i, s) is s \ {i} pre memberExists(i, s),

  modifyMember: I.MemberId >< E.Member >< State -~-> State
  modifyMember(i, e, s) is s !! [ i +> e ] pre memberExists(i, s)
```

Definido por encima de STATE, GATEWAY especifica cinco funciones principales para registrar y des-registrar miembros, para enviar y recibir mensajes por parte de miembros, y transferir mensajes entre ellos. En particular, la registración (`register`) agrega un miembro no inscripto al estado, mientras que des-registración (`unregister`) elimina del estado a un miembro inscripto:

```
value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is S.addMember(i, E.init, s) pre canRegister(i, s),

  canRegister: I.MemberId >< S.State -> Bool
  canRegister(i, s) is ~S.memberExists(i, s),

  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is S.deleteMember(i, s) pre canUnregister(i, s),

  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is S.memberExists(i, s)
```

Enviar un mensaje (send) significa insertarlo en la bandeja de entrada del remitente (`inbox`), siempre y cuando el mensaje esté bien formado. Recibir un mensaje (`receive`) significa removerlo de la bandeja de salida (`outbox`) del receptor, siempre y cuando la bandeja de salida no esté vacía.

```
value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let i = M.sender(m), e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end pre canSend(m, s),

  canSend: M.Message >< S.State -> Bool
  canSend(m, s) is S.iswfMessage(m, s),

  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let (m, e) = E.dequeueOutbox(S.getMember(i, s))
    in (m, S.modifyMember(i, e, s)) end pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is ~E.isEmptyOutbox(S.getMember(i, s))
```

Para poder transferir un mensaje desde el remitente al receptor, la función `transfer` remueve el mensaje de la bandeja de salida (no vacía) del remitente y lo envía a todos los receptores a través de `transferAll`. Inicializado con el conjunto de receptores, la función `transferAll` elige uno por uno a un miembro de este grupo, inserta el mensaje en la bandeja de salida de este miembro, remueve al miembro del conjunto y elige a otro miembro, etc., hasta que el conjunto quede vacío.

```
value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s)
    in transferAll(m, M.recepients(m), s') end pre canTransfer(i, s),
```

```
  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let
          e = E.enqueueOutbox(m, S.getMember(i, s)), s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is S.membersExist(ids, s) /\ ids <<= M.recepients(m)
```

4.6.2 MODELO 2 – Mensajería a través de Canales

La principal limitación del MODELO 1 es el requerimiento que todos los receptores de un mensaje sean explícitamente incluidos dentro del mensaje. Esto requiere un conocimiento detallado de los miembros que están inscriptos en el Gateway y quienes están involucrados en el intercambio de mensajes, a pesar de que la inscripción de miembros cambiará a través del tiempo y que el rol de los miembros puede cambiar de un intercambio de mensajes a otro. Modelos posteriores van a apuntar gradualmente a introducir más flexibilidad a la mensajería, alejándose de la mensajería directa del primer modelo y apuntando hacia la mensajería programable.

Con este fin, el MODELO 2 introduce canales para facilitar el intercambio de mensajes entre miembros. El modelo contiene todos los esquemas presentes en el primer modelo - `ID`, `MESSAGE`, `QUEUE`, `MEMBER`, `STATE` y `GATEWAY`. Sin embargo, sólo `ID`, `MESSAGE`, `STATE` y `GATEWAY` son modificados con respecto al MODELO 1. Además, el MODELO 2 contiene el esquema `CHANNEL` que define canales. Mostrados detalladamente en el Apéndice A.2, la relación entre los esquemas se muestra a continuación:

```
scheme ID = ...
scheme MESSAGE(I:ID) = ...
scheme QUEUE(I:ID, M:MESSAGE(I)) = ...
scheme MEMBER(I:ID, M:MESSAGE(I)) = extend QUEUE(I, M) with ...
scheme CHANNEL(I:ID) = ...
scheme STATE(I:ID, M:MESSAGE(I), E:MEMBER(I,M), C:CHANNEL(I)) = ...
scheme GATEWAY(I:ID, M:MESSAGE(I), E:MEMBER(I,M), C:CHANNEL(I), S:STATE(I,M,E,C)) = ...
```

Además de los identificadores de miembros `MemberId`, `ID` define a los identificadores de canales `ChannelId`. Como resultado, el esquema define la función `recipients` en `Message` para devolver un identificador de canal, no un conjunto de identificadores de miembros.

```
type                 type
  MemberId,            Message
  ChannelId          value
                       sender: Message -> I.MemberId,
                       recepients: Message -> I.ChannelId
```

Los esquemas `QUEUE` y `MEMBER` son iguales a los presentados en el MODELO 1. El nuevo esquema `CHANNEL` introduce un tipo `Channel` abstracto y dos funciones en este tipo: `owner` para devolver el identificador del miembro dueño del canal – `MemberId`; y `subscribers` para devolver el conjunto de identificadores de miembros de todos los suscriptores del canal – conjunto de `MemberId`. También hay un axioma presente para asegurar que ningún suscriptor de un canal sea también dueño de un canal. Asimismo, se definen tres funciones observadoras en `Channel` para revisar si un miembro es dueño o suscriptor, y si el canal no tiene suscriptores.

```
type
  Channel
value
  owner: Channel -> I.MemberId,
  subscribers: Channel -> I.MemberId-set
axiom
  (all c: Channel :-
    owner(c) ~isin subscribers(c)
  )
```

```
value
  isOwner: I.MemberId >< Channel -> Bool
  isOwner(i, c) is i = owner(c),

  isSubscriber: I.MemberId >< Channel -> Bool
  isSubscriber(i, c) is i isin subscribers(c),

  noSubscribers: Channel -> Bool
  noSubscribers(c) is subscribers(c) = {}
```

Un canal es inicializado a través de la función `init` para contener a un dueño (`MemberId`) y a un conjunto vacío de suscriptores. Los Miembros pueden suscribirse a un canal y desuscribirse de un a canal a través de las funciones `subscribe` y `unsubscribe`. No existe función para cambiar al dueño de un canal.

```
value
  init: I.MemberId -> Channel
  init(i) as c post
    owner(c) = i /\
    subscribers(c) = {}
```

```
value
  subscribe: I.MemberId >< Channel -~-> Channel
  subscribe(i, c) as c' post
    owner(c') = owner(c) /\
    subscribers(c') = subscribers(c) union {i}
    pre ~isSubscriber(i, c),

  unsubscribe: I.MemberId >< Channel -~-> Channel
  unsubscribe(i, c) as c' post
    owner(c') = owner(c) /\
    subscribers(c') = subscribers(c) \ {i}
    pre isSubscriber(i, c)
```

El estado es definido en el esquema STATE como un registro de `Members` – un mapeo determinístico de identificadores de miembros a miembros y `Channels` – un mapeo determinpistico de identificadores de canales a canales, restringidos por el predicado `iswf` definido posteriormente. El estado inicial `init` contiene mapeos vacíos de miembros y canales.

```
type
  Members = I.MemberId -m-> E.Member,
  Channels = I.ChannelId -m-> C.Channel,
  State'::
    members: Members <-> re_members
    channels: Channels <-> re_channels,
  State = {| s:State' :- iswf(s) |}
```

```
value
  init: Members = [],
  init: Channels = [],
  init: State' = mk_State'(init, init)
```

Varias funciones observadoras son definindas en el estado para verificar si: un miembro existe en el estado (`memberExists`), un conjunto de miembros existe en el estado (`membersExist`), un canal existe en el estado (`channelExists`) y si un conjunto de canales existe en el estado (`channelsExist`). Adicionalmente, `STATE` define las funciones para obtener un miembro (`getMember`) o un canal (`getChannel`) para un identificador dado, y para obtener un conjunto de canales que le pertenecen (`channelsOwned`) a un miembro dado y canales suscriptos (`channelsSubscribed`) por un miembro dado.

Se definen predicados para determinar si todos los elementos de un estado están bien formados. En particular: un mensaje está bien formado si el remitente está registrado, si el canal receptor existe y si el remitente es el dueño o un suscriptor de este canal (`iswfMessage`); una cola está bien formada si todos sus mensajes están bien formados (`iswfQueue`); un miembro está bien formado si sus bandejas de entrada y salida están bien formadas, es el remitente de cada mensaje en su bandeja de entrada y el dueño o suscriptor del canal receptor de cada mensaje en su bandeja de salida (`iswfMember`); un canal está bien formado si su dueño y todos los suscriptores están inscriptos (iswfChannel); un mapeo de miembros está bien formado si todos sus miembros están bien formados (`iswfMembers`); un mapeo de canales está bien formado si todos sus canales están bien formados (`iswfChannels`); y un estado está bien formado si sus mapeos de miembros y canales están bien formados (`iswf`). Aquí hay dos ejemplos de definiciones:

```
value
  iswfMessage: M.Message >< State' -> Bool
  iswfMessage(m, s) is
    let ms = M.sender(m), mr = getChannel(M.recepients(m), s) in
```

```
      memberExists(ms, s) /\
      channelExists(M.recepients(m), s) /\
      (C.isOwner(ms, mr) \/ C.isSubscriber(ms, mr))
    end,

  iswfMember: I.MemberId >< State' -~-> Bool
  iswfMember(i, s) is
    let e = getMember(i, s) in
      iswfQueue(E.inbox(e), s) /\ iswfQueue(E.outbox(e), s) /\
      (all m: M.Message :-
        let ms = M.sender(m), mr = getChannel(M.recepients(m), s) in
          (E.isInInbox(m, e) => ms = i) /\
          (E.isInOutbox(m, e) => (C.isOwner(i, mr) \/ C.isSubscriber(i, mr)))
        end
      )
    end pre memberExists(i, s),
```

Además de las funciones para agregar, eliminar y modificar miembros – `addMember`, `deleteMember`, `modifyMember`, STATE también provee las funciones correspondientes para canales – `addChannel`, `deleteChannel`, `modifyChannel`. Las seis funciones usan los re-constructores `re_members` and `re_channels` para modificar los mapeos. Aquí hay dos ejemplos:

```
value
  addMember: I.MemberId >< E.Member >< State -~-> State
  addMember(i, e, s) is
    re_members(members(s) union [ i +> e ], s)
    pre ~memberExists(i, s),

  addChannel: I.ChannelId >< C.Channel >< State -~-> State
  addChannel(i, c, s) is
    re_channels(channels(s) union [ i +> c ], s)
    pre ~channelExists(i, s)
```

Como en el MODELO 1, el esquema `GATEWAY` introduce las funciones para registrar y des-registrar miembros – `register` y `unregister`. Las funciones son idénticas a las anteriores, excepto que la precondición `canUnregister` requiere que el miembro a desregistrar exista y que no sea dueño de un canal o que no esté suscripto a ninguno:

```
value
  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is
    S.memberExists(i, s) /\ S.channelsOwned(i, s) = {} /\
    S.channelsSubscribed(i, s) = {}
```

Además, GATEWAY introduce las funciones para crear y destruir canales y para suscribirse y desuscribirse de ellos – `create`, `destroy`, `subscribe` y `unsubscribe`, todas dependientes de las funciones `addChannel`, `deleteChannel` y `modifyChannel` provistas por `STATE`, y funciones `subscribe` y `unsubscribe` provistas por `CHANNEL`. Todas las funciones salvo `destroy` requieren un miembro y un canal como parámetros; `destroy` solo requiere de un canal. Las precondiciones para estas funciones son las siguientes: creación de un canal – el dueño está inscripto y el canal no existe; destrucción de un canal – el canal existe, pero no tiene suscriptores; subscripción a un canal – el miembro y el canal existen, y el miembro no es ni dueño ni suscriptor de este canal; y des-suscripción de un canal – el miembro y el canal existen y el miembro está suscripto a este canal. Aquí hay dos ejemplos:

```
value
  subscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribe(im, ic, s) is
    let c = C.subscribe(im, S.getChannel(ic, s)) in  S.modifyChannel(ic, c, s) end
    pre canSubscribe(im, ic, s),

  canSubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canSubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    ~C.isSubscriber(im, S.getChannel(ic, s)) /\ ~C.isOwner(im, S.getChannel(ic, s)),

  unsubscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
```

```
  unsubscribe(im, ic, s) is
    let c = C.unsubscribe(im, S.getChannel(ic, s)) in S.modifyChannel(ic, c, s) end
    pre canUnsubscribe(im, ic, s),

  canUnsubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canUnsubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    C.isSubscriber(im, S.getChannel(ic, s))
```

Ambas funciones `send` y `receive`, al igual que `canTransfer` y `transferAll` son idénticas a las funciones correspondientes en el `MODELO 1`. En cuanto a las funciones `transfer` y `canTransferAll`, las referencias al conjunto de receptores de un mensaje `m - recipients(m)` son reemplazadas con los suscriptores del canal receptor - `C.subscribers(S.getChannel(M.recepients(m), s))`, como se muestra debajo.

```
value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      ids = C.subscribers(S.getChannel(M.recepients(m), s))
    in transferAll(m, ids, s') end
    pre canTransfer(i, s)

value
  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is
    S.membersExist(ids, s) /\
    ids <<= C.subscribers(S.getChannel(M.recepients(m), s))
```

4.6.3 MODELO 3 – Introduciendo Roles de Miembros y Canales

Mientras que el MODELO 2 permite a miembros enviar y recibir mensajes a través de canales creados y suscriptos por usuarios, deja abierto cómo las operaciones para registración y desregistración de miembros, creación y destrucción de canales, y susbcripción y des-suscripción de miembros a/de canales son llevadas a cabo. Nuestro objetivo es que tal operación se lleve a cabo a través de mensajería que involucre a dos miembros especiales del Gateway – `admin` and `visitor`. El rol del miembro `visitor` es el de facilitar la registración de nuevos miembros. El rol de `admin` es de administrar los cambios en el Gatewayl en términos de la existencia de miembros y canales. A fin de que `admin` y `visitor` puedan comunicarse entre ellos y con otros miembros, incluso en ausencia de cualquier canal definido por miembros, se introducen canales especiales para establecer conexiones uno-a-uno entre cualquier miembro (incluyendo `visitor`) y `admin`.

El MODELO 3 apunta a introducir los miembros `admin` y `visitor`, además del conjunto de canales de administración, con principales operaciones del Gatewayl redefinidas para poder tomar en cuenta a tales miembros y canales. El modelo comprende el mismo conjunto de esquemas que el MODELO 2 - `ID`, `MESSAGE`, `QUEUE`, `MEMBER`, `CHANNEL`, `STATE` y `GATEWAY` con las mismas relaciones. Entre ellos, solo `ID`, `STATE` y `GATEWAY` son modificados en el MODELO 3, mostrados en detalle en el Apéndice A.3.

Además del tipo `MemberId`, ID introduce dos valores distinguidos para este tipo – `admin` y `visitor`, de forma tal que los dos valores son diferentes. Asimismo, se introduce `ChannelID` junto con la función `adminChannel` que dado un identificador de miembro devuelve el identificador del canal de administración de ese miembro; un axioma está presente para asegurar que está asignación es única. Un observador `isAdminChannel` verifica si un canal dado es un canal de administración para algún miembro.

```
type
  MemberId
value
  admin, visitor: MemberId
axiom
  admin ~= visitor
```

```
type
  ChannelId
value
  adminChannel: MemberId -> ChannelId
axiom
  (all im1, im2: MemberId :-
    im1 ~= im2 => adminChannel(im1) ~= adminChannel(im2)
  )
value
  isAdminChannel: ChannelId -> Bool
  isAdminChannel(ic) is
    (exists im: MemberId :- adminChannel(im) = ic)
```

Como en el MODELO 2, el estado es definido como un registro de mapeos de miembros y canales, restringiodos por el predicado `iswf`. Inicialmente, el estado contiene solo miembros `admin` y `visitor` y un canal `adminChannel` que los une. Creado a través de `initAdmin`, el canal pertenece a `admin` y es suscripto por `visitor`.

```
value
  initAdmin: I.MemberId -> C.Channel
  initAdmin(i) as c post C.owner(c) = I.admin /\ C.subscribers(c) = {i}

value
  init: Members = [I.admin +> E.init, I.visitor +> E.init],
  init: Channels = [ I.adminChannel(I.visitor) +> initAdmin(I.visitor) ],
  init: State' = mk_State'(init, init)
```

Todos los observadores de estado – `memberExists`, `membersExist`, `channelExists`, `channelsExist`, `getMember`, `getChannel`, `channelsOwned` y `channelsSubscribed`; son iguales en el MODELO 2 y en el MODELO 3. También son iguales un número de predicados para revisar si varios elementos del estado están bien formados – `iswfMessage`, `iswfQueue`, `iswfMember`, `iswfChannel`, `iswfMembers` y `iswfChannels`, Sin embargo, `iswf` en si mismo, aplica dos nuevos predicados para controlar si los miembros `admin` y `visitor` están bien formados – `iswfAdmin` y `iswfVisitor`. El primero asegura que el miembro `admin` esté siempre presente en el estado y que `adminChannel` exista para todos los miembros inscriptos, que sean propiedad de `admin` y estén susbcriptos solamente por el miembro. El segundo asegura que el miembro `visitor` esté siempre presente en el estado, sin ser dueño de ningún canal y que esté solamente inscripto al canal `adminChannel(visitor)`. Finalmente, `iswf` requiere que, en un estado bien formado todos los miembros y canales, además de los miembros `admin` y `visitor` estén bien formados.

```
value
  iswfAdmin: State' -> Bool
  iswfAdmin(s) is
    memberExists(I.admin, s) /\
    (all mi: I.MemberId :-
      memberExists(mi, s) =>
      channelExists(I.adminChannel(mi), s) /\
      C.owner(getChannel(I.adminChannel(mi), s)) = I.admin /\
      C.subscribers(getChannel(I.adminChannel(mi), s)) = {mi}
    ),

  iswfVisitor: State' -> Bool
  iswfVisitor(s) is
    memberExists(I.visitor, s) /\
    (all ci: I.ChannelId :-
      channelExists(ci, s) =>
      I.visitor ~= C.owner(getChannel(ci, s)) /\
      (I.visitor isin C.subscribers(getChannel(ci, s))
      => ci = I.adminChannel(I.visitor))
    ),

  iswf: State' -> Bool
  iswf(s) is
    iswfMembers(members(s), s) /\ iswfChannels(channels(s), s) /\
    iswfAdmin(s) /\ iswfVisitor(s)
```

Las funciones restantes para agregar, eliminar y modificar miembros y canales son todas iguales. En GATEWAY, se producen muchas diferencias en la manera que se se llevan a cabo varias operaciones del Gateway y bajo qué condiciones. En particular, `register` no sólo agrega un miembro al estado, sino que también agrega el canal de administración para este miembro asumiendo, como antes, que el miembro no está todavía presente en el estado. Asimismo, `unregister` no solo elimina a un miembro del estado, sino también elimina al canal de administración de este miembro, asumiendo que el miembro existe en el estado, no es ni `admin` ni `visitor`, no es dueño de ningún canal, y está suscripto solamente a su correspondiente canal de administración.

```
value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is
    let
      s1 = S.addMember(i, E.init, s),
      s2 = S.addChannel(I.adminChannel(i), S.initAdmin(i), s1)
    in s2 end pre canRegister(i, s),

  canRegister: I.MemberId >< S.State -> Bool
  canRegister(i, s) is ~S.memberExists(i, s)

value
  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is
    let
      s1 = S.deleteMember(i, s),
      s2 = S.deleteChannel(I.adminChannel(i), s1)
    in s2 end pre canUnregister(i, s),

  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is
    S.memberExists(i, s) /\ i ~= I.admin /\ i ~= I.visitor /\
    S.channelsOwned(i, s) = {} /\
    S.channelsSubscribed(i, s) = {I.adminChannel(i)}
```

Las funciones para crear y destruir canales, al igual que para suscribir y desuscribir miembros a/de canales, son las mismas que en el MODELO 2, pero son ejecutadas bajo diferentes precondiciones: `canCreate` requiere que el canal no exista y que el miembro si exista, pero que no sea el miembro `visitor`; `canDestroy` requiere que el canal exista, que no sea un canal de administración y que no tenga suscriptores; `canSubscribe` requiere que el miembro y el canal existan, que el miembro no este suscripto ni sea dueño de este canal, y que no sea un miembro `visitor`; y `canUnsubscribe` requiere que el miembro y el canal existan, que el miembro esté suscripto a este canal y que el canal no sea un canal de administración.

```
value
  canCreate: I.ChannelId >< I.MemberId >< S.State -> Bool
  canCreate(ic, im, s) is
    ~S.channelExists(ic, s) /\ S.memberExists(im, s) /\ im ~= I.visitor,

  canDestroy: I.ChannelId >< S.State -> Bool
  canDestroy(ic, s) is
    S.channelExists(ic, s) /\ ~I.isAdminChannel(ic) /\
    C.noSubscribers(S.getChannel(ic, s)),

  canSubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canSubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    ~C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~C.isOwner(im, S.getChannel(ic, s)) /\ im ~= I.visitor

  canUnsubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canUnsubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~I.isAdminChannel(ic)
```

Las funciones restantes para envío, recepción y transferencia de mensajes entre miembros son iguales.

4.6.4 MODELO 4 –Distribución del Estado entre Miembros

Todos los modelos anteriores asumieron un estado global del Gateway, equipados con funciones centralizadas para modificar este estado. Mientras que esta suposición simplifica les especificaciones, debe ser revisada cuando el modelo es refinado hacia la implementación actual. El objetivo del Modelo 4 es el de distribuir el estado global del Gateway entre los estados locales de sus miembros registrados. En particular, tales miembros van a contener información acerca de los canales que poseen y que suscriben. Para facilitar los cálculos, mientras se paga el precio por la complejidad agregada, la información de canal es replicada entre sus dueños que registran a todos los suscriptores de los canales y a los suscriptores que registran al dueño del canal.

El modelo comprende el mismo conjunto de módulos que el MODELO 3 – `ID`, `MESSAGE`, `QUEUE`, `CHANNEL`, `MEMBER`, `STATE` y `GATEWAY`, con `ID`, `MESSAGE`, `QUEUE` y `CHANNEL` sin presentar cambios con respecto a la versión anterior. Sin embargo, hay un cambio en la relación entre los esquemas, donde `CHANNEL` es uno de los parámetros de `MEMBER`:

```
scheme ID = ...
scheme MESSAGE(I:ID) = ...
scheme QUEUE(I:ID, M:MESSAGE(I)) = ...
scheme CHANNEL(I:ID) = ...
scheme MEMBER(I:ID, M:MESSAGE(I), C:CHANNEL(I)) = extend QUEUE(I, M) with ...
scheme STATE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C)) = ...
scheme GATEWAY(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) = ...
```

Además de las bandejas de entrada y salida, el tipo `Member` es definido en `MEMBER` como un registro bien formado que comprende cinco campos - identificador (`id`), bandeja de entrada (`inbox`), bandeja de salida (`outbox`), canales propios (`owned`) y canales suscriptos (`subscribed`). El campo `owned` es un mapeo de identificadores de canales (`ChannelId`) a canales `Channel`) – todo el registro de cada canal que pertenece a un miembro. El campo `subscribed` es definido como un mapeo de identificadores de canales (`ChannelId`) a identificadores de miembros `MemberId`) – manteniendo la identidad de los dueños de canales por cada canal suscripto. Como todos los campos, excepto `id`, pueden cambiar, son definidos junto con los reconstructores correspondientes de la siguiente manera:

```
type
  Member::
    id: I.MemberId
    inbox: Queue <-> re_inbox
    outbox: Queue <-> re_outbox
    owned: I.ChannelId -m-> C.Channel <-> re_owned
    subscribed: I.ChannelId -m-> I.MemberId <-> re_subscribed,
  Member = {| m: Member' :- iswfMember(m) |}
```

Los observadores del MODELO 3 – `isEmptyInbox`, `isEmptyOutbox`, `isInInbox` y `isInOutbox` se mantienen, y también se agregan varios observadores nuevos: `isChannelOwned` – verifica si un canal pertenece a algún miembro, `isChannelSubscribed` – verifica si un canal es suscripto por un miembro, `channelsOwned` – devuelve el conjunto de canales que pertenecen a un miembro, `channelsSubscribed` – devuelve el conjunto de canales subcriptos por un miembro, y `subscribers` – devuelve el conjunto de suscriptores de un canal. Los nuevos observadores son definidos de la siguiente manera:

```
value
  isChannelOwned: I.ChannelId >< Member' -> Bool
  isChannelOwned(ic, e) is ic isin dom owned(e),

  isChannelSubscribed: I.ChannelId >< Member' -> Bool
  isChannelSubscribed(ic, e) is ic isin dom subscribed(e),

  channelsOwned: Member' -> I.ChannelId-set
  channelsOwned(e) is dom owned(e),

  channelsSubscribed: Member' -> I.ChannelId-set
  channelsSubscribed(e) is dom subscribed(e),

  subscribers: I.ChannelId >< Member' -~-> I.MemberId-set
  subscribers(ic, e) is C.subscribers(owned(e)(ic)) pre isChannelOwned(ic, e)
```

Para poder establecer si el registro de un miembro dado está bien formado (`iswfMember`), sus mensajes, su bandeja de entrada, su bandeja de salida y el registro de los canales propios y suscriptos deben estar todos bien formados: por cada mensaje en las bandejas de entrada y salida, el miembro debe ser dueño o suscriptor del canal receptor de este mensaje (`iswfMessage`); cada mensaje en la bandeja de entrada de un miembro debe estar bien formadoy el remitente debe ser dicho miembro (`iswfOutbox`); y ningún canal puede simultáneamente ser propiedad y estar suscripto por un mismo miembro, y un miembro siempre está suscripto a su canal de administración, y cada canal que pertenece a un miembro registra a este miembro como su dueño (`iswfChannels`).

```
value
  iswfMessage: M.Message >< Member' -> Bool
  iswfMessage(m, e) is
    let mr = M.recepients(m) in
      (isInInbox(m, e) \/ isInOutbox(m, e)) /\
      (isChannelOwned(mr, e) \/ isChannelSubscribed(mr, e))
    end,

  iswfInbox: Queue >< Member' -> Bool
  iswfInbox(q, e) is
    isEmpty(q) \/
    M.sender(hd q) = id(e) /\ iswfMessage(hd q, e) /\ iswfInbox(tl q, e),

  iswfOutbox: Queue >< Member' -> Bool
  iswfOutbox(q, e) is
    isEmpty(q) \/ iswfMessage(hd q, e) /\ iswfOutbox(tl q, e),

  iswfChannels: Member' -> Bool
  iswfChannels(e) is
    dom owned(e) inter dom subscribed(e) = {} /\
    isChannelSubscribed(I.adminChannel(id(e)), e) /\
    (all ic: I.ChannelId :- isChannelOwned(ic, e) => C.owner(owned(e)(ic)) = id(e)),

  iswfMember: Member' -> Bool
  iswfMember(e) is iswfInbox(inbox(e), e) /\ iswfOutbox(outbox(e), e) /\ iswfChannels(e)
```

Inicialmente, un miembro bien formado no contiene mensajes ni en su bandeja de entrada ni de salida, no es dueño de ningún canal, y solo se suscribe al canal de administración correspondiente (`init`):

```
value
  init: I.MemberId -> Member
  init(im) as e post
    id(e) = im /\ inbox(e) = init /\ outbox(e) = init /\
    owned(e) = [] /\ subscribed(e) = [ I.adminChannel(im) +> I.admin ]
```

Las funciones del MODELO 3 agregar y remover mensajes de las colas de las bandejas de entrada y salida son redefinidas para tomar en cuenta la nueva estructura de los miembros. Por ejemplo, para insertar un mensaje en la cola de la bandeja de entrada del miembro, el mensaje de estar bien formado; para remover el mensaje de la bandeja de salida del miembro, la cola de la bandeja de salida no debe estar vacía.

```
value
  enqueueInbox: M.Message >< Member -~-> Member
  enqueueInbox(m, e) is
    re_inbox(enqueue(m, inbox(e)), e) pre canEnqueueInbox(m, e),

  canEnqueueInbox: M.Message >< Member -> Bool
  canEnqueueInbox(m, e) is iswfMessage(m, e),

  dequeueOutbox: Member -~-> (M.Message >< Member)
  dequeueOutbox(e) is
    let (m', q) = dequeue(outbox(e)) in (m', re_outbox(q, e)) end
    pre ~canDequeueOutbox(e),

  canDequeueOutbox: Member -> Bool
  canDequeueOutbox(e) is ~isEmptyOutbox(e)
```

Tres pares de funciones son también provistas por MEMBER con el objetivo de: agregar y eliminar canales que pertenecen a los miembros (addChannelOwned, deleteChannelOwned), para agregar y eliminar suscriptores a los canales que pertenecen al miembro (addSubscriber, deleteSubscriber), para agregar y eliminar al miembro mismo como miembro suscripto a otros canales (addChannelSubscribed, deleteChannelSubscribed). Aquí están las funciones para agregar canales, suscriptores y subscripciones. La función addSubscriber ejecutada por el dueño del canal debe ser complementada por la función addChannelSubscribed ejecutada por el suscriptor, y viceversa. Como no se tiene acceso al estado de otros miembros a este nivel, esta invocación complementaria esta hecha en el módulo STATE.

```
value
  addChannelOwned: I.ChannelId >< Member -~-> Member
  addChannelOwned(ic, e) is
    re_owned(owned(e) !! [ ic +> C.init(id(e)) ], e)
    pre canAddChannelOwned(ic, e),

  canAddChannelOwned: I.ChannelId >< Member -> Bool
  canAddChannelOwned(ic, e) is ~isChannelOwned(ic, e) /\ ~isChannelSubscribed(ic, e)

value
  addSubscriber: I.ChannelId >< I.MemberId >< Member -~-> Member
  addSubscriber(ic, im, e) is
    re_owned(owned(e) !! [ic +> C.subscribe(im, owned(e)(ic))], e)
    pre canAddSubscriber(ic, im, e),

  canAddSubscriber: I.ChannelId >< I.MemberId >< Member -> Bool
  canAddSubscriber(ic, im, e) is
    isChannelOwned(ic, e) /\ ~C.isSubscriber(im, owned(e)(ic))

value
  addChannelSubscribed: I.ChannelId >< I.MemberId >< Member -~-> Member
  addChannelSubscribed(ic, io, e) is
    re_subscribed(subscribed(e) !! [ ic +> io ], e)
    pre canAddChannelSubscribed(ic, e),

  canAddChannelSubscribed: I.ChannelId >< Member -> Bool
  canAddChannelSubscribed(ic, e) is
    ~isChannelOwned(ic, e) /\ ~isChannelSubscribed(ic, e)
```

Las funciones tienen definidas varias precondiciones: a fin de agregar un canal propio, el canal no debe pertenecer or estar suscrito por el miembro (canAddChannelOwned); para eliminar un canal propio, el canal debe pertenecer al miembro (canDeleteChannelOwned); para agregar un suscriptor a un canal propio, el miembro no debe ser un suscriptor (canAddSubscriber); para eliminar a un suscriptor de un canal propio, el miembro debe estar suscripto (canDeleteSubscriber); para agregar una nueva suscripcion de un miembro a un canal, el canal no debe pertenecer ni estar suscripto por el miembro (canaddChannelSubscribed); y para eliminar la suscripción de un miembro a un canal, el miembro debe estar subscrito al canal (canDeleteChannelSubscribed).

El estado en el MODELO 4 se define simplemente como un mapeo de identificadores de miembros a miembros, donde el predicado iswf define qué significa que un estado esté bien formado. Un estado inicial init contiene dos entradas en este mapeo, una para el miembro admin y otra para el miembro visitor. Como en un modelo anterior, dos observadores en el estado determinan si un miembro identificado existe (memberExists), y si es así, devuelve este miembro (getMember).

```
type
  State' = I.MemberId -m-> E.Member,
  State = {| s:State' :- iswf(s) |}

value
  init: State' = [
    I.admin +> E.init(I.admin),
    I.visitor +> E.init(I.visitor)]
```

```
value
  memberExists: I.MemberId >< State' -> Bool
  memberExists(i, s) is i isin dom s,

  getMember: I.MemberId >< State' -~-> E.Member
  getMember(im, s) is s(im)
    pre memberExists(im, s)
```

Tres observadores más son definidos en State: para verificar si un canal existe en el estado (channelExists), devolver un canal existente (getChannel) y devolver al dueño de un canal existente (getOwner).

```
value
  channelExists: I.ChannelId >< State' -> Bool
  channelExists(ic, s) is
    (exists im: I.MemberId :-
      memberExists(im, s) /\ E.isChannelOwned(ic, getMember(im, s))
    ),

  getChannel: I.ChannelId >< State' -~-> C.Channel
  getChannel(ic, s) is E.owned(getMember(getOwner(ic, s), s))(ic) pre channelExists(ic, s)

  getOwner: I.ChannelId >< State' -~-> I.MemberId
  getOwner(ic, s) as im
    post E.isChannelOwned(ic, getMember(im, s)) pre channelExists(ic, s),
```

Como antes, para que un estado esté bien formado (`iswf`), sus miembros (`iswfMembers`), canales (`iswfChannels`), al igual que los miembros `admin` (`iswfAdmin`) y `visitor` (`iswfVisitor`), deben estar bien formados también. No es necesario revisar los mensajes y las colas de los mensajes, ya que esto se hace en `MEMBER`, y muchas funciones son idénticas a las del MODELO 3. La mayor diferencia es cómo los identificadores y los canales son registrados dentro de las estructuras de los miembros. Por ejemplo, `iswfMember` verifica si el miembro identificado está bien formado, si almacena correctamente al identificador del miembro, si todos los canales propios están bien formados, y si todos los canales suscriptos existen y si su propiedad está correctamente registrada. Ya no garantizada en el estado distribuido entre los miembros, también se requiere que todos los canales tengan solamente un dueño (`iswfChannels`).

```
value
  iswfMember: I.MemberId >< State' -> Bool
  iswfMember(im, s) is
    let e = getMember(im, s) in
      E.id(e) = im /\ E.iswfMember(e) /\
      (all ic: I.ChannelId :-
        (E.isChannelOwned(ic, e) => iswfChannel(getChannel(ic, s), s)) /\
        (E.isChannelSubscribed(ic, e) =>
          channelExists(ic, s) /\ getOwner(ic, s) = E.subscribed(e)(ic))
      )
    end,

  iswfChannels: State' -> Bool
  iswfChannels(s) is
    (all im1, im2: I.MemberId :- im1 ~= im2 =>
      E.channelsOwned(getMember(im1, s)) inter E.channelsOwned(getMember(im2, s)) = {}
    )
```

Las funciones para agregar (`addMember`), eliminar (`deleteMember`) y modificar (`modifyMember`) miembros están todavía incluídas en `STATE`, pero no aquellas funciones para los canales que ahora pertenecen a `MEMBER`. Dos nuevas funciones en `STATE` son `addAdminChannel` y `deleteAdminChannel`, para agregar y eliminar canales de administración, modificando el miembro `admin`. La función `addAdminChannel` agrega el canal de administración de un miembro dado a los canales que pertenecen a `admin`, y luego introduce al miembro como suscriptor al canal.

```
value
  addAdminChannel: I.MemberId >< State -~-> State
  addAdminChannel(im, s) is
    let
      e1 = getMember(I.admin, s),
      e2 = E.addChannelOwned(I.adminChannel(im), e1),
      e3 = E.addSubscriber(I.adminChannel(im), im, e2)
    in s !! [ I.admin +> e3 ] end
    pre ~channelExists(I.adminChannel(im), s),

  deleteAdminChannel: I.MemberId >< State -~-> State
  deleteAdminChannel(im, s) is
    let
      e1 = getMember(I.admin, s),
      e2 = E.deleteChannelOwned(I.adminChannel(im), e1),
      e3 = E.deleteSubscriber(I.adminChannel(im), im, e2)
    in s !! [ I.admin +> e3 ] end
    pre channelExists(I.adminChannel(im), s)
```

GATEWAY define a las funciones `register` and `unregister` agregando y eliminando miembros usando las funciones `addMember` y `deleteMember` definidas en STATE, y agregando y eliminando canales de administración a través de las funciones `addAdminChannel` y `deleteAdminChannel` definidas arriba. Las precondiciones son esencialmente las mismas.

```
value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is
    let s1 = S.addMember(i, E.init(i), s), s2 = S.addAdminChannel(i, s1) in s2 end
    pre canRegister(i, s),

  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is
    let s1 = S.deleteMember(i, s), s2 = S.deleteAdminChannel(i, s1) in s2 end
    pre canUnregister(i, s)
```

La creación y destrucción de canales – `create` y `destroy`, se hace a través de la función `modifyMember` definida en STATE, y a través de las funciones `addChannelOwned` y `deleteChannelOwned` definidas directamente en MEMBER. La subscripción de miembros a canales – `subscribe`, requiere dos invocaciones de `modifyMember` – una para el dueño del canal para agregar al miembro como otro suscriptor (`addSubscriber`) y otro para que el miembro suscriptor registre la nueva suscripción del canal (`addChannelSubscribed`). Asimismo, `unsubscribe` también requiere dos invocaciones de `modifyMember` – uno para que el dueño elimine al miembro como suscriptor (`deleteSubscriber`) y otro para que el suscriptor elimine la suscripción del canal (`deleteChannelSubscribed`). Las precondiciones no se modifican.

```
value
  subscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribe(im, ic, s) is
    let
      io = S.getOwner(ic, s),
      eo = E.addSubscriber(ic, im, S.getMember(io, s)),
      es = E.addChannelSubscribed(ic, io, S.getMember(im, s))
    in S.modifyMember(io, eo, S.modifyMember(im, es, s)) end
    pre canSubscribe(im, ic, s),

  unsubscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  unsubscribe(im, ic, s) is
    let
      io = S.getOwner(ic, s),
      eo = E.deleteSubscriber(ic, im, S.getMember(io, s)),
      es = E.deleteChannelSubscribed(ic, S.getMember(im, s))
    in S.modifyMember(io, eo, S.modifyMember(im, es, s)) end
    pre canUnsubscribe(im, ic, s)
```

Finalmente, las funciones provistas por GATEWAY para enviar (`send`), recibir (`receive`) y transferir (`transfer`) mensajes son idénticas a las funciones del MODELO 3, exceptuando `send`, cuya precondición verifica si el mensaje está bien formado basándose en el predicado `iswfMessage` definido en el esquema `MEMBER` y no el esquema `STATE`.

4.6.5 MODEL 5 –Tipos de Mensaje y Todo-Incluido por Mensajería

El MODELO 4 transformó al Gateway de una estructura centralizada a una distribuida, en donde todos los miembros mantienen parte del estado localmente. Sin embargo, las operaciones para registrar y desregistrar a miembros, para crear y destruir canales, y para suscribir y desuscribir miembros a/de canales se seguían llevando a cabo de una manera centralizada, permitiendo acceder y modificar partes locales y remotas del estado dentro de una sola operación.

El objetivo del MODELO 5 es permtir la especificación de todas las operaciones del Gateway a través de la mensajería, con miembros designados requiriendo y procesando la ejecución de registrar, desregistrar, crear, destruir, suscribir, desuscribir y otras operaciones mediante el envío, recepción y procesamiento de cierto tipo de mensajes. El modelo comprende los siete esquemas introducidos en el MODELO 4 – `ID`, `MESSAGE`, `QUEUE`, `CHANNEL`, `MEMBER`, `STATE` y

GATEWAY, además de dos esquemas nuevos – OPERATIONS que capturan las signaturas de las funciones para enviar y recibir para todas las operaciones del Gateway, e INTERFACE – contiene especificaciones completas de tales operaciones a invocarse directamente por los usuarios del Gateway, dejando a GATEWAY para que defina solamente las funciones internas de send, receive y transfer. Se requiere que en el modelo el esquema OPERATIONS resuelva la mutua dependencia entre los esquemas GATEWAY e INTERFACE: mientras que la función send en GATEWAY es invocada por varias funciones send en INTERFACE (e.g. sendRegister), la función receive invoca a varias funciones receive en INTERFACE (e.g. receiveRegister). Las dependencias entre varios esquemas presentes en el MODELO 5 son las mismas que las presentes en el MODELO 4, como se muestra abajo:

```
scheme ID = ...
scheme MESSAGE(I:ID) = ...
scheme QUEUE(I:ID, M:MESSAGE(I)) = ...
scheme CHANNEL(I:ID) = ...
scheme MEMBER(I:ID, M:MESSAGE(I), C:CHANNEL(I)) = extend QUEUE(I, M) with ...
scheme STATE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C)) = ...
scheme OPERATIONS(I:ID, M:MESSAGE(I),C:CHANNEL(I),E:MEMBER(I,M,C), S:STATE(I,M,C,E)) = ...
scheme GATEWAY(
         I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C),
         S:STATE(I,M,C,E), O:OPERATIONS(I,M,C,E,S))
       ) = ...
scheme INTERFACE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) =
  class
    object
      O: class ... end,
      G: GATEWAY(I,M,C,E,S,O)
    ...
  end
```

En especial, el objeto O en el esquema INTERFACE implementa las signaturas en el esquema OPERATIONS, reemplazando tales signaturas con un conjunto concreto de funciones de enviar y recibir definidas en INTERFACE. Entre los esquemas que aparecen arriba, ID, QUEUE, CHANNEL, MEMBER y STATE son iguales que en el MODELO 4, y solo MESSAGE, GATEWAY, OPERATIONS e INTERFACE requieren definiciones nuevas o revisadas.

Considere el esquema MESSAGE que define al tipo Message como un registro que contiene cuatro campos: sender – identificador del miembro, recipients – identificador del canal, type – el tipo del mensaje y error – un flag Booleano para señalar que ocurrió un error durante el procesamiento del mensaje. MessageType es definido como un tipo variante con seis valores que solicitan operaciones de register, unregister, create, destroy, subscribe y unsubscribe, cada una conteniendo un número de parámetros requeridos para la operación solicitada, y un número desconocido de valores adicionales. Por ejemplo, register contiene al identificador del miembro a ser registrado (MemberId), mientras que subscribe contiene al miembro a ser suscripto (MemberId) y el canal al cual el miembro requiere suscribirse. (ChannelId).

```
type
  MessageType ==
    register(I.MemberId) |
    unregister(I.MemberId) |
    create(I.ChannelId, I.MemberId) |
    destroy(I.ChannelId, I.MemberId) |
    subscribe(I.MemberId, I.ChannelId) |
    unsubscribe(I.MemberId, I.ChannelId) |
    _
```

```
type
  Message::
    mtype: MessageType
    sender: I.MemberId
    recepients: I.ChannelId
    error: Bool <-> re_error
```

El esquema OPERATIONS introduce la signatura de todas las operaciones usadas para procesar varios tipos de mensajes, junto con sus precondiciones. Cada operación toma un mensaje (Message) y un estado (State) y devuelve un nuevo estado; el nuevo tipo de Message contiene toda la información requerida para procesarlo. Asimismo, cada precondición toma un mensaje y un estado y devuelve un valor Booleano.

```
value
  register: M.Message >< S.State -~-> S.State,
  unregister: M.Message >< S.State -~-> S.State,
  create: M.Message >< S.State -~-> S.State,
  destroy: M.Message >< S.State -~-> S.State,
  subscribe: M.Message >< S.State -~-> S.State,
  unsubscribe: M.Message >< S.State -~-> S.State

value
  canRegister: M.Message >< S.State -> Bool,
  canUnregister: M.Message >< S.State -> Bool,
  canCreate: M.Message >< S.State -> Bool,
  canDestroy: M.Message >< S.State -> Bool,
  canSubscribe: M.Message >< S.State -> Bool,
  canUnsubscribe: M.Message >< S.State -> Bool
```

El esquema GATEWAY ya no contiene las especificaciones de tales funciones, que han sido movidos al esquema INTERFACE. En cambio, se refiere a tales funciones a través de sus signaturas presentes en el esquema OPERATIONS, usado como parámetro. Con las funciones send y transfer sin cambios, la función receive ha sido reestructurada para obtener el mensaje de la cola de la bandeja de salida del miembro receptor, siempre y cuando tal cola no esté vacía, quitar el mensaje y proceder posteriormente según el tipo de mensaje. En cada caso, primero determina si se cumple la precondición de la operación correspondiente. Si es así, invoca a la operación y devuelve el mensaje y el estado resultante. Si no, cambia el campor error en el mensaje y devuelve el mensaje modificado y el estado original.

```
value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let
      (m, e) = E.dequeueOutbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s)
    in
      case M.mtype(m) of
        M.register(i) ->
          if O.canRegister(m, s')
          then (m, O.register(m, s')) else (M.re_error(true, m), s') end,
        M.unregister(i) ->
          if O.canUnregister(m, s')
          then (m, O.unregister(m, s')) else (M.re_error(true, m), s') end,
        M.create(ic, im) ->
          if O.canCreate(m, s')
          then (m, O.create(m, s')) else (M.re_error(true, m), s') end,
        M.destroy(ic, im) ->
          if O.canDestroy(m, s')
          then (m, O.destroy(m, s')) else (M.re_error(true, m), s') end,
        M.subscribe(im, ic) ->
          if O.canSubscribe(m, s')
          then (m, O.subscribe(m, s')) else (M.re_error(true, m), s') end,
        M.unsubscribe(im, ic) ->
          if O.canUnsubscribe(m, s')
          then (m, O.unsubscribe(m, s')) else (M.re_error(true, m), s') end
      end
    end pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is ~E.isEmptyOutbox(S.getMember(i, s))
```

El esquema INTERFACE introduce un par de funciones de enviar y recibir para cada operación de register, unregister, create, destroy, subscribe y unsubscribe. Por ejemplo, la operación register es especificada a través de las funciones registerSend y registerReceive. Cada función de enviar invoca la función send definida en GATEWAY, mientras que cada función de recibir es invocada por la función receive definida en GATEWAY. Esta invocación mutua es posible declararando primero al objeto O, una instancia de OPERATIONS, para contener las signaturas de todas las funciones de recibir introducidas en el esquema INTERFACE, por ejemplo, register es instanciada para registerReceive, y pasando este objeto como el parámetro OPERATIONS al objeto GATEWAY declarado.

Por ejemplo, `registerSend` formula primero un mensaje con el tipo, remitente, receptores y flag de error correctos, y luego envía este mensaje a través de la función `send` provista por `GATEWAY`. El mensaje tiene el tipo `register`, `visitor` como remitente, el canal de administración de `visitor` como el receptor, y el flag de error indicando `false`. La precondición `canRegisterSend` requiere que el miembro a registrar no exista.

```
value
  registerSend: I.MemberId >< S.State -~-> S.State
  registerSend(i, s) is
    let
      mtype = M.register(i),
      sender = I.visitor, recepients = I.adminChannel(I.visitor)
    in G.send(M.mk_Message(mtype, sender, recepients, false), s) end
    pre canRegisterSend(i, s),

  canRegisterSend: I.MemberId >< S.State -> Bool
  canRegisterSend(i, s) is ~S.memberExists(i, s),
```

Según el canal receptor del mensaje `register`, el mensaje es recibido por `admin` que invoca a la función `registerReceive`. Por consiguiente, `admin` agrega al miembro registrado junto con su canal de administración al estado, devolviendo el nuevo estado, al igual que en el MODELO 4. La precondición `canRegisterReceive` requiere que el mensaje sea del tipo `register` y que el miembro solicitado no esté registrado.

```
value
  registerReceive: M.Message >< S.State -~-> S.State
  registerReceive(m, s) is
    case M.mtype(m) of
      M.register(i) ->
        let
          s1 = S.addMember(i, E.init(i), s),
          s2 = S.addAdminChannel(i, s1)
        in s2 end
    end
    pre canRegisterReceive(m, s),

  canRegisterReceive: M.Message >< S.State -> Bool
  canRegisterReceive(m, s) is
    case M.mtype(m) of
      M.register(i) -> ~S.memberExists(i, s),
      _ -> false
    end
```

Las funciones restantes son definidas de la misma manera. Por ejemplo, aquí está el par de funciones de subscripción:

```
value
  subscribeSend: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribeSend(im, ic, s) is
    let mtype = M.subscribe(im, ic), sender = im, recepients = I.adminChannel(im)
    in G.send(M.mk_Message(mtype, sender, recepients, false), s) end
    pre canSubscribeSend(im, ic, s),

  subscribeReceive: M.Message >< S.State -~-> S.State
  subscribeReceive(m, s) is
    case M.mtype(m) of
      M.subscribe(im, ic) ->
      let
        io = S.getOwner(ic, s),
        eo = E.addSubscriber(ic, im, S.getMember(io, s)),
        es = E.addChannelSubscribed(ic, io, S.getMember(im, s))
      in S.modifyMember(io, eo, S.modifyMember(im, es, s)) end
    end pre canSubscribeReceive(m, s),
```

4.6.6 MODELO 6 – Mensajería a través de los Dueños de Canal

El MODELO 5 tiene una gran limitación – la funcíón `transfer` mueve mensajes directamente entre la cola de la bandeja de entrada del remitente y las colas de las bandejas de salida de los receptores (todos los suscriptores del canal). Esto reduce la funcionalidad del portal a un simple transporte de mensajes entre miembros. A fin de extender esta funcionalidad, debería ser posible que los mensajes sean procesados cuando son transportados por el Gateway entre miembros. El objetivo del MODELO 6 es permitir esto.

Una decisión clave en el MODELO 6 es que el transporte de mensajes en los canales se haga en dos etapas: el mensaje es reenviado primero al dueño del canal quien disemina el mensaje resultante a todos los suscriptores del canal. Esto abre la posibilidad para que el dueño del canal lleve a cabo el procesamiento del mensaje recibido antes de que se disemine. El modelo también introduce otros dos tipos de mensaje – forward y otro – para resolver las necesidades de mensajes regulares, además de las operaciones especiales del Gateway consideradas hasta ahora. El MODELO 6 tiene el mismo conjunto de esquemas que el MODELO 5 – `ID`, `MESSAGE`, `QUEUE`, `CHANNEL`, `MEMBER`, `STATE`, `OPERATIONS`, `GATEWAY` e `INTERFACE` y las mismas relaciones estructurales entre ellos, sólo los esquemas `MESSAGE`, `OPERATIONS`, `GATEWAY` e `INTERFACE` han sido modificados.

Habiendo considerado los mensajes regulares, además de `mtype`, `sender`, `recipients` y `error`, el registro de `Message` también contiene el cuerpo del mensaje - `body`. Se introdujeron dos valores más en `MessageType` – `forward` y `other`.

```
type
  Message::
    mtype: MessageType
    sender: I.MemberId
    recepients: I.ChannelId
    error: Bool <-> re_error
    body: MessageBody

type
  MessageBody
value
  empty: MessageBody
```

```
type
  MessageType ==
    register(I.MemberId) |
    unregister(I.MemberId) |
    create(I.ChannelId, I.MemberId) |
    destroy(I.ChannelId, I.MemberId) |
    subscribe(I.MemberId, I.ChannelId) |
    unsubscribe(I.MemberId, I.ChannelId) |
    forward(I.MemberId) |
    other |
```

Con dos nuevos tipos de mensajes, el esquema `OPERATIONS` también debe contener las signaturas de las funciones para recibir los tipos de mensaje `forward` y `other`. Adicionalmente, la transferencia del mensaje en dos etapas hace necesario que todas las funciones de recibir sepan la identidad del miembro que recibe el mensaje. En especial, se debería saber si el receptor es dueño de un canal o suscriptor a fin de poder determinar cómo se debe tratar el mensaje. A su vez, esto requiere que las signaturas de todas las funciones en el esquema `OPERATIONS` sean extendidas con el argumento `MemberId`.

```
value
  register: I.MemberId >< M.Message >< S.State -~-> S.State,
  unregister: I.MemberId >< M.Message >< S.State -~-> S.State,
  forward: I.MemberId >< M.Message >< S.State -~-> S.State,
  other: I.MemberId >< M.Message >< S.State -~-> S.State,
  ...
  canRegister: I.MemberId >< M.Message >< S.State -> Bool,
  canUnregister: I.MemberId >< M.Message >< S.State -> Bool,
  canForward: I.MemberId >< M.Message >< S.State -> Bool,
  canOther: I.MemberId >< M.Message >< S.State -> Bool,
  ...
End
```

En `GATEWAY`, las funciones send y `transferAll` no han cambiado con respecto al MODELO 5. Este no es el caso para `receive` y `transfer`. La primera función es modificada para llamar a las funciones de recibir en el objecto `OPERATIONS` con el argumento adicional `MemberID` y para incluir los tipos `forward` y `other` en la expresión case, de la siguiente manera:

```
value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let
      (m, e) = E.dequeueOutbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s)
    in
      case M.mtype(m) of
        M.register(i) ->
          if O.canRegister(i, m, s')
          then (m, O.register(i, m, s')) else (M.re_error(true, m), s') end,
        ...
        M.forward(im) ->
          if O.canForward(i, m, s')
          then (m, O.forward(i, m, s')) else (M.re_error(true, m), s') end,
        M.other ->
          if O.canOther(i, m, s')
          then (m, O.other(i, m, s')) else (M.re_error(true, m), s') end
      end
    end
    pre canReceive(i, s),
```

Luego de recibir un mensaje del tipo `other`, el Gateway lo envía al dueño del canal quien luego lo disemina a todos los suscriptores del canal. Un comportamiento ya establecido para todos los mensajes intercambiados a través del Gateway, esto es capturado a través de la siguiente definición revisada de la función `transfer`. La función toma como argumentos al identificador del miembro receptor y el actual estado del Gateway, y devuelve un nuevo estado. Asumiendo que la cola de la bandeja de entrada del emisor no está vacía, `transfer` extrae y remueve el primer mensaje de la cola y determina el dueño del canal receptor en el mensaje. Si el receptor no es el dueño del canal, el mensaje es transferido a través de la función `transferAll` al dueño. De lo contrario, es transferido a todos los suscriptores.

```
value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      io = S.getOwner(M.recepients(m), s)
    in
      if i ~= io then transferAll(m, {io}, s') else
        let ids = C.subscribers(S.getChannel(M.recepients(m), s))
        in transferAll(m, ids, s') end
      end
    end
    pre canTransfer(i, s),
```

Como en el MODELO 5, el esquema `INTERFACE` contiene especificaciones de todas las funciones de enviar y recibir invocadas para operaciones de `register`, `unregister`, `create`, `destroy`, `subscribe` y `unsubscribe`. La única gran diferencia es la inclusión del identificador del receptor en todas las funciones de recibir. Además, `INTERFACE` también contiene funciones para dos nuevos tipos de mensajes definidos – `forward` and `other`.

Enviar un mensaje del tipo `forward` (`forwardSend`) requiere saber la identidad del remitente y del receptor; la precondición `canForwardSend` asegura que ambos existan. En consecuencia, el mensaje es formulado y enviado a `admin` a través del canal de administración del remitente. Para esto, se usa la operación `send` del objeto `GATEWAY` G.

```
value
  forwardSend: I.MemberId >< I.MemberId >< M.MessageBody >< S.State -~-> S.State
  forwardSend(i1, i2, b, s) is
    let
      mtype = M.forward(i2),
      sender = i1, recepients = I.adminChannel(i1)
    in G.send(M.mk_Message(mtype, sender, recepients, false, b), s) end
    pre canForwardSend(i1, i2, b, s),
```

```
canForwardSend: I.MemberId >< I.MemberId >< M.MessageBody >< S.State -> Bool
canForwardSend(i1, i2, b, s) is S.memberExists(i1, s) /\ S.memberExists(i2, s),
```

Luego de recibir un mensaje del tipo `forward`, `admin` cambia el tipo del mensaje a `other`, copia el cuerpo, y envía el mensaje al receptor a través de su canal de administración (`forwardReceive`). La precondición para la función requiere que el mensaje tenga el tipo `forward`, que el recipiente actual del mensaje sea `admin`, y que el receptor final exista en el estado (`canForwardReceive`).

```
value
  forwardReceive: I.MemberId >< M.Message >< S.State -~-> S.State
  forwardReceive(ir, m, s) is
    case M.mtype(m) of
      M.forward(i) ->
        let
          mtype = M.other,
          sender = I.admin,
          recepients = I.adminChannel(i)
        in G.send(M.mk_Message(mtype, sender, recepients, false, M.body(m)), s) end
    end
    pre canForwardReceive(ir, m, s),

  canForwardReceive: I.MemberId >< M.Message >< S.State -> Bool
  canForwardReceive(ir, m, s) is
    case M.mtype(m) of
      M.forward(i) -> S.memberExists(i, s) /\ ir = I.admin,
      _ -> false
    end
```

Enviar un mensaje tipo `other` requiere conocer la indentidad del remitente, el canal receptor y el cuerpo del mensaje (`otherSend`). La precondición afirma que el miembro y en canal existan, y que el miembro sea el dueño o un suscriptor de este canal (`canOtherSend`). En consecuencia el mensaje es formulado y enviado a través de la función send disponible a través de la instancia G del esquema `GATEWAY`.

```
value
  otherSend: I.MemberId >< I.ChannelId >< M.MessageBody >< S.State -~-> S.State
  otherSend(im, ic, b, s) is
    let
      mtype = M.other,
      sender = im, recepients = ic
    in G.send(M.mk_Message(mtype, sender, recepients, false, b), s) end
    pre canOtherSend(im, ic, b, s),

  canOtherSend: I.MemberId >< I.ChannelId >< M.MessageBody >< S.State -> Bool
  canOtherSend(im, ic, b, s) is
    S.memberExists(im, s) /\
    S.channelExists(ic, s) /\
    (C.isOwner(im, S.getChannel(ic, s)) \/ C.isSubscriber(im, S.getChannel(ic, s)))
```

Luego de recibido, se determina primero si el receptor es el dueño del canal a través del cual el mensaje fue recibido (`otherReceive`). Si no, el suscriptor acaba de recibir el mensaje y no se necesitan otras acciones – el estado es devuelto sin cambios. De lo contrario, el dueño envía el mensaje al todos los suscriptores del canal, usando la función send disponible a través de la instancia `G` del `GATEWAY`. La precondición afirma que tanto el receptor del mensaje como el canal a donde fue enviado el mensaje existen, y que el receptor es el dueño o un suscriptor de este canal (`canOtherReceive`).

```
value
  otherReceive: I.MemberId >< M.Message >< S.State -~-> S.State
  otherReceive(ir, m, s) is
    let
      ic = M.recepients(m),
      io = S.getOwner(ic, s)
    in
      if ir ~= io then s else
```

```
        G.send(M.mk_Message(M.mtype(m), ir, ic, false, M.body(m)), s)
      end
    end
    pre canOtherReceive(ir, m, s),

  canOtherReceive: I.MemberId >< M.Message >< S.State -> Bool
  canOtherReceive(ir, m, s) is
    let ic = M.recepients(m) in
      S.memberExists(ir, s) /\ S.channelExists(ic, s) /\
      (C.isOwner(ir, S.getChannel(ic, s)) \/ C.isSubscriber(ir, S.getChannel(ic, s)))
    end
```

4.6.7 MODELO 7 – Mensajería Programable Dirigida por Tipos

Todos los modelos anteriores consideran la recepción de mensajes de diferentes tipos y los cambios de estado correspondientes causados por tales mensajes, caso por caso. De este modo, la presencia de una extensa expresión case en la función `receive` del `GATEWAY`, con casos `register`, `unregister`, `create`, `destroy`, `subscribe`, `unsubscribe`, `follow` y `other` llamando a las funciones de recibir correspondientes del esquema `INTERFACE`, donde cada una efectúa los cambios requeridos al estado. Como resultado, el agregar nuevos tipos de mensajes con sus propios requerimientos de procesamiento y mensajería requerirían cambios invasivos al Gateway propiamente dicho. El objetivo del MODELO 7 es el de equipar al Gateway con acciones genéricas y programables para modificar el estado del Gateway, y aplicar tales acciones para llevar a cabo el procesamiento para los tipos de mensaje existentes.

El modelo consta de todos los esquemas introducidos en el MODELO 6 – `ID`, `MESSAGE`, `QUEUE`, `CHANNEL`, `MEMBER`, `STATE`, `GATEWAY` y `INTERFACE`, en donde ningún esquema, salvo GATEWAY e INTERFACE sufrió alteraciones. Además, el modelo contiene los esquemas `ACTION`, `PREDICATE` y `NUMBER` para definir expresiones de acción, predicado y números respectivamente, todos relacionados entre sí, y todos ejecutados en un estado dado para devolver: un nuevo estado (para la expresión de acción), un valor Booleano (para la expresión de predicado) y un número natural (para la expresión de número). Las relaciones entre estos esquemas, hasta el grado en que difieren de las presentadas en el MODELO 6, se definen debajo:

```
scheme ACTION(
         I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C),
         S:STATE(I,M,C,E), O:OPERATIONS(I,M,C,E,S), G:GATEWAY(I,M,C,E,S,O)) = ...
scheme PREDICATE(
         I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C),
         S:STATE(I,M,C,E), O:OPERATIONS(I,M,C,E,S), G:GATEWAY(I,M,C,E,S,O)) = ...
scheme NUMBER(
         I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C),
         S:STATE(I,M,C,E), O:OPERATIONS(I,M,C,E,S), G:GATEWAY(I,M,C,E,S,O)) = ...
scheme INTERFACE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) =
class
  object
    O: OPERATIONS(I,M,C,E,S),
    G: GATEWAY(I,M,C,E,S,O),
    A: ACTION(I,M,C,E,S,O,G),
    P: PREDICATE(I,M,C,E,S,O,G),
    N: NUMBER(I,M,C,E,S,O,G)
  ...
end
```

El tipo `Action` contiene 13 tipos de expresiones para cambiar el estado del Gateway para: agregar un nuevo miembro (`addNewMember`), eliminar un miembro (`deleteMember`), agregar un canal de administración (`addAdminChannel`), eliminar un canal de administración (`deleteAdminChannel`), agregar un nuevo canal perteneciente a un miembro (`addChannelOwned`), eliminar un canal perteneciente a un miembro (`deleteChannelOwned`), agregar un suscriptor a canal del dueño (`addSubscriber`), eliminar un suscriptor del canal del dueño (`deleteSubscriber`), suscribir un miembro a un canal (`addChannelSubscribed`), desuscribir un miembro de un canal (`deleteChannelSubscribed`), enviar un mensaje con un determinado tipo, remitente, receptores y flag de error (send), ejecutar una de dos acciones dependiendo si se satisface un predicado (cond), y ejecutando dos acciones, una después de la otra (seq). El tipo se define de la siguiente manera:

```
type
  Action ==
    addNewMember(I.MemberId) |
    addAdminChannel(I.MemberId) |
    deleteMember(I.MemberId) |
    deleteAdminChannel(I.MemberId) |
    addChannelOwned(I.ChannelId, I.MemberId) |
    deleteChannelOwned(I.ChannelId, I.MemberId) |
    addSubscriber(I.ChannelId, I.MemberId) |
    addChannelSubscribed(I.ChannelId, I.MemberId) |
    deleteSubscriber(I.ChannelId, I.MemberId) |
    deleteChannelSubscribed(I.ChannelId, I.MemberId) |
    send(M.MessageType, I.MemberId, I.ChannelId, Bool) |
    cond(O.Predicate, Action, Action) |
    seq(Action, Action)
```

Las acciones son ejecutadas a través de la función exec que, dado una expresión de acción, el identificador de un miembro receptor, el mensaje mismo y el estado actual del Gateway, devuelve un nuevo estado. Las expresiones que no contengan más acciones invocan a las expresiones correspondientes en el estado. Las expresiones que sí contengan más acciones, como cond o seq, son ejecutadas recursivamente en base a la estructura de las expresiones de acción. Por ejemplo, la composición secuencial ejecuta la primera expresión de acción en el estado actual y luego la segunda expresión en el estado resultante, mientras que la composición condicional ejecuta primero el predicado (a través de la función exec provista por OPERATIONS) y luego ejecuta la primera o segunda expresión de acción, dependiendo si el predicado fue `true` o `false`.

```
value
  exec: Action >< I.MemberId >< M.Message >< S.State -~-> S.State
  exec(a, ir, m, s) is
    case a of
      addNewMember(im) -> S.addMember(im, E.init(im), s),
      addAdminChannel(im) -> S.addAdminChannel(im, s),
      deleteMember(im) -> S.deleteMember(im, s),
      deleteAdminChannel(im) -> S.deleteAdminChannel(im, s),
      addChannelOwned(ic, im) -> S.addChannelOwned(ic, im, s),
      deleteChannelOwned(ic, im) -> S.deleteChannelOwned(ic, im, s),
      addSubscriber(ic, im) -> S.addSubscriber(ic, im, s),
      addChannelSubscribed(ic, im) -> S.addChannelSubscribed(ic, im, s),
      deleteSubscriber(ic, im) -> S.deleteSubscriber(ic, im, s),
      deleteChannelSubscribed(ic, im) -> S.deleteChannelSubscribed(ic, im, s),
      send(mt, im, ic, b) -> G.send(M.mk_Message(mt, im, ic, b, M.body(m)), s),
      cond(p, a1, a2) ->
        if O.exec(p, ir, m, s)
        then exec(a1, ir, m, s) else exec(a2, ir, m, s) end,
      seq(a1, a2) -> let s' = exec(a1, ir, m, s) in exec(a2, ir, m, s') end
    end
    pre O.exec(iswf(a, ir, m, s), ir, m, s),
```

.

La función de ejecución para las expresiones de acción es condicional de la función `iswf` que devuelve una expresión de predicado que representa la precondición de la expresión de acción. El resultado es ejecutado a través de la función exec provista por OPERATIONS, finalmente devolviendo `true` or `false`. La función `iswf` es ejecutada en la estructura de las expresiones de acción y devuelve las expresiones de predicado según el tipo `Predicate`. Por ejemplo, para agregar a un nuevo suscriptor (`addSubscriber`), el canal debe existir (`channelExists`) y el miembro no debe estar suscripto a este canal (`isChannelSubscriber`). Otro ejemplo es la composición secuencial en donde ambas acciones deben ser bien formadas, la primera en el estado original, pero la segunda en el estado obtenido al ejecutar la primera acción.

```
value
  iswf: Action >< I.MemberId >< M.Message >< S.State -~-> O.Predicate
  iswf(a, ir, m, s) is
    case a of
      addNewMember(im) -> O.not(O.memberExists(im)),
      addAdminChannel(im) -> O.not(O.channelExists(I.adminChannel(im))),
      deleteMember(im) -> O.memberExists(im),
      deleteAdminChannel(im) -> O.channelExists(I.adminChannel(im)),
      addChannelOwned(ic, im) -> O.not(O.channelExists(ic)),
      deleteChannelOwned(ic, im) -> O.isChannelOwner(im, ic),
      addSubscriber(ic, im) ->
        O.and(O.channelExists(ic), O.not(O.isChannelSubscriber(im, ic))),
      addChannelSubscribed(ic, im) -> O.andAll(<.
        O.channelExists(ic),
        O.not(O.isChannelOwner(im, ic)),
        O.not(O.isChannelSubscriber(im, ic)).>),
      deleteSubscriber(ic, im) ->
        O.and(O.channelExists(ic), O.isChannelSubscriber(im, ic)),
      deleteChannelSubscribed(ic, im) -> O.isChannelSubscribed(ic, im),
      send(mt, im, ic, b) -> O.andAll(<.
        O.memberExists(im), O.channelExists(ic),
        O.or(O.isChannelOwner(im, ic), O.isChannelSubscriber(im, ic)).>),
      cond(p, a1, a2) -> O.and(iswf(a1, ir, m, s), iswf(a2, ir, m, s)),
      seq(a1, a2) -> O.and(iswf(a1, ir, m, s), iswf(a2, ir, m, exec(a1, ir, m, s)))
    end
```

La función `getAction` construye una expresión de acción adecuada para ser ejecutada cuando un miembro recibe varios tipos de mensaje. Por ejemplo: luego de recibir un mensaje `register`, el receptor debería agregar secuencialmente a un nuevo miembro (`addNewMember`) y agregar el canal de administración correspondiente (`addAdminChannel`); luego de recibir un mensaje `unregister` el receptor debería secuencialmente borrar el miembro (`deleteMember`) y a su canal de administración (`deleteAdminChannel`); luego de recibir un mensaje `subscribe`, el receptor debería agregar al miembro como suscriptor del canal, según los registros del dueño (`addSubscriber`) y del suscriptor (`addChannelSubscriber`); luego de recibir un mensaje `forward`, el receptor debería enviarlo a `admin` a través de su canal de administración (`send`); y luego de recibir un mensaje `other`, el receptor debería enviarlo a todos los suscriptores del canal (`send`).

```
value
  getAction: I.MemberId >< M.Message -> Action
  getAction(ir, m) is
    case M.mtype(m) of
      M.register(im) -> seq(addNewMember(im), addAdminChannel(im)),
      M.unregister(im) -> seq(deleteMember(im), deleteAdminChannel(im)),
      M.create(ic, im) -> addChannelOwned(ic, im),
      M.destroy(ic, im) -> deleteChannelOwned(ic, im),
      M.subscribe(im, ic) -> seq(addSubscriber(ic, im), addChannelSubscribed(ic, im)),
      M.unsubscribe(im, ic) ->
        seq(deleteSubscriber(ic, im), deleteChannelSubscribed(ic, im)),
      M.forward(im) -> send(M.other, I.admin, I.adminChannel(im), false),
      M.other -> send(M.other, ir, M.recepients(m), false)
    end

end
```

Las expresiones de predicado son formuladas en el esquema `PREDICATE` de una forma similar. Primero, definir el tipo `Predicate` con varios tipos de expresiones para formular preguntas acerca del estado del Gateway: ¿Es `admin` el receptor de un mensaje? ¿Existe un miembro determinado? ¿Existe un canal determinado? ¿Es un miembro determinado `admin`? ¿Es un miembro determinado `visitor`? ¿Es un miembro determinado el dueño de un canal determinado? ¿Es un miembro determinado un suscriptor de un canal determinado según el registro del dueño? ¿Según el registro del suscriptor? ¿Es un canal determinado un canal de administración? Además, constructores lógicos son provistos por el tipo `Predicate` representando: `true`, `false`, negación, conjunción, disyunción, conjunción extendida e igualdad entre dos expresiones de números.

```
type
  Predicate ==
```

```
    isReceiverAdmin |
    memberExists(I.MemberId) |
    channelExists(I.ChannelId) |
    isMemberAdmin(I.MemberId) |
    isMemberVisitor(I.MemberId) |
    isChannelOwner(I.MemberId, I.ChannelId) |
    isChannelSubscriber(I.MemberId, I.ChannelId) |
    isChannelSubscribed(I.ChannelId, I.MemberId) |
    isAdminChannel(I.ChannelId) |
    cons(Bool) |
    not(Predicate) |
    and(Predicate, Predicate) |
    or(Predicate, Predicate) |
    andAll(Predicate-list) |
    equal(O.Number, O.Number)
```

Las expresiones de predicados son evaluadas a través de la siguiente función exec que, dada una expresión, el receptor de un mensaje, el mensaje mismo y el estado del Gateway, devuelve `true` or `false`. Por ejemplo, la expresión `isChannelSubscriber` es evaluada al ejecutar la función `isSubscriber` del esquema STATE sobre el miembro, el canal y el estado, mientras que `equal(n1,n2)` es evaluado al ejecutar la función exec del esquema NUMBER en forma separada en n1 y n2, y comparando si los resultados son iguales.

```
value
  exec: Predicate >< I.MemberId >< M.Message >< S.State -> Bool
  exec(p, ir, m, s) is
    case p of
      cons(b) -> b,
      isReceiverAdmin -> ir = I.admin,
      memberExists(im) -> S.memberExists(im, s),
      isMemberAdmin(im) -> im = I.admin,
      isMemberVisitor(im) -> im = I.visitor,
      isChannelOwner(im, ic) -> S.isChannelOwned(ic, im, s),
      isChannelSubscriber(im, ic) -> S.isSubscriber(im, ic, s),
      isChannelSubscribed(ic, im) -> S.isChannelSubscribed(ic, im, s),
      isAdminChannel(ic) -> I.isAdminChannel(ic),
      not(p') -> ~exec(p', ir, m, s),
      and(p1, p2) -> exec(p1, ir, m, s) /\ exec(p2, ir, m, s),
      or(p1, p2) -> exec(p1, ir, m, s) \/ exec(p2, ir, m, s),
      andAll(pl) ->
        pl = <..> \/ exec(hd pl, ir, m, s) /\ exec(andAll(tl pl), ir, m, s),
      equal(n1, n2) -> O.exec(n1, ir, m, s) = O.exec(n2, ir, m, s)
    end
```

Un equivalente de `getAction`, la función `getPrecondition` devuelve una expresión de predicado que representa una precondición de la acción generada para un tipo de mensaje determinado. Por ejemplo, luego de recibir un mensaje `register`, la precondición es que el miembro no esté registrado y que el receptor del mensaje sea `admin`. Luego de recibir un mensaje `unregister`, la precondición es que el receptor sea `admin`, que el miembro exista, pero que no sea ni `admin` ni `visitor`, que el miembro no sea dueño de ningún canal y que esté suscripto a un solo canal – su canal de administración. Luego de recibir un mensaje `forward`, la precondición es que el receptor sea `admin` y que exista el miembro a quien se le reenvía el mensaje. Luego de recibir un mensaje `other`, la precondición es que existan el miembro receptor y el canal de destino, y que el miembro sea o dueño de un canal o que esté suscripto a él.

```
value
  getPrecondition: I.MemberId >< M.Message -> Predicate
  getPrecondition(ir, m) is
    case M.mtype(m) of
      M.register(im) -> and(not(memberExists(im)), isReceiverAdmin),
      M.unregister(im) ->
        andAll(<. isReceiverAdmin, memberExists(im), not(isMemberAdmin(im)),
          not(isMemberVisitor(im)), equal(O.channelsOwned(im), O.constant(0)),
          equal(O.channelsSubscribed(im), O.constant(1)).>),
      M.create(ic, im) ->
        andAll(<.isReceiverAdmin, not(channelExists(ic)), memberExists(im),
          not(isMemberVisitor(im)).>),
      M.destroy(ic, im) ->
```

```
        andAll(<.isReceiverAdmin, channelExists(ic), isChannelOwner(im, ic),
          not(isAdminChannel(ic)), equal(O.channelSubscribers(ic), O.constant(0)).>),
      M.subscribe(im, ic) ->
        andAll(<.isReceiverAdmin, memberExists(im), channelExists(ic),
          not(isChannelSubscriber(im, ic)), not(isChannelOwner(im,ic)),
          not(isMemberVisitor(im)).>),
      M.unsubscribe(im, ic) ->
        andAll(<.isReceiverAdmin, memberExists(im), channelExists(ic),
          isChannelSubscriber(im, ic), not(isAdminChannel(ic)).>),
      M.forward(im) -> and(memberExists(im), isReceiverAdmin),
      M.other ->
        andAll(<.memberExists(ir), channelExists(M.recepients(m)),
          or(isChannelOwner(ir, M.recepients(m)),
          isChannelSubscriber(ir, M.recepients(m))).>)
    end
```

Las expresiones de números son formulados usando las funciones `channelsOwned`, `channelsSubscribed` and `channelSubscribers`, junto con constantes numéricas (`constant`) y el operador de adición (`add`). Como se espera, la ejecución de tales expresiones en el miembro receptor, el mensaje y el estado devuelve un número natural.

```
type
  Number ==
    constant(Nat) |
    channelsOwned(I.MemberId) |
    channelsSubscribed(I.MemberId) |
    channelSubscribers(I.ChannelId) |
    add(Number, Number)
value
  exec: Number >< I.MemberId >< M.Message >< S.State -> Nat
  exec(n, ir, m, s) is
    case n of
      constant(m) -> m,
      channelsOwned(im) -> card S.channelsOwned(im, s),
      channelsSubscribed(im) -> card S.channelsSubscribed(im, s),
      channelSubscribers(ic) -> card S.subscribers(ic, s),
      add(n1, n2) -> exec(n1, ir, m, s) + exec(n2, ir, m, s)
    end
```

Una vez que los esquemas ACTION, PREDICATE y NUMBER están definidos, sus tipos y signaturas de funciones están disponibles para ellos, así como para los esquemas GATEWAY e INTERFACE a través del esquema OPERATIONS:

```
type
  Action, Predicate, Number

value
  exec: Action >< I.MemberId >< M.Message >< S.State -~-> S.State,
  exec: Predicate >< I.MemberId >< M.Message >< S.State -~-> Bool,
  exec: Number >< I.MemberId >< M.Message >< S.State -~-> Nat,
  iswf: Action >< I.MemberId >< M.Message >< S.State -> Bool,
  getAction: I.MemberId >< M.Message -> Action,
  getPrecondition: I.MemberId >< M.Message -> Predicate
```

En el esquema GATEWAY, la principal diferencia es la definición de la función `receive`. La función ya no contiene una expresión case que considera de manera separada la recepción de mensajes con diferentes tipos de mensaje. En cambio, luego de extraer y remover el primer mensaje de la cola de la bandeja de salida del receptor, `receive` genera la acción (`getAction`) y su precondición (`getPrecondition`) para el mensaje y su receptor, y verifica si se cumple la precondición generada en el estado actual del Gateway. Si es así, la acción generada es ejecutada y el estado resultante es devuelto junto con el mensaje mismo. De lo contrario, se setea en `true` el flag de `error` dentro del mensaje que luego es devuelto por `receive` junto con el nuevo estado.

```
value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let
      (m, e) = E.dequeueOutbox(S.getMember(i, s)),
```

```
      s' = S.modifyMember(i, e, s)
    in
      let
        a = O.getAction(i, m), p = O.getPrecondition(i, m)
      in
        if O.exec(p, i, m, s)
        then (m, O.exec(a, i, m, s'))
        else (M.re_error(true, m), s') end
      end
    end pre canReceive(i, s),
```

En el esquema INTERFACE, el único cambio es la eliminación de las funciones de recibir para todas las operaciones. Como receive es ejecutada internamente por el Gateway basándose en los tipos de mensajes recibidos, ya no están disponibles para usuarios externos, quienes sólo tienen acceso a las operaciones de envío. Por ejemplo, la operación de envío para mensajes other es:

```
value
  other: I.MemberId >< I.ChannelId >< M.MessageBody >< S.State -~-> S.State
  other(im, ic, b, s) is
    let
      mtype = M.other,
      sender = im, recepients = ic
    in G.send(M.mk_Message(mtype, sender, recepients, false, b), s) end
    pre canOther(im, ic, b, s)
```

4.7 Discusión

Este capítulo presentó el Government-Enterprise Ecosystem Gateway (G-EEG) como una realización concreta del concepto de Mensajería Programable. El capítulo introdujo el modelo conceptual para el Gateway, construido sobre los conceptos de miembro, canal, mensaje y extensión; introdujo una notación gráfica para representar estructuras de comunicación dinámicas creadas por el Gateway y el comportamiento de mensajería facilitado por ellas; explicó la arquitectura del Gateway con componentes que soportan la mensajería básica y extendida, y el desarrollo de nuevas extensiones; introdujo un rango de comportamientos de mensajería básicos y extendidos – horizontal y vertical – por por comportamientos de mensajería realizado por los miembros del Gateway, usando la notación gráfica introducida; explicó un rango de extensiones para auditoría, validación, transformación, criptografía, autenticación, descubrimiento, alianza, orden, puntualidad y composición; y presentó siete pasos de formalización para el Gateway, desde el intercambio directo de mensajes entre miembros del Gateway hasta comportamientos de mensajería programables más enriquecidos y más flexibles, usando la notación formal de RSL.

Con respecto al estado del desarrollo de varios componentes del Gateway – básico, extendido y desarrollo, como se indican en este capítulo y en el siguiente (implementación): el componente de mensajería básica ha sido totalmente especificado usando la notación gráfica, formalizado usando RSL e implementado con Java; el componente extendido ha sido totalmente especificado, implementado (aunque con un grupo de extensiones limitado) en Java y preparado para la formalización; mientras que el componente de desarrollo está listo para la especificación y formalización, parte de un trabajo futuro.

La formalización procedió a través de una serie de siete modelos del Gateway de complejidad y comportamiento incremental: MODELO 1 – mensajería directa entre miembros registrados dinámicamente; MODELO 2 – mensajería indirecta entre miembros a través de canales creados y suscriptos dinámicamente; MODELO 3 – mensajería con miembros y canales designados que proveen soporte administrativo; MODELO 4 - distribución del estado del portal entre sus miembros; MODELO 5 – introducción de varios tipos de mensaje para indicar el procesamiento y la mensajería requerida por el Gateway, y ejecución de todas las operaciones del Gateway a través del intercambio de mensajes; MODELO 6 – introducción de la entrega de mensajes en dos etapas, desde el remitente al dueño del canal y desde el dueño a todos los suscriptores del canal; y MODELO 7 – introducción de acciones programables genéricas, con términos y predicados subyacentes, para determinar el procesamiento requerido para los mensajes, al igual que cualquier intercambio de mensajes a posteriori por parte de los miembros para varios tipos de mensajes.

Desde su estado actual de desarrollo, el modelo puede ser extendido naturalmente con las siguientes características:

8) MODEL 8 – Mensajería de seguimiento y acciones a través de procesos. Más allá del procesamiento de mensajes por sus receptores inmediatos, determinado por las acciones generadas por diferentes tipos de mensajes, proveer el apoyo para el procesamiento y para la mensajería de seguimiento por receptores de segunda-capa, tercera-capa y posteriores receptores de tales mensajes. Esto puede ser realizado limitando las acciones a una ejecución local, actuando solamente sobre el mensaje y el estado local de un solo miembro receptor, e introduciendo procesos para facilitar la ejecución global con acciones locales y funciones de mensajería globales, todos combinados dentro de expresiones de proceso. Esto requeriría definir el estado actual de un proceso e insertar el proceso y el estado dentro de un mensaje. Luego de la recepción, el estado determinaría la siguiente acción a ejecutar, y la mensajería de seguimiento involucraría enviar más mensajes con el nuevo estado.

9) MODELO 9 – Un modelo de acciones/proceso para definir/ejecutar extensiones de canal. Con el envío de mensajes en dos etapas, usar la recepción de mensajes por los dueños de canales para invocar cualquier extensión habilitada en el canal, enviando el mensaje al administrador de la extensión correspondiente, recibiendo la respuesta, enviando el resultado a más extensiones, etc. En vez de invocar extensiones en un orden predefinido, usar el modelo de proceso del MODELO 8 para combinarlas de manera más poderosa y flexible, basándose en tipos de extensiones para determinar el tipo de intercambio de mensajes (send-forget, send-reply, etc) requerido por el administrador de la extensión. Al mismo tiempo, usar el modelo del MODELO 7 para implementar varias extensiones a través de acciones programables.

10) MODELO 10 – Unificar estructuras de estado, mensaje y mensajería a través de XML. Basados en el modelo formal de XML y la familia de lenguajes XML para ayudar a la validación, transformación, accesibilidad, cálculos y otras acciones sobre documentos XML (Apéndice A.8), representar las estructuras de estado, mensaje y mensajería usando el modelo XML. Posteriormente, usar la familia de lenguajes XML para llevar a cabo el procesamiento de las estructuras de estado, mensaje y mensajería requeridas de una manera uniforme. Este paso, sumado a la uniformidad lograda en la representación y procesamiento de varias estructuras, acercará el modelo del Gateway a la implementación actual.

Más modelos son ciertamente posibles y serán tema de trabajos futuros.

Capítulo 5

Implementación de Mensajería Programable

El modelo formal introducido en el Capítulo 4 provee fundamento para construir la solución de mensajería provista por la tesis. Para ilustrar la factibilidad de la solución propuesta, se desarrolló un prototipo de software de investigación. El prototipo implementa un subconjunto del modelo formal, con modelo e implementación basados en el conjunto de conceptos presentados en la Sección 4.1. En particular, el prototipo implementa: el núcleo de los servicios de comunicaciones – registrar y des-registrar miembros, crear y destruir canales, suscribir y des-suscribir miembros a canales, y enviar y recibir mensajes; y tres extensiones horizontales – auditoría de mensajes en tránsito, validación del formato de los mensajes y transformación de mensajes de un formato a otro.

El objetivo de este capítulo es introducir el desarrollo y uso del prototipo, incluyendo la presentación de artefactos seleccionados de desarrollo; el Apéndice B presenta una versión más completa de dichos artefactos. El desarrollo fue llevado a cabo usando UML y los artefactos resultantes están documentados como varios tipos de diagramas UML. Específicamente, el capítulo documenta: requerimientos, casos de uso, arquitectura, diseño, implementación, entrega y uso. También se ilustra de qué manera se puede usar el prototipo como parte de la infraestructura de software para Gobierno Integrado.

El capítulo está organizado en siete secciones, siguiendo varias etapas del desarrollo del prototipo. La Sección 5.1 presenta requerimientos funcionales y no funcionales, seguidos por los casos de uso para modelar los requerimientos funcionales en la Sección 5.2. Las Secciones 5.3 y 5.4 presentan la arquitectura y diseño detallado del prototipo por medio de diagramas estructurales y de comportamiento, respectivamente. La implementación es presentada en la Sección 5.5 y un posible escenario de entrega es descripto en la Sección 5.6. Finalmente, el uso del software es descripto en la Sección 5.7.

5.1 Requerimientos

Esta sección describe los requerimientos funcionales y no funcionales para el prototipo G-EEG. Los requerimientos se basan en tres conceptos principales de G-EEG: Miembros – usuarios registrados del sistema que ejecutan servicios de mensajería; Canales – estructuras lógicas que permiten la comunicación entre miembros; y Mensajes – unidades de comunicación entre miembros. Basado en estos conceptos, el prototipo provee servicios para: registrar y des-registrar miembros, crear y destruir canales, suscribir miembros a canales, des-suscribir miembros de canales, y enviar y recibir mensajes. Adicionalmente, G-EEG define el concepto de extensiones horizontales (independientes de procesos) y el prototipo implementa tres extensiones concretas para auditoría, validación y transformación de mensajes.

El resto de esta sección presenta cinco grupos de requerimientos, cuatro que comprenden requerimientos funcionales relacionados con miembros, canales, mensajes y extensiones y el quinto que comprende requerimientos no funcionales (ver Tabla 18):

1) *Requerimientos Relacionados con Miembros* – Los requerimientos F1, F2 y F3 definen la implementación de miembros visitantes, usuarios y administrador. Hay dos operaciones relacionadas: registración (F4) – requerida por un miembro visitante, permite a una entidad la creación de un nuevo miembro por medio de la cual es capaz de usar servicios de mensajería; y des-registración (f5) – requerida por el mismo miembro, remueve al miembro del sistema cuando no necesita más servicios de mensajería. Una operación (F6) es adicionalmente requerida por razones de implementación, solicitada por un miembro visitante y dirigida a recuperar un miembro después de

reiniciar el sistema. Todas las operaciones son llevadas a cabo por miembros autorizados enviando mensajes al administrador y esperando por confirmación.

2) *Requerimientos Relacionados con Canales* – Los requerimientos F7 y F8 definen la implementación de canales para administrador y visitante. Mientras que tales canales están siempre disponibles, los canales definidos por el usuario deben ser creados explícitamente y suscriptos antes de ser usados. Para crear un canal definido por el usuario, un miembro envía un requerimiento al administrador especificando el nombre del canal. El administrador confirma si la operación fue satisfactoria, en cuyo caso asigna un identificador único al canal, o de lo contrario envía un mensaje de error al miembro solicitante. Después de que el canal es creado, el propietario puede solicitar su destrucción enviando un requerimiento al administrador. El administrador concederá este requerimiento sólo si el canal no tiene suscriptores y no es un canal predefinido. Los requerimientos F9 y F10 especifican las funciones para creación y destrucción de canales. Los Miembros pueden también solicitar la suscripción o des-suscripción a canales enviando el pedido a los propietarios de los canales. Los requerimientos F11 y F12 especifican las funciones para suscripción y des-suscripción de miembros a canales.

3) *Requerimientos Relacionados con Mensajes* – Cuatro requerimientos son definidos para manejar la estructura de mensajes – Para componer un mensaje desde sus elementos constitutivos (F13) y para descomponer un mensaje para obtener sus elementos constitutivos: cabecera de mensaje (F14), cuerpo de mensaje (F15), y adjuntos al mensaje (F16). Cualquier miembro que suscriba o posea un canal puede enviar un mensaje por medio de dicho canal, el comportamiento depende de quién sea el emisor del mensaje. Si es enviado por el propietario del canal – requerimiento F17, el mensaje es procesado y enviado a todos los suscriptores; si el proceso falla, el propietario envía un mensaje de notificación de falla al emisor. Si es enviado por un suscriptor – requerimiento F18, el mensaje es reenviado al propietario del canal quien procesa el mensaje como se describió anteriormente. Una vez enviado satisfactoriamente al canal, cualquier miembro suscriptor o propietario del canal, excepto el emisor, recibirá el mensaje – requerimiento F19. Además de canales definidos por el usuario, un miembro puede enviar mensajes usando canales predefinidos. En particular, un mensaje puede ser reenviado entre dos miembros por medio del miembro administrador y canales de administración del usuario – requerimiento F20.

4) *Requerimientos Relacionados con las Extensiones* – Para responder a una variedad de requerimientos de mensajería que poseen las aplicaciones que se comunican, más allá del núcleo de mensajería habilitado por G-EEG-CORE, G-EEG introduce un mecanismo de extensión llamado G-EEG-EXTEND. G-EEG-EXTEND comprende un repositorio de extensiones y un mecanismo para habilitar, configurar o deshabilitar dichas extensiones sobre G-EEG-CORE en tiempo de ejecución. El requerimiento F21 define el manejo de las extensiones en el repositorio, incluyendo la entrega de nuevas extensiones y remoción de las existentes. Después que una extensión es correctamente adicionada al repositorio, el administrador puede habilitar (F22), deshabilitar (F23) o configurar (F24) dicha extensión sobre un canal, a requerimiento de su propietario. Lo último responde a la necesidad de algunas extensiones de requerir parámetros. Un ejemplo es el formato de mensaje requerido por las extensiones de validación. Tres extensiones concretas son implementadas: Auditoría de Canal – creando un registro histórico de los mensajes transferidos a través del canal usando una base de datos especializada en grabación de documentos XML (F25); Validación de Mensajes – validando la sintaxis de mensajes destinados a un canal, de acuerdo al formato de mensaje expresado con un esquema XML [W3C04c] (F26); y Transformación de Mensajes – transformando los mensajes en tránsito a través de un canal de un formato a otro, de acuerdo a la transformación de mensajes expresada en XSLT [W3C99a] (F27).

5) *Requerimientos No Funcionales* – Tres requerimientos no funcionales tratan los atributos de dependabilidad de G-EEG: el miembro administrador está siempre disponible en el sistema (NF28), todos los mensajes enviados deben ser entregados al menos una vez (NF29), y los mensajes deben ser entregados en la misma secuencia en que fueron enviados (NF30).

Tabla 18: Requerimientos de G-EEG

TIPO	CATEGORÍA	ID	DESCRIPCION
Funcional	Miembros	F1	Proveer Miembro Visitante
		F2	Proveer Miembro Usuario
		F3	Proveer Miembro Administrador

		F4	Registrar Miembro Nuevo
		F5	Des-registrar Miembro Existente
		F6	Recuperar Miembro
Funcional	Canales	F7	Proveer Canal de Administración
		F8	Proveer Canal Visitante
		F9	Crear Canal
		F10	Destruir Canal
		F11	Suscribir Miembro a Canal
		F12	Des-suscribir Miembro de Canal
Funcional	Mensajes	F13	Componer Mensaje
		F14	Obtener Cabecera de Mensaje
		F15	Obtener Cuerpo de Mensaje
		F16	Obtener Adjunto de Mensaje
		F17	Enviar Mensaje como Propietario de Canal
		F18	Enviar Mensaje como Suscriptor de Canal
		F19	Recibir Mensaje
		F20	Reenviar Mensaje entre Miembros
Funcional	Extensiones	F21	Administrar Repositorio de Extensiones
		F22	Habilitar Extensión
		F23	Configurar Extensión
		F24	Deshabilitar Extensión
		F25	Proveer Extensión de Auditoría
		F26	Proveer Extensión de Validación
		F27	Proveer Extensión de Transformación
No Funcional	Disponibilidad	NF28	Disponibilidad del Miembro Administrador
		NF29	Aseguramiento de Al-Menos-Una Entrega
		NF30	Aseguramiento de Entrega-No-Permutada

La especificación completa de los requerimientos se incluye en el Apéndice B.

5.2 Modelado

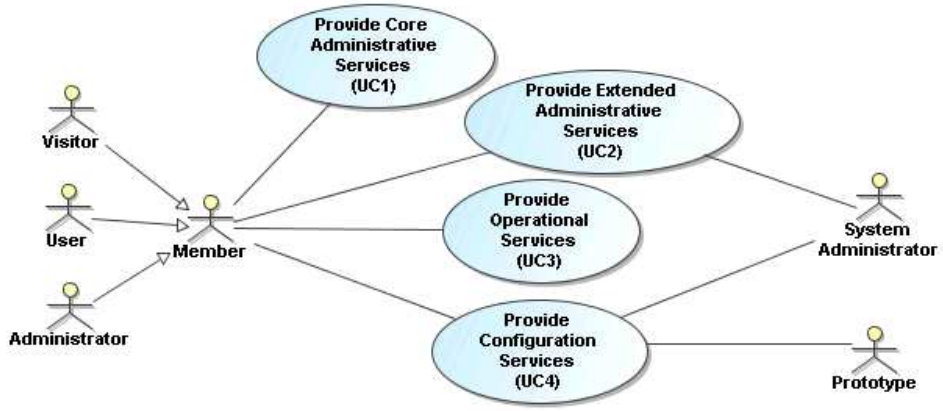
Esta sección presenta el modelo de casos de uso para implementar el prototipo. El modelo presenta los servicios que implementan los requerimientos funcionales definidos en la Sección 5.1, con cuatro grupos de servicios identificados:

1) *Servicios Administrativos Básicos* (UC1) – Requeridos por el administrador y los miembros usuario para crear y modificar las estructuras usadas para comunicación. Los servicios incluidos en este grupo son: registrar y des-registrar miembros, crear y destruir canales, suscribir y des-suscribir miembros a/de canales.
2) *Servicios Operacionales* (UC2) – Requeridos por todos los miembros para enviar y recibir mensajes.
3) *Servicios Administrativos Extendidos* (UC3) – Requeridos por el miembro administrador para entregar nuevas extensiones, y por miembros usuario para habilitar, configurar y deshabilitar extensiones.
4) *Servicios de Configuración* (UC4) – Requeridos por el miembro administrador para personalizar parámetros de configuración y establecer las estructuras de comunicación requeridas por el sistema.

La Figura 60 describe el diagrama de casos de uso de alto nivel.

Los Servicios Administrativos Básicos especifican funciones para crear y modificar las estructuras de comunicaciones usadas por los miembros para intercambiar mensajes, tales como: registrar y des-registrar miembros, crear y destruir canales, suscribir y des-suscribir miembros a/de canales. Adicionalmente, los servicios para reenviar mensajes entre miembros y para reiniciar miembros son también provistos. El diagrama de casos de uso es presentado en la Figura 61 y explicado a continuación:

Figura 60: Diagrama de Casos de Uso de Alto Nivel



1) *Registrar Miembro* – Un visitante envía un requerimiento al administrador a través del canal visitante para registrar un miembro incluyendo el nombre del nuevo miembro. Si es satisfactorio, el administrador asigna un nuevo identificador al usuario, modifica la base de datos central agregando al nuevo miembro, y contesta al visitante, mientras que el visitante crea el nuevo miembro y modifica la base de datos local. Los casos de uso UC5 (por visitante) y UC6 (por administrador) implementan los requerimientos F2 y F4.

2) *Des-Registrar Miembro* – Un usuario envía un mensaje de requerimiento al administrador para ser removido por medio del canal de administración. Después de recibir este mensaje, el administrador verifica si el usuario no posee algún recurso como propietario o suscriptor, de ser así envía un mensaje al usuario para conceder el requerimiento y borra al usuario de la base de datos central. Al recibir el mensaje, el usuario elimina sus datos de la base de datos local y destruye su propio objeto. Si el usuario posee recursos, el administrador envía un mensaje al usuario declinando el requerimiento. Los casos de uso UC7 (por usuario) y UC8 (por administrador) implementan el requerimiento F5.

3) *Crear Canal* – Un usuario envía un mensaje de requerimiento al administrador por medio del canal de administración para crear un nuevo canal, incluyendo el nombre del mismo. En respuesta, el administrador asigna un identificador al nuevo canal, agrega el canal al repositorio central, de ser satisfactorio y contesta al miembro. Después de recibir una respuesta positiva, el usuario crea un nuevo canal y actualiza la base de datos local con información del mismo. De lo contario, no toma acción. Los casos de uso UC7 (por usuario) y UC8 (por administrador) implementan el requerimiento F9.

4) *Destruir Canal* – Un usuario envía un mensaje de requerimiento al administrador para destruir un canal existente. En respuesta, si el canal no tiene suscriptores, el administrador remueve todos los datos relacionados al canal del repositorio central y contesta al usuario. Al recibir una respuesta satisfactoria, el usuario borra todos los datos relacionados al canal de la base de datos local. De lo contrario, no toma acción. Los casos de uso UC7 (por usuario) y UC8 (por administrador) implementan el requerimiento F10.

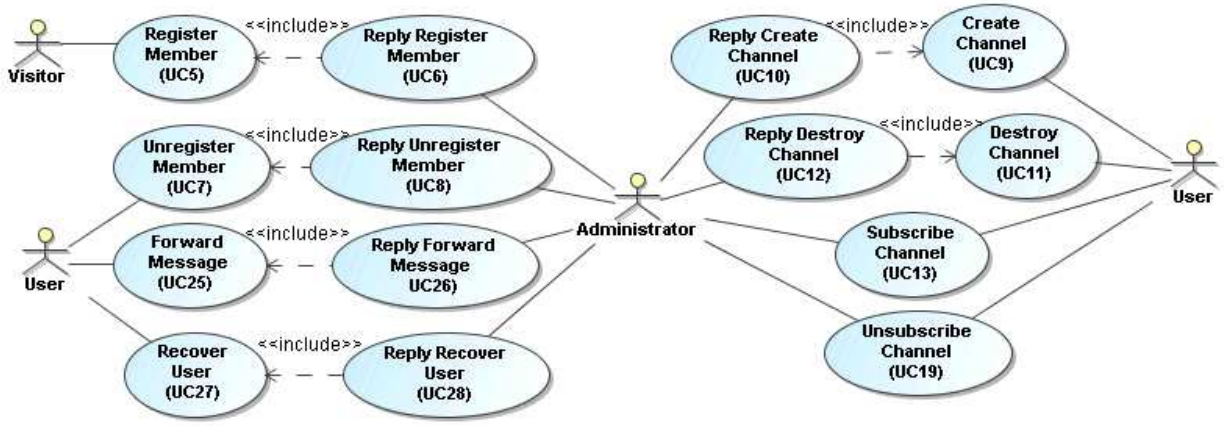
5) *Suscribir a Canal* – La funcionalidad (UC13) está descompuesta en cinco funciones, cada una especificada por un caso de uso más detallado: Requerir Suscripción al Administrador (UC14) – Ejecutado por un usuario, se almacena un requerimiento de suscripción pendiente en la base de datos local, y se prepara para enviar un mensaje al administrador requiriendo la suscripción; Requerir Suscripción al Propietario (UC15) – Ejecutado por el administrador, se almacena un requerimiento de suscripción pendiente en el repositorio central y se reenvía el requerimiento al propietario del canal, si el canal existe, de lo contrario notifica al usuario; Recibir Requerimiento de Suscripción (UC16) – Ejecutado por el propietario del canal, agrega al usuario como un nuevo suscriptor en la base de datos local y contesta al administrador; Responder Suscripción al Suscriptor (UC17) – ejecutado por el administrador, actualiza el estado del requerimiento en la base de datos central y contesta al usuario; y Recibir Respuesta a Suscripción (UC18) – Ejecutado por el usuario, si la respuesta es positiva, actualiza la lista de canales suscriptos en la base de datos local. Los casos de uso UC13 a UC18 implementan el requerimiento F11.

6) *Des-Suscribir de Canal* – Similar a UC13, este caso de uso (UC19) está descompuesto en cinco funciones, cada una especificada por un caso de uso más detallado: Requerir Des-suscripción al Administrador (UC20), Requerir Des-suscripción al Propietario (UC21), Recibir Requerimiento de Des-suscripción (UC22), Responder Des-Suscripción al Suscriptor (UC23), y Recibir Respuesta a Des-suscripción (UC24). UC20 a UC23 son similares a los casos de uso correspondientes para suscripción, con los miembros actualizando sus bases de datos y reenviando mensajes. En el último caso de uso (UC24), si la respuesta del administrador fue satisfactoria, el suscriptor borra el objeto para manejar el canal suscripto y remueve el canal de la base de datos local. Los casos de uso UC19 a UC24 implementan el requerimiento F12.

7) *Reenviar Mensaje* – Un miembro envía un mensaje a otro por medio del administrador usando los canales de administración. Basado en el tipo de mensaje y dirección de destino, el mensaje es reenviado al receptor por medio de canal de administración. Los casos de uso UC25 (por usuarios) y UC26 (por administrador) implementan el requerimiento F20.

8) *Recuperar Usuario* – Este caso de uso permite a los visitantes recuperar información de la base de datos local relacionada a un usuario, como canales poseídos o suscriptos, e instanciar todos los objetos requeridos para reiniciar al usuario luego de reiniciar el sistema. Luego que un visitante solicitó la recuperación al administrador, el administrador comprueba si el requerimiento refiere a un usuario existente y responde al visitante. Si es correcto, visitante recupera la información de usuario de la base de datos local e instancia los objetos. Los casos de uso UC27 (por visitante) y UC28 (por administrador) implementan el requerimiento F6.

Figura 61: Diagrama de Casos de Uso – Servicios Administrativos Básicos



Los Servicios Operacionales permiten a los miembros enviar y recibir mensajes. Este caso de uso de alto nivel está descompuesto en tres: Enviar Mensaje como Propietario, Enviar Mensaje como Suscriptor y Recibir Mensaje, como se describe a continuación:

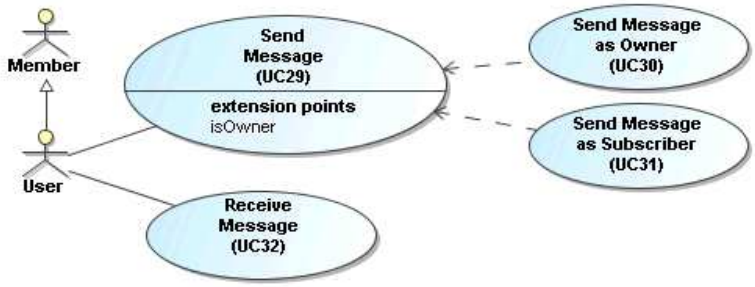
1) *Enviar Mensaje* – El caso de uso para envío de mensajes (UC29) está descompuesto en dos casos de uso:

- *Enviar Mensaje como Propietario* – El caso de uso (UC30) involucra al propietario de un canal en el envío de mensajes a través del mismo. El envío es seguido por el procesamiento de todas las extensiones habilitadas en el canal las cuales requieren procesamiento pre envío, tales como: auditoría, validación, transformación, etc. De ser correcto, el mensaje es enviado a todos los suscriptores del canal excepto al emisor del mensaje. De lo contario, el propietario reenvía el mensaje nuevamente al emisor para notificar la ocurrencia de un error. El caso de uso implementa los requerimientos F13 y F17.

- *Enviar Mensaje como Suscriptor* – El caso de uso involucra a un suscriptor de canal enviando un mensaje al canal (UC31). El envío es seguido por el reenvío del mensaje al propietario del canal, el cual lleva a cabo el procesamiento posterior como se especifica en el caso de uso UC30. El caso de uso implementa los requerimientos F13 y F18.

2) *Recibir Mensaje* – El caso de uso consiste en un miembro recibiendo un mensaje (UC32). La recepción es precedida por el proceso de todas las extensiones habilitados en el canal que requieren procesamiento pos envío, como Criptografía, que requiere la desencriptación de mensajes. De ser satisfactorio, el mensaje es recibido por el miembro y reenviado a una aplicación externa. El caso de uso implementa los requerimientos F14, F15, F16 y F19.

Figura 62: Diagrama de Casos de Uso – Servicios Operacionales



Los Servicios Administrativos Extendidos le permiten a los usuarios administrar extensiones, incluyendo tres extensiones concretas provistas – Auditoría, Validación y Transformación. Los casos de uso detallados son descriptos en la Figura 63 y explicados a continuación:

1) *Entregar Extensión* – Este caso de uso (UC33) le permite al administrador del sistema agregar, borrar y actualizar datos relacionados a una extensión en la base de datos central – tipo de extensión, número de parámetros requeridos, etc. Consiste en que el prototipo le presente un formulario al administrador del sistema quien completa y envía este formulario; el prototipo ejecuta la acción solicitada en el formulario. El caso de uso implementa el requerimiento F21.

2) *Habilitar Extensión de Canal* – El propietario de un canal envía un requerimiento al administrador para habilitar una extensión. Después de recibir el requerimiento, el administrador controla si la extensión está disponible (entregada), actualiza la base de datos central y responde al propietario. Si la respuesta es satisfactoria, el propietario actualiza la base de datos local e instancia el miembro de extensión. Los casos de uso UC34 (por propietario) y UC35 (por administrador) implementan el requerimiento F22.

3) *Configurar Extensión de Canal* – El propietario de un canal solicita al administrador configurar una extensión. Recibido el requerimiento, el administrador valida si la extensión está habilitada en el canal, y responde al miembro. Si la respuesta es satisfactoria, el propietario pasa un parámetro al miembro responsable de implementar la extensión. Los casos de uso UC36 (por propietario) y UC37 (por administrador) implementan el requerimiento F23.

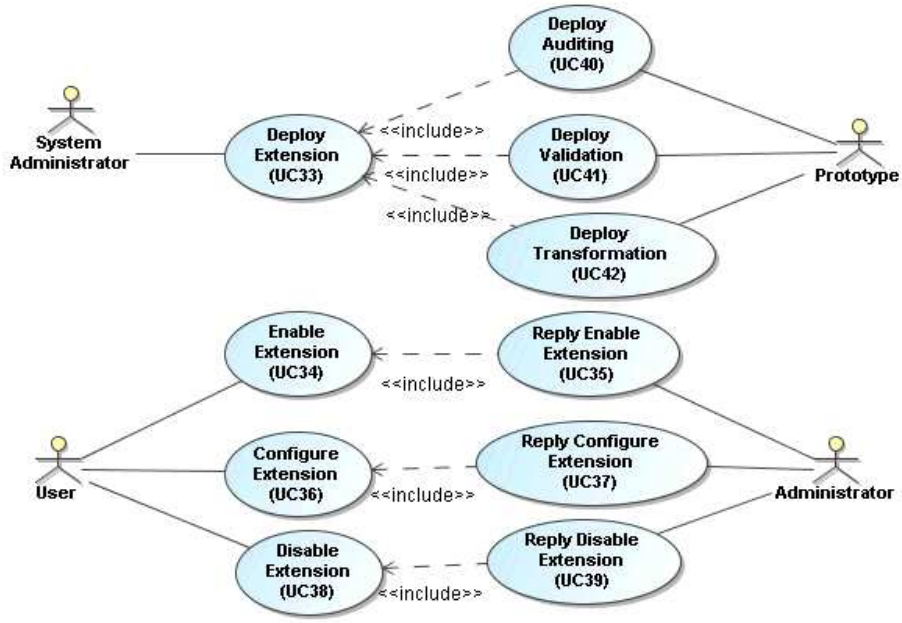
4) *Deshabilitar Extensión de Canal* – Después que el propietario reenvía al administrador un requerimiento para deshabilitar una extensión, el administrador controla si la extensión está habilitada en el canal y si el miembro solicitante es el propietario del canal, y de ser así, actualiza la base de datos central y responde al usuario. De lo contrario, sólo envía al usuario un mensaje de error. De ser satisfactorio, el usuario destruye el objeto usado para procesar la extensión y actualiza la base de datos local. Los casos de uso UC38 (por propietario) y UC39 (por administrador) implementan el requerimiento F24.

5) *Distribuir Extensión de Auditoría* – Este caso de uso (UC40) especifica una extensión a través de la cual cada mensaje en tránsito por un canal auditado puede ser grabado en una base de datos, el registro estará disponible para ser visto y recuperado en el fututo. El caso de uso implementa el requerimiento F25.

6) *Distribuir Extensión de Validación* – Este caso de uso (UC41) especifica una extensión a través de la cual cada mensaje en tránsito por un canal validado es validado de acuerdo a un formato predefinido especificado por el parámetro de la extensión (usualmente un esquema XML). De ser válido, el mensaje es enviado a todos los suscriptores del canal. Si no lo es, es devuelto al emisor notificando el error de validación. Este caso de uso implementa el requerimiento F26.

7) *Distribuir Extensión de Transformación* – Este caso de uso (UC42) especifica una extensión a través de la cual cada mensaje en tránsito por un canal de transformación es transformado de un formato a otro, aplicando un conjunto de reglas de transformación especificadas por el parámetro de la extensión (usualmente una plantilla XSLT). Si la transformación no es satisfactoria, un mensaje de error es enviado al emisor. El caso de uso implementa el requerimiento F27.

Figura 63: Diagrama de Casos de Uso – Servicios Administrativos Extendidos



Los Servicios de Configuración aseguran la existencia de estructuras básicas de mensajería, como los miembros administrador y visitante, y canales en el estado inicial del prototipo. Los casos de uso son descriptos en la Figura 64 y explicados a continuación:

1) *Suministrar Miembro Visitante* – Este caso de uso (UC43) asegura la existencia del tipo de miembro visitante, permitiendo la instanciación de tantos objetos visitante como sean necesarios. Un objeto visitante es instanciado cada vez que una entidad solicita registrar un nuevo usuario o reiniciar uno existente. Una vez enviado el requerimiento al administrador y recibida la respuesta, el objeto cumple su tarea y es destruido. El caso de uso implementa el requerimiento F1.

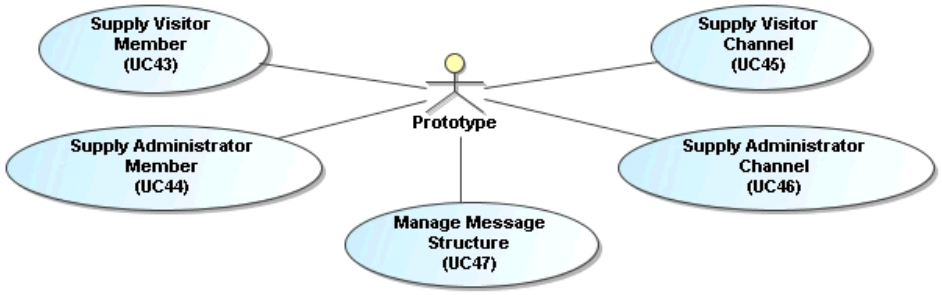
2) *Suministrar Miembro Administrador* – Este caso de uso (UC44) asegura la existencia del miembro administrador. Con exactamente un miembro administrador siempre disponible en el sistema, los datos del administrador deben estar disponibles en el archivo de configuración del prototipo. El caso de uso implementa el requerimiento F3.

3) *Suministrar Canal Visitante* – Este caso de uso (UC45) requiere en todo momento la presencia de un canal de comunicación entre cada miembro visitante y el administrador. Implementa el requerimiento F8.

4) *Suministrar Canal de Administración* – Este caso de uso (UC46) requiere en todo momento la presencia de un canal que comunique los miembros usuario con el administrador. Al recibir un requerimiento de registración de un nuevo usuario, el administrador crea el canal de administración del usuario, se designa a sí mismo como propietario del canal, y al usuario como el único suscriptor del canal. Como resultado, hay tantos canales de administración como usuarios registrados en el sistema. El caso de uso implementa el requerimiento F7.

5) *Manejar Estructura de Mensaje* – Este caso de uso (UC47) permite manejar la estructura de mensajes, tales como: construir una cabecera de mensaje a partir de la información de identificador, emisor, tipo y canal; construir un cuerpo de mensaje a partir del contenido, e incluir los adjuntos; recuperar la información de la cabecera para enviar mensajes a los receptores apropiados; y hacer que el contenido y adjuntos estén disponibles para entidades externas. El caso de uso implementa los requerimientos F13 a F16.

Figura 64: Diagrama de Casos de Uso – Servicios de Configuración

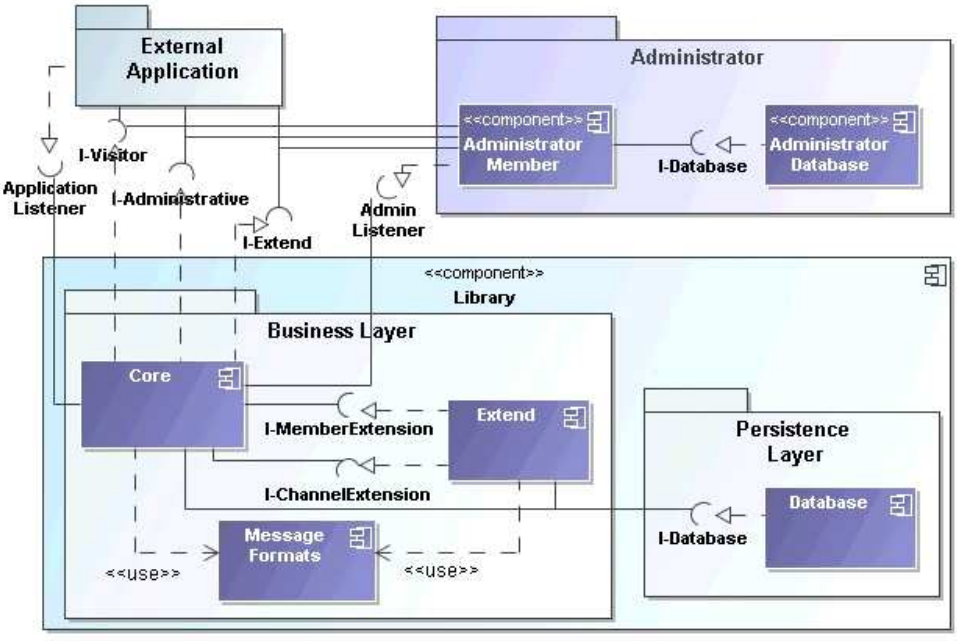


5.3 Diseño

Esta sección presenta vistas estáticas y dinámicas de la arquitectura del prototipo G-EEG. Las primeras definen los componentes principales y sus relaciones. Las segundas definen cómo interactúan los componentes intercambiando mensajes. Asimismo, se presenta el diseño detallado de componentes de la arquitectura en términos de clase y sus asociaciones, e interacciones entre instancias de clases para proveer interacción de componentes como se recomienda en la vista dinámica de la arquitectura.

La vista estática de la arquitectura comprende `Capa de Negocios`, `Capa de Persistencia` y `Administrador`, todas interactuando con `Aplicaciones Externas`. La vista es descripta en la Figura 65 y explicada a continuación:

Figura 65: Arquitectura del Prototipo G-EEG – Vista Estática



- `Capa de Negocios` – Incluye tres componentes. (1) `Core` implementa servicios de mensajería básicos llevados a cabo con diferentes tipos de miembros o canales, exceptuando las respuestas del `Administrador`. Ofrece tres interfaces: `I-Visitor`, `I-Administrative` and `I-Extend`, todas usadas por las aplicaciones externas (`External Application`) y el administrador (`Administrator`). Adicionalmente, utiliza los listeners implementados por aplicaciones externas (`ApplicationListener`) y la aplicación que implementa el administrador (`AdminListener`), también utiliza la interfaces provista por el manejador de la base de datos (`I-Database`) y el repositorio de formato de mensajes (`Message Formats`). (2) `Extend` implementa los servicios para habilitar, configurar y deshabilitar extensiones, ofrecidos por dos interfaces (`I-MemberExtension` y `I-`

Channel Extension) usadas por Core. Como Core, utiliza la interface de la base de datos I-Database y el repositorio de formato de mensajes. (3) Message Formats define la estructura de varios tipos de mensajes.

- Capa de Persistencia – Maneja la persistencia de datos por medio del componente Database. El componente incluye una base de datos local y ofrece sus servicios a través de la interface I-Database, usadas por Core y Extend.

- Administrador – El paquete incluye dos componentes: (1) Administator Member implementa el comportamiento del miembro administrador, esencialmente para responder a los requerimientos de los miembros; y (2) Administrator Database maneja el repositorio central que almacena toda la información relacionada a los miembros existentes, canales y extensiones. Ofrece la misma interface que Database (I-Database), que es usada por Core.

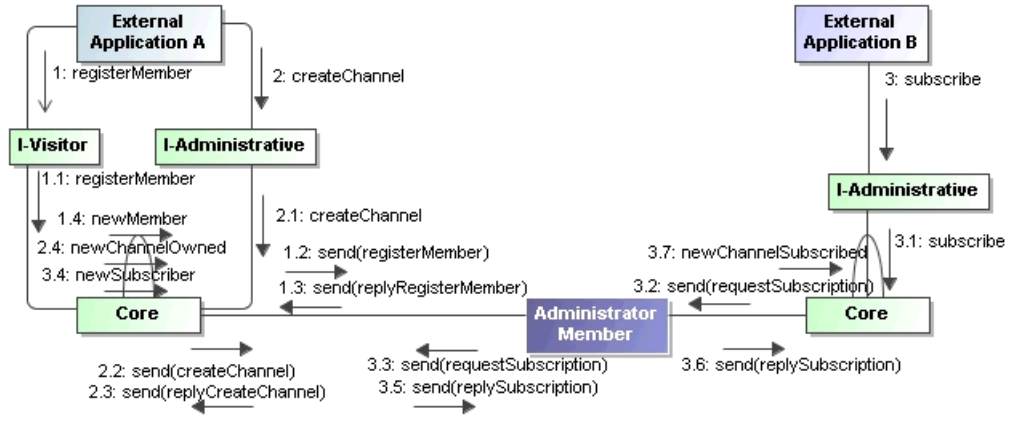
Una vista dinámica de la arquitectura se presenta en la Figura 66. El siguiente intercambio de mensajes ilustra los tres escenarios principales de G-EEG – registración de miembros, creación de canales, y suscripción de miembros a canales, como sigue:

1) *Registración de Miembro* – El escenario es disparado por una aplicación externa – External Application A, requiriendo registrar un miembro a través de I-Visitor – 1:registerMember. Recibido por el componente Core – 1.1:registerMember, el requerimiento es reenviado al miembro administrador – Administrator Member – 1.2:send(registerMember) el cual asigna un identificador al nuevo miembro y envía la respuesta – 1.3:send(replyRegisterMember). Al recibir la respuesta satisfactoria del miembro administrador, un nuevo miembro es instanciado por el componente Core – 1.4:newMember.

2) *Creación de Canal* – Después de registrar un miembro, External Application A solicita a través de la interface I-Administrative la creación de un canal – 2:createChannel. Recibido por el componente Core – 2.1:createChannel, el requerimiento es reenviado al miembro administrador – 2.2:send(createChannel) el cual asigna un nuevo identificador al canal y envía una respuesta – 2.3:send(replyCreateChannel). Al recibir el mensaje, el nuevo canal es creado – 2.4:newChannelOwned.

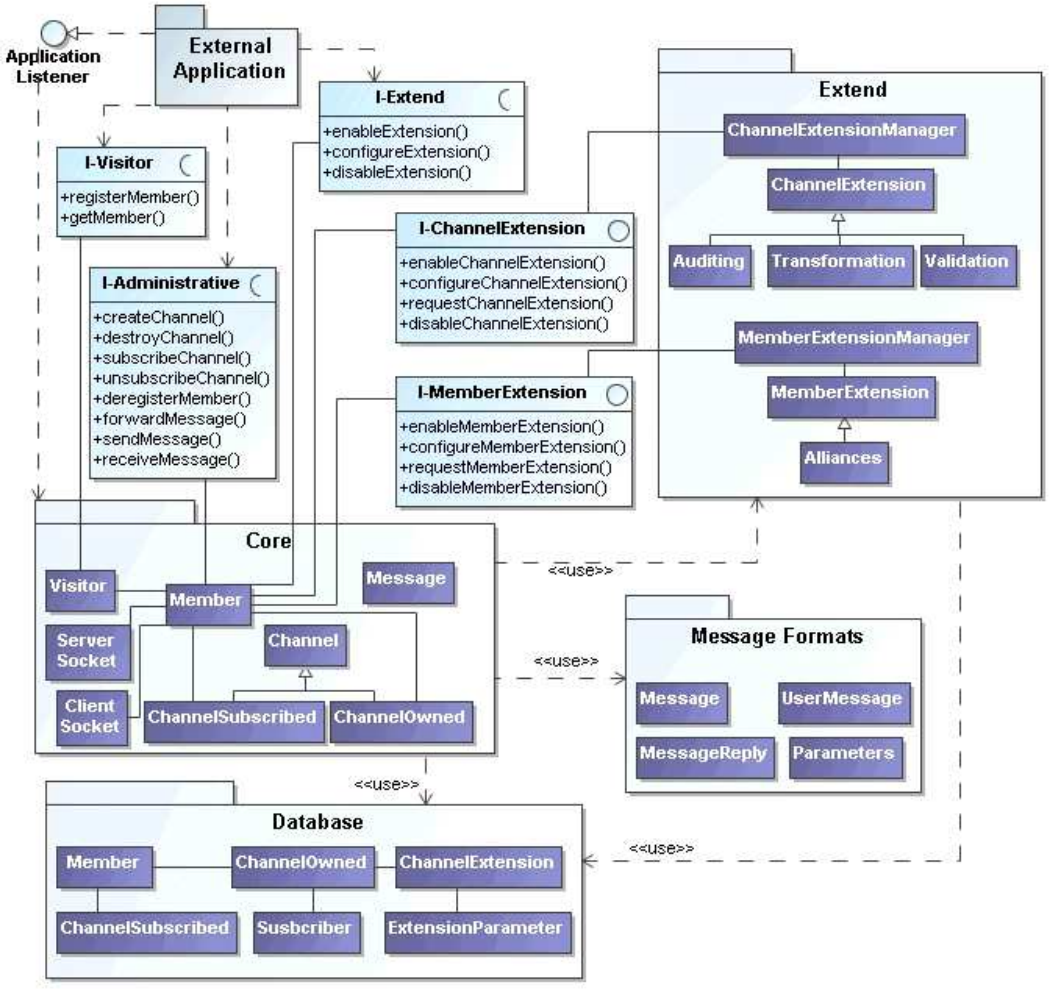
3) *Suscribir Miembro a Canal* – Después de registrado el miembro y creado el canal, External Application B solicita suscripción al canal por medio de I-Administrative – 3:subscribe. Una vez recibido por Core – 3.1:subscribe, el requerimiento es reenviado al miembro administrador – 3.2:send(request Subscription)el cual a su vez, lo reenvía al propietario del canal – 3.3:send(requestSubscription), un componente de Core. Core, por medio del propietario, suscribe al miembro al canal – 3.4:newSubscriber, y responde al miembro administrador – 3.5:send(replySubscription) el cual lo reenvía al componente que solicitó la suscripción – 3.6:send(replySubscription). Una vez recibida la respuesta satisfactoria, se crea un nuevo objeto para manejar el canal suscripto – 3.7:newChannelSubscribed.

Figura 66: Arquitectura del Prototipo G-EEG – Vista Dinámica



La Figura 67 a continuación representa el diseño detallado del prototipo G-EEG con la estructura completa propuesta para los principales componentes de la arquitectura – `Core`, `Extend`, `Database` y `Message Formats` en términos of clases and asociaciones.

Figura 67: Diagrama de Clases de Diseño



Los cuatro componentes ofrecen en conjunto una variedad de servicios de mensajería para aplicaciones externas – `External Application` – servicios de visitante para registrar y recuperar miembros (`I-Visitor`); servicios administrativos para crear y destruir canales, suscribir y des-suscribir miembros a/de canales, des-registrar miembros y reenviar mensajes (`I-Administrative`); servicios básicos para enviar y recibir mensajes (`I-SendReceive`); y servicios de extensión para habilitar, configurar y deshabilitar extensiones basadas en canales y miembros (`I-ChannelExtension` e `I-MemberExtension`).

`Core` utiliza la interface `ApplicationListener` para pasar los mensajes recibidos y los resultados de los servicios a las aplicación externas. Implementado por `External Application`, una instancia de `ApplicationListener` es creada cada vez que la aplicación solicita registrar o recuperar un miembro. `Core` contiene las siguientes clases:

1) `Visitor` – Provee la implementación de servicios ofrecidos por la interface `I-Visitor`, con atributos para almacenar el nombre, identificador y dirección física de un miembro.
2) `Member` – Adicionalmente a los atributos previos, incluye tablas con datos acerca de canales suscriptos y poseídos por un miembro, e implementa los servicios ofrecidos por la interfaces `I-Administrative` e `I-SendReceive`.
3) `Channel` – Una clase abstracta, define las relaciones entre miembros y canales usados por ellos, e incluye las signaturas para las operaciones de enviar y recibir mensajes.

4) `ChannelOwned` – Una sub-clase concreta de `Channel`, contiene atributos para almacenar el identificador y la dirección física del propietario y todos los suscriptores de un canal. Implementa métodos para enviar y recibir mensajes.
5) `ChannelSubscribed` – Una sub-clase concreta de `Channel`, contiene atributos para almacenar el identificador y la dirección física del propietario del canal.
6) `Message` – La clase provee operaciones para obtener elementos individuales de un mensaje XML.
7) `ClientSocket` – Implementa un socket cliente para enviar mensajes en nombre de un miembro.
8) `ServerSocket` – Implementa un socket servidor, verifica continuamente si un miembro recibe algún mensaje.

Como el prototipo sólo implementa extensiones orientadas a canal, `Extend` incluye las siguientes clases:

1) `ChannelExtensionManager` – Contiene un atributo que almacena todas las extensiones orientadas a canal habilitadas en un canal dado. La clase implementa métodos para habilitar y deshabilitar extensiones sobre canales y para procesar los mensajes de acuerdo a todas las extensiones habilitadas.
2) `ChannelExtension` – Una clase abstracta que define las operaciones que debe implementar cada extensión concreta orientada a canal como `processMessage` y `configureExtension`.
3) Adicionalmente, `Extend` incluye tres subclases concretas de `ChannelExtension` - `Auditing`, `Validation` y `Transformation` que implementan las extensiones de auditoría, validación y transformación, respectivamente.

El componente `Message Formats` mantiene definiciones XML para todos los tipos de mensajes y archivos de configuración. Incluye cuatro clases: (1) `Message` define y procesa la estructura de todos los mensajes interpretados por G-EEG; (2) `MessageReply` define la estructura de todos los mensajes producidos por G-EEG para enviar a las aplicaciones externas en respuesta a requerimientos de servicio; (3) `UserMessage` – define el cuerpo de todos los mensajes generados por las aplicaciones externas a ser transferidos por medio de G-EEG, y (4) `Parámetros` – define el archivo de configuración del prototipo.

El componente `Database` define clases para el mapear objetos a tablas de una base de datos relacional para almacenar datos en la base de datos. Este paquete contiene una clase por cada tabla definida en la base de datos: (1) `Member`, (2) `ChannelOwned`, (3) `ChannelSubscribed`, (4) `Subscriber`, (5) `ChannelExtension` y (6) `ExtensionParameter`.

Diagramas detallados de secuencia que describen el comportamiento del sistema para crear canales, enviar y recibir mensajes y habilitar extensiones de canal, son presentados y explicados en el Apéndice C.

5.4 Implementación

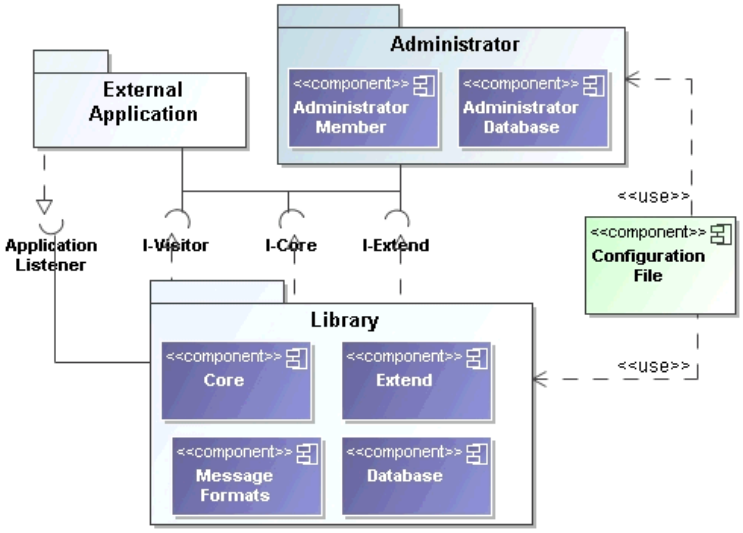
El prototipo G-EEG ha sido implementado en Java usando una familia de estándares abiertos y tecnologías de código abierto. Los mensajes son escritos usando XML [W3C08][Ray01] y manipulados desde programas Java usando XMLBeans [Apa09]. Se usa MySQL [Sun09][Dub06] como motor de base de datos e Hibernate [Hib06] le permite a una clase Java manipular tablas de una base de datos relacional, como MySQL. Las extensiones de validación se basan en el lenguaje XML Schema [W3C04c] y las extensiones de transformación aplican el lenguaje eXtensible Stylesheet Language Transformation (XSLT) [W3C99a].

El diagrama de implementación para el prototipo es descripto en la Figura 68. Los componentes principales de la implementación son: `Core`, `Extend`, `Message Formats`, `Database` y `Administrator`. Detalles técnicos de estos componentes, como las clases contenidas e interfaces usadas e implementadas, fueron provistas en la Sección 5.3.

5.5 Entrega

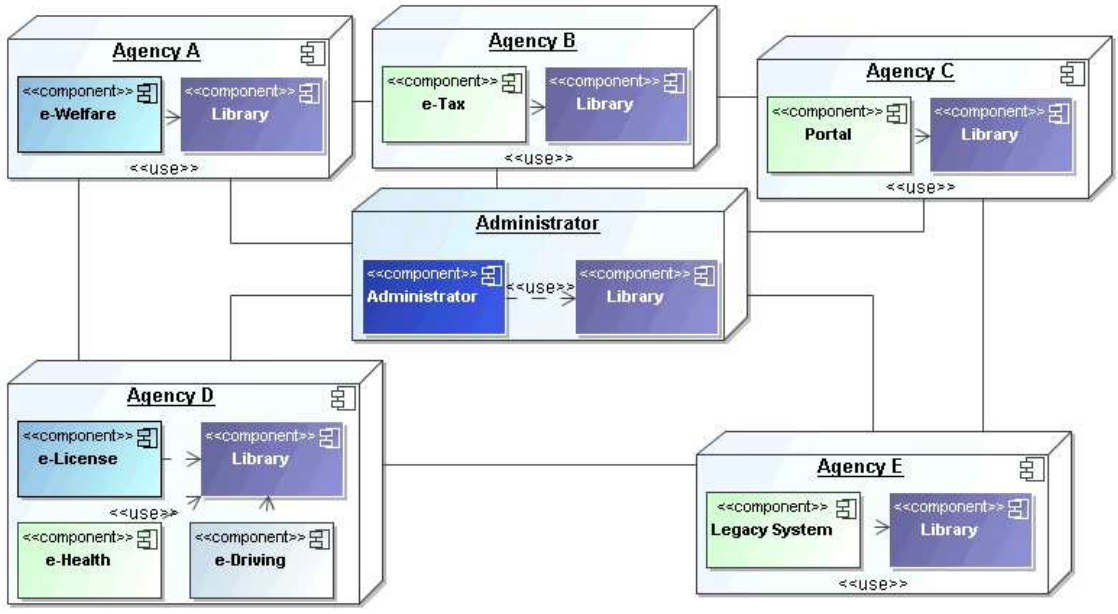
El prototipo G-EEG comprende tres componentes de entrega: (1) `Library` – un conjunto de APIs usadas por las aplicaciones externas – `External Application`, para invocar servicios de mensajería; (2) `Administrator` – la aplicación que implementa el comportamiento del miembro administrador; y (3) `Configuration File` – un archivo XML con parámetros que requiere personalización cuando se instala el prototipo en un computador. Los requerimientos de instalación para cada uno de estos componentes se explican a continuación.

Figura 68: Diagrama de Implementación



Una copia del componente `Library` debe ser entregada en cada servidor que aloje una aplicación que requiera ejecutar servicios de mensajería. Para asegurar la disponibilidad del miembro administrador, `Administrator` debe ser implementado en un servidor non-stop, junto con una copia del componente `Library`. El archivo de configuración – `Configuration File`, debe ser entregado en cada servidor que aloje el componente `Library`, ubicado en la carpeta `G-EEG` en el directorio raíz del usuario.

Figura 69: Diagrama de Entrega



El archivo `Configuration File` debe existir en cada nodo que aloje a un miembro de G-EEG, incluyendo datos acerca del miembro administrador, el miembro visitante instanciado en el servidor local, y algunos parámetros que restringen al prototipo. Cuatro parámetros debe ser definidos para los miembros administrador y visitante: identificador, nombre, URL del servidor que aloja la aplicación `Administrator` y el número de puerto usado por el socket del servidor que recibe mensajes del miembro administrador. Adicionalmente, los siguientes parámetros deben ser configurados: el máximo número de canales que puede poseer cada miembro, el máximo número de canales que puede suscribir cada miembro, el número máximo de suscriptores permitidos para cada canal, el número máximo de extensiones que pueden ser habilitadas en un canal dado, y el tamaño máximo de mensaje. Adicionalmente, el archivo

debe también definir un rango de números de puertos reservados para G-EEG en el nodo corriente, con puertos dentro de este rango asignados unívocamente a cada miembro (más precisamente, el socket del servidor usado por este miembro) que reside en el nodo. Finalmente, dos valores iniciales deben ser definidos para identificar miembros y canales.

La Figura 69 ilustra un escenario posible para la entrega el prototipo. En este escenario, un nodo aloja el miembro administrador (`Administrador`) y varios servidores de agencia alojan las aplicaciones de software que se comunican por medio de G-EEG: `Agency A` aloja la aplicación `e-Welfare`; `Agency B` aloja la aplicación `e-Tax`; `Agency C` aloja la aplicación `Portal`; `Agency D` aloja las aplicaciones `e-License`, `e-Health` e `e-Driving`; y `Agency E` aloja un sistema legado `- Legacy System`. Todos los servidores deben tener instalada una copia del componente `Library`, aún `Agency D`, donde las tres aplicaciones deben registrar su propios miembros G-EEG para poder usar los servicios de mensajería.

Capítulo 6

Evaluación de Mensajería Programable

Luego de presentar los fundamentos (Capítulo 4) y la implementación (Capítulo 5) de Government-Enterprise Ecosystem Gateway – G-EEG, este capítulo provee una evaluación de G-EEG, como una concreta realización de Mensajería Programable.

La evaluación se lleva a cabo en cuatro partes. Primero, revisamos el planteo del problema formulado en la Sección 1.4, y explicamos cómo G-EEG satisface todos los requerimientos de este enunciado. Segundo, revisamos los diez desafíos tecnológicos de Gobierno Integrado introducidos en la Sección 2.5.2 y explicamos cómo G-EEG puede ayudar a enfrentarlos. Tercero, revisamos el caso de estudio de la Sección 1.3, explicamos los requerimientos de comunicación del proceso de negocios definidos por el caso de estudio, y definimos una solución basada en G-EEG que satisface tales requerimientos. Finalmente, visitamos el marco de evaluación de la Sección 3.1 y extendemos los resultados de la evaluación de las soluciones organizacionales, tecnológicas y fundacionales relacionadas, presentadas en la Sección 3.5, con una evaluación comparativa de G-EEG a través de las tres categorías.

El capítulo está dividido en cuatro secciones, que evalúan a G-EEG con respecto a: planteo del problema (Sección 6.1), desafíos tecnológicos (Sección 6.2), el caso de estudio (Sección 6.3) y trabajos relacionados (Sección 6.4).

6.1 Planteo del Problema

Según el planteo del problema formulado en la Sección 1.4, nuestro objetivo es entregar "una plataforma de comunicación y coordinación, con el modelo y la teoría subyacentes, para facilitar el establecimiento, la operación y la evolución de redes organizativas, capaces de entregar servicios públicos integrados".

Alegamos que G-EEG es una solución válida para este fundamento, basándonos en los siguientes argumentos:

- *Plataforma de Comunicación* – G-EEG soporta comunicaciones asincrónicas entre miembros de una red organizacional, donde cada miembro posee un par de colas de mensajes (entrada y salida) para guardar mensajes entrantes y salientes. Implementando una arquitectura totalmente distribuida y usando bandejas de entradas para intercambiar y procesar mensajes, G-EEG tiene las características de una plataforma de comunicación similar a las soluciones de tipo MOM (Definición 17).

- *Plataforma de Comunicación* – G-EEG soporta la comunicación entre miembros de redes organizativas que participan conjuntamente en procesos de negocios inter-organizacionales, con dos extensiones de mensajería que proveen apoyo específico a tales procesos: (1) Orden – asegura que la secuencia de mensajes intercambiados por miembros durante la colaboración cumple con el patrón de comunicación prescripto por el proceso de negocios subyacentes; y (2) Puntualidad – monitorea si el patrón de mensajes intercambiados entre los miembros cumple con las políticas predefinidas, como por ejemplo políticas para acusar recibo de las solicitudes de servicio, o políticas para responder a las solicitudes de servicio dentro de un tiempo predeterminado. En vista de esto, G-EEG realmente soporta la ejecución y el monitoreo de procesos de negocios llevados a cabo por socios colaborativos, y puede ser considerado una plataforma de coordinación.

- *Modelo y Teoría Subyacente* – Una especificación formal de G-EEG fue presentada en el Capítulo 4, desde operaciones primitivas para manejar estructuras de datos tipo XML, a través de una familia de lenguajes (Familia de Tecnologías XML) con sintaxis y semánticas formales para representar varios tipos de expresiones sobre estas

estructuras, hasta servicios concretos de mensajería ofrecidos por G-EEG a sus miembros. Esta especificación constituye el modelo y la teoría de G-EEG.

- *Establecimiento de Redes Organizacionales* – En su estado inicial, G-EEG contiene dos miembros distinguidos: administrador – para procesar la registración de miembros y otros servicios requeridos por miembros, y visitante – para registrar nuevos miembros, conectados a través de un canal único. Desde este estado, las operaciones de G-EEG para registrar miembros y crear canales permiten arbitrariamente el desarrollo de estructuras de comunicación complejas que relacionan miembros entre sí, conduciendo de este modo al establecimiento de redes organizacionales.

- *Operación de Redes Organizacionales* – Los miembros de G-EEG actúan en nombre de miembros de redes organizacionales. Una vez registrado, un miembro puede solicitar varios servicios provistos por G-EEG – para crear nuevos canales, para suscribirse a canales existentes, para enviar y recibir mensajes a través de canales, etc. Todos los servicios, incluidos los dedicados a crear y modificar las estructuras de comunicación, se llevan a cabo a través del envío y la recepción de mensajes. Con la aplicación uniforme del paradigma de mensajería, G-EEG soporta la operación de redes organizacionales.

- *Evolución de Redes Organizacionales* – Una vez establecidas, las estructuras de comunicación creadas por los miembros de G-EEG, pueden ser utilizadas para intercambiar mensajes, donde un mensaje enviado a un canal por un suscriptor es entregado a todos los otros suscriptores. Sin embargo, algunos tipos de mensajes particulares, cuando son enviados y recibidos por miembros autorizados, pueden provocar un cambio en las estructuras subyacentes. Por ejemplo, el mensaje de registración enviado por el miembro visitante al miembro administrador. De este modo, G-EEG soporta la evolución de redes organizacionales.

El planteo del problema también presenta los siguientes cuatro requerimientos:

R1) *Conectar software y personas trabajando dentro de las organizaciones miembros a través de un ambiente de trabajo en red, para permitir el intercambio, interpretación y procesamiento de información* – G-EEG conecta los nodos de redes organizacionales – software, personas y organizaciones – para facilitar el intercambio de información. Además, sus extensiones, como la Transformación y Validación, permiten a los miembros interpretar y procesar adecuadamente la información.

R2) *Apoyar procesos de negocios inter-organizacionales a través de los cuales las organizaciones miembros puedan de manera conjunta, entregar servicios públicos a clientes* – Como una plataforma de coordinación, G-EEG puede brindar soporte tecnológico a organizaciones miembros para que juntas alcancen importantes objetivos inter-organizacionales, como por ejemplo la entrega de Servicios Integrados a clientes.

R3) *Permitir el monitoreo de la entrega de servicios públicos, los procesos subyacentes y el comportamiento de organizaciones que participan en ellos, en base a políticas y regulaciones prescriptas por el gobierno* – Como se explicó anteriormente, Orden y otras extensiones afines permiten a los miembros de G-EEG cumplir con este requerimiento.

R4) *Permitir la evolución de redes organizacionales a través de varios cambios en su entorno, afectando la membrecía, contratos, regulaciones y servicios provistos a través de tales redes* – Con operaciones para registrar y des-registrar miembros, crear y destruir canales, y suscribir y des-suscribir a miembros a/de canales, G-EEG soporta la evolución de redes organizacionales en cuanto a su membrecía y conectividad. Cambios futuros son soportados a través de varias extensiones entregadas por G-EEG.

Finalmente, el problema requiere que cualquier solución válida ayude a resolver los desafíos tecnológicos (Sección 2.5.2) y a implementar el caso de estudio de servicios de licencias (Sección 1.3), ambos cumplidos por G-EEG como se argumenta en las Secciones 6.2 y 6.3 respectivamente.

6.2 Desafíos Tecnológicos

Las siguiente secciones presentan la evaluación de G-EEG con respecto a los diez desafíos tecnológicos presentados en la Sección 2.5 – Acceso Único (Sección 6.2.1), Procesos Inter-organizacionales (Sección 6.2.2), Monitoreo de Cumplimiento de Políticas (6.2.3), Integración de Aplicaciones (Sección 6.2.4), Interoperabilidad Sintáctica (Sección 6.2.5), Interoperabilidad Semántica (Sección 6.2.6), Subcontratación Flexible (Sección 6.2.7) Ecosistema Dinámico (Sección 6.2.8), Entrega por Múltiples Canales (Sección 6.2.9) y Requerimientos de Dependabilidad (Sección 6.2.10).

En cada sección presentamos: (1) Requerimientos – qué requerimientos deberían ser satisfechos por cualquier solución válida para el desafío; (2) Solución – cómo puede G-EEG enfrentar al desafío; (3) Justificación – por qué la solución propuesta en base a G-EEG es válida; y (4) Ejemplo – una solución concreta en base a G-EEG para el ejemplo de la Sección 2.5.

6.2.1 Acceso Único

Requerimientos – A fin de proveer acceso único a EPS, los siguientes requerimientos técnicos deberían ser cumplidos: (1) reunir información relacionada a servicios – criterio de elegibilidad, formularios de solicitud, etc., de los proveedores de servicios y hacerla disponible a través del acceso único; (2) despachar solicitudes de servicios presentadas a través del acceso único a varios proveedores de servicios; y (3) recabar el estado de las solicitudes de servicios de los proveedores de servicios y hacerlos disponibles cuando los postulantes soliciten el seguimiento de sus solicitudes a través del acceso único.

Solución - Suponga que el acceso único depende de un repositorio central para mantener información relacionada a servicios y de otro para mantener el estado de las solicitudes de servicios presentadas. Bajo esta suposición, una solución de acceso único en base a G-EEG podría ser desarrollada de la siguiente manera. El primer paso es registrar miembros en G-EEG para todas las aplicaciones de software que participan en la entrega de SPE a través del acceso único: (1) el repositorio central R1 de información relacionada a servicios; (2) proveedores de servicios P1, P2, etc. que contribuyan a R1 con información relacionada a servicios; (3) aplicaciones de front-office F01, F02, etc. que apoyen la entrega de EPS a través del acceso único; (4) el repositorio central R2 con el estado de las solicitudes de servicios presentadas y que pueden seguirse a través del acceso único; y (5) aplicaciones de software SA1, SA2, etc. que participen en la producción de servicios solicitados a través del acceso único y que sean capaces de cambiar el estado de las solicitudes de servicios. Además, un miembro `portal` es registrado para el acceso único. El segundo paso consiste en que los miembros creen estructuras de comunicación entre ellos, de la siguiente manera: (1) R1 crea un canal para comunicarse con cada proveedor de servicio – R1-P1,R1-P2, etc. y cada proveedor se suscribe al canal correspondiente – P1 se suscribe a R1-P1, P2 se suscribe a R1-P2, etc.; (2) `portal` crea un canal para comunicarse con cada aplicación de front-office – `portal`-F01, `portal`-F02, etc. y cada aplicación de front-office se suscribe al canal correspondiente – F01 se suscribe a `portal`-F01, F02 se suscribe al `portal`-F02, etc.; y (3) R2 crea un canal para comunicarse con todas las aplicaciones productoras de servicios – R2-SA1, R2-SA2, etc., y cada una de estas aplicaciones se suscribe al canal correspondiente – SA1 se suscribe a R2-SA1, SA2 se suscribe a R2-SA2, etc. Finalmente, si fuera necesario, se podrán crear y configurar extensiones en estos canales. Por ejemplo, R1 y R2 podrían habilitar la Autenticación para los canales que los conectan con aplicaciones proveedoras y productoras de servicios, mientras que `portal` puede habilitar la Validación, Auditoría y Cifrado en todos los canales que posee.

Justificación - Justificamos que la solución basada en G-EEG descripta arriba es válida para enfrentar el desafío de acceso único, demostrando como cumple con los tres requerimientos planteados por este desafío:

1) La solución facilita la comunicación entre proveedores de servicios y el repositorio central de información relacionada a servicios, proveyendo acceso único a tal información. Por ejemplo, cada proveedor de servicios P1 puede especificar criterios de elegibilidad y enviarlos al depósito R1 a través del canal R1-P1. Además, R1 puede habilitar la extensión de Autenticación en todos los canales R1-Pi, asegurando que el repositorio central sólo recibe información relacionada a servicios de aquellos proveedores de servicios que han sido autorizados.
2) La solución facilita la comunicación entre el acceso único (`portal`) y todas las aplicaciones de front-office que soportan la entrega de servicios ofrecidos a través de él. Por ejemplo, si F01 es la aplicación de front-office que soporta la entrega de licencias para negocios de comidas y bebidas, `portal` puede despachar solicitudes para la emisión de tales licencias a F01 a través del canal `portal`-F01. Como este canal sólo conecta las partes

involucradas, ninguna otra entidad recibirá la información intercambiada. En particular, solo F01 recibirá solicitudes para emitir licencias para negocios de comidas y bebidas presentados a través del acceso único, ya que tales solicitudes son enviadas por `portal` a través del canal `portal-F01`, y F01 es el único suscripto a este canal. Adicionalmente, habilitando extensiones en los canales usados para despachar solicitudes a las aplicaciones de front-office se puede asegurar que: las aplicaciones de front-office sólo reciben información válida – Validación; todas las solicitudes presentadas a través del acceso único son registradas – Auditoría; y la información sensitiva del postulante es protegida de posibles escuchas a escondidas – Cifrado.

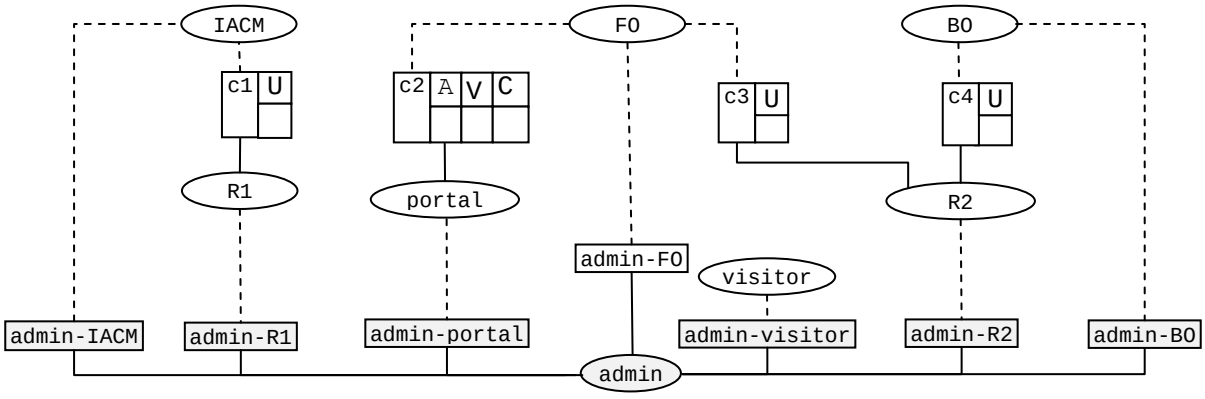
3) La solución facilita la comunicación entre las aplicaciones productoras de servicios y el repositorio central que mantiene información relacionada al estado de las solicitudes de servicio, con las aplicaciones enviando los cambios de estado y el repositorio manteniendo actualizada esta información. Por ejemplo, cuando F01 cambia el estado de una solicitud de servicio, envía un mensaje de notificación al repositorio R2 a través del canal `F01-R2`.

A continuación se presenta una solución basada en G-EEG para el desafío de implementar Acceso Único.

Ejemplo 47: Solución Basada en G-EEG para Ofrecer Servicios a través de Acceso Único

La Figura 70 representa una solución basada en G-EEG que ofrece el servicio de licencias a través del acceso único, tratando la instancia concreta del desafío de acceso único presentado en el Ejemplo 23. Los miembros son: (1) R1 - repositorio central con información acerca del SPE ofrecido a través del acceso único, (2) `IACM` - proveedor del servicio de licencias, (3) F0 - aplicación de front-office que soporta la entrega del servicio de licencias en IACM, (4) B0 - aplicación de back-office que soporta la entrega del servicio de licencias en IACM, (5) R2 - repositorio central con información del estado de las solicitudes, y (6) `portal` - portal de acceso único. El `Visitante` y el `Administrador`, distinguidos miembros de G-EEG, también son representados. Cada miembro tiene su propio canal de administración para poder comunicarse con el Administrador: `admin-R1- admin-IACM, admin-portal, admin-F0, admin-B0, admin-R2` y `admin-visitor`. Los siguientes canales también fueron creados con miembros suscribiéndose a ellos: (1) c1 - creado por R1 y suscripto por IACM, (2) c2 - creado por `portal` y suscripto por F0, (3) c3 - creado por R2 y suscripto por F0, y (4) c4 - creado por R2 y suscripto por B0. Además, la extensión de Autenticación (U) está habilitada en c1, c3 y c4, mientras que Auditoría (A), Validación (V) y Cifrado (C) están habilitadas en c2. En base a esta estructura, IACM puede enviar información relacionada a los servicios a R1 a través de c1, solicitudes y documentos necesarios presentados a través del `portal` de acceso único son enviados a F0 a través de c2, y el estado de las solicitudes es enviado por F0 y B0 a R2 a través de c3 y c4 respectivamente.

Figura 70: Solución basada en G-EEG para Ofrecer Servicios a través de Acceso Único



6.2.2 Procesos Inter-organizacionales

Requerimientos – A fin de soportar la ejecución colaborativa de procesos de negocios inter-organizacionales, identificamos cinco requerimientos técnicos: (1) permitir la comunicación entre las organizaciones asociadas, (2) monitorear que el proceso de negocios subyacentes es ejecutado según su especificación, con cada paso del proceso

ejecutado en el orden correcto y por el socio correcto, (3) determinar el estado actual de la ejecución del proceso, (4) intercambiar información específica sobre el proceso - quejas de clientes, políticas y reglamentaciones, etc. entre socios, y (5) permitir sólo a socios autorizados que se involucren en la comunicaciones relacionadas con el proceso.

Solución – Un enfoque general para construir soluciones basadas en G-EEG para apoyar los procesos inter-organizacionales incluye: (1) registrar un miembro en G-EEG para comunicarse en nombre de cada organización asociada o una aplicación ejecutada por ésta, (2) crear un canal para cada tipo de colaboración requerida por el miembro registrado, y (3) suscribir a todos los miembros que tomen parte en tales colaboraciones al canal correspondiente. Una vez que las estructuras de comunicación existen, los miembros pueden habilitar y configurar las siguientes extensiones: (1) el miembro responsable de iniciar una instancia del proceso de negocios habilita y configura la extensión de Orden; (2) todos los miembros involucrados en el proceso habilitan y configuran la extensión de Seguimiento; y (3) cada propietario de canal habilita y configura la extensión de Autenticación en todos sus canales. Más aún, a fin de garantizar diferentes tipos de procesamientos a posteriori de los mensajes recibidos en un canal, varios tipos de extensiones de Composición de Canal pueden ser habilitadas y configuradas: Vinculación – despachar los mensajes recibidos en un canal a otro canal; Distribución – copiar los mensajes recibidos en un canal a todos los otros canales; Unión – juntar los mensajes recibidos por dos o mas canales y enviarlos por otro canal; Filtrado - decidir si despachar o no los mensajes recibidos en un canal a otro canal, dependiendo del contenido de los mensajes; y Ruteo - despachar mensajes recibidos en un canal a uno de varios canales restantes dependiendo del contenido de los mensajes o del estado del sistema.

Justificación - Justificamos que la solución basada en G-EEG descripta arriba es válida para enfrentar el desafío de procesos inter-organizacionales demostrando como cumple con los cinco requerimientos planteados por este desafío:

1) La solución facilita la comunicación entre todas las organizaciones asociadas que participan en el proceso de negocios, con un miembro registrado por cada socio. Independientemente de la plataforma de TI usada por cada organización asociada para ejecutar los diferentes pasos de proceso, ellas pueden enviar y recibir mensajes a través de sus miembros.
2) La ejecución del proceso puede ser controlada por la extensión de Orden. La extensión controla si los mensajes son recibidos por canales específicos y enviados por miembros específicos - controlando cuales socios ejecutan los diferentes pasos del proceso, y si el intercambio de mensajes cumple con los requerimientos de orden.
3) La extensión de Seguimiento puede determinar la ubicación de los mensajes en tránsito entre miembros, y así determinar el estado de la ejecución del proceso global.
4) Al habilitar la extensión de Alianza, todos los miembros registrados que representan a organizaciones asociadas pueden intercambiar mensajes relacionados al proceso a través de un canal dedicado.
5) La Autenticación puede ser habilitada en todos los canales a los fines de garantizar que sólo los miembros autorizados puedan generar mensajes.

A continuación, una solución basada en G-EEG para el desafío de implementar Procesos Inter-Organizacionales.

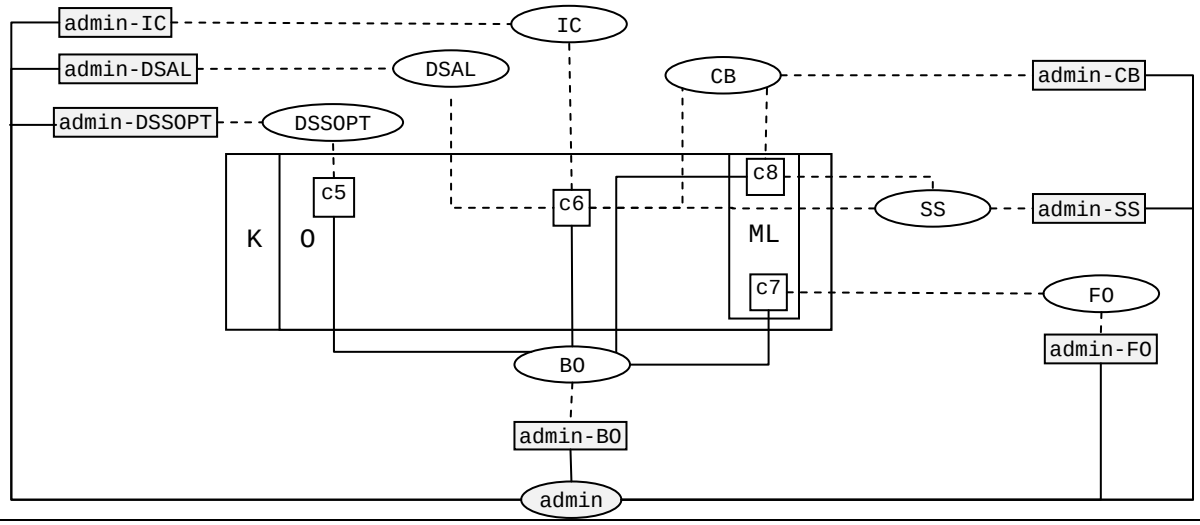
Ejemplo 48: Solución Basada en G-EEG para Soportar Procesos Inter-Organizacionales

La Figura 71 presenta una solución basada en G-EEG para la entrega colaborativa del servicio de licencias del Ejemplo 24. Siete miembros están registrados en la solución, actuando en nombre de los siguientes socios o aplicaciones: (1) `BO` - sistema de back-office en IACM responsable de solicitar colaboraciones a otros socios, tomar decisiones acerca de solicitudes basadas en la información recibida de los socios, y emitir licencias; (2) `DSSOPT` - Lands, Public Works and Transport Bureau responsable de proveer opiniones técnicas acerca de planos de construcción e infraestructura de los establecimientos; (3) `DSAL` - Labour and Employment Bureau responsable de emitir opiniones acerca de la seguridad del ambiente de trabajo y las condiciones sanitarias de los establecimientos; (4) `IC` - Cultural Affairs Bureau responsable de emitir opiniones acerca de la apariencia de los establecimientos ubicados en las zonas de preservación de patrimonio cultural; (5) `CB` - Fire Services Bureau responsable de controlar las medidas de prevención de incendios implementadas en los establecimientos; (6) `SS` - Health Bureau responsable de inspeccionar las condiciones sanitarias de los establecimientos; y (7) `FO` - sistema de front-office en IACM responsable de las interacciones con los postulantes. Cada miembro tiene su propio canal de administración: `admin-BO`, `admin-DSSOPT`, `admin-DSAL`, `admin-IC`, `admin-CB`, `admin-SS` y `admin-FO`. Cuatro canales son creados por `BO` para solicitar colaboraciones a los socios: `c5`, `c6`, `c7` y `c8`. El canal `c5`, suscripto por `DSSOPT`, es usado por `BO` para enviar planos de construcción a `DSSOPT` para solicitar opiniones técnicas y para

recibir las opiniones de DSSOPT. El canal c6, suscripto por DSAL, IC, CB y SS, es usado por BO para solicitar opiniones de estas agencias, recibir respuestas de DSAL e IC, y recibir propuestas de horarios para llevar a cabo visitas de inspección por parte de CB y SS. El canal c7, suscripto por FO, es usado por BO para solicitar que se notifique a los postulantes sobre las visitas de inspección que serán llevadas a cabo por CB y SS. Una vez que FO recibe la aprobación del postulante, le responde a BO a través del canal c7. El canal c8 es suscripto por CB y SS y es usado por BO para informar a estas agencias acerca de citas confirmadas con los postulantes para la visita a los establecimientos. Finalmente, luego que CB y SS completan sus inspecciones, le informan a BO acerca de los resultados a través del canal c8.

La solución habilita tres extensiones verticales: Orden (O), Seguimiento (K) y Composición de Canales (ML). Orden, habilitada sobre los canales c5,c6, c7 y c8 controla el orden y el emisor de los mensajes intercambiados sobre esos canales. Seguimiento, habilitada sobre los mismos canales que Orden, permite saber cual fue el último mensaje enviado a través de esos canales para una instancia de proceso dada. Finalmente, Composición de Canales es usada para unir c7 y c8, de este modo, todos los mensajes enviados por FO a través de c7 con la confirmación de las citas para la inspección de los comercios son recibidos por CB y SS a través de c8.

Figura 71: Solución Basada en G-EEG para Soportar Procesos Inter-Organizacionales



6.2.3 Monitoreo de Cumplimiento de Políticas

Requerimientos: Dos problemas técnicos se identifican para monitorear la conformidad de políticas para la producción y entrega colaborativa de SPE: (1) describir políticas relacionadas a servicios de tal manera que la conformidad de políticas puedan ser verificadas automáticamente por software y (2) monitorear en tiempo de ejecución la conformidad a la política en base a las descripciones de la política de servicio.

Solución: A fin de construir una solución basada en G-EEG que pueda cumplir con ambos requerimientos, las estructuras de comunicación que permiten el intercambio de mensajes entre las organizaciones miembro deben estar creadas, de acuerdo a lo descrito en la Sección 6.2.2. Dos pasos más de desarrollo son requeridos para que esta solución pueda monitorear el cumplimiento de políticas: (1) registrar un miembro dedicado para que actúe como Supervisor de Política para un servicio determinado, y (2) habilitar y configurar la extensión Puntualidad en todos los canales usados para la producción y entrega del servicio por parte del Supervisor de Política.

Justificación: Justificamos que la solución basada en G-EEG descripta arriba es válida para enfrentar el desafío de monitorear el cumplimiento de políticas, demostrando como cumple con los dos requerimientos planteados por este desafío:

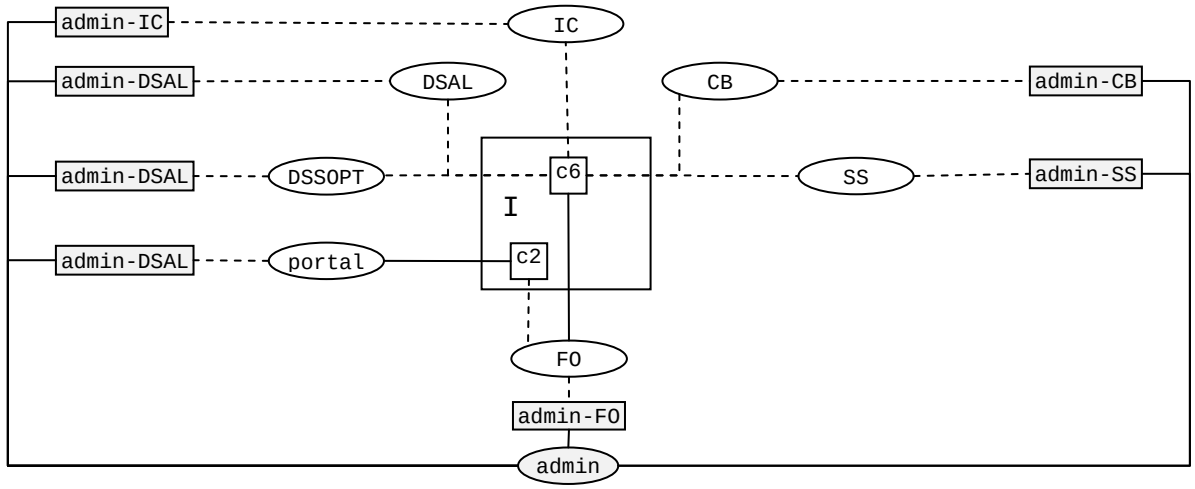
1) La extensión de Puntualidad provee un mecanismo para describir políticas de tal modo que la conformidad a las mismas pueda ser decidida automáticamente. En particular, el ejemplo de lenguaje definido en el Capítulo 4 expresa requerimientos para controlar tiempo en las comunicaciones entre organizaciones asociadas. Tal lenguaje puede ser extendido para cubrir otros tipos de políticas.
2) La extensión de Puntualidad permite controlar la conformidad con respecto a la política especificada y los mensajes enviados a través de un grupo de canales. Envía notificaciones al Supervisores de Política, si la política es violada.

A continuación se presenta una solución basada en G-EEG para el desafío de Monitorear el Cumplimiento de Políticas.

Ejemplo 49: Solución Basada en G-EEG para Monitorear la Conformidad de Políticas

Según el Ejemplo 25, cada solicitud de colaboración emitida por F0 a DSAL, DSSOPT, CB, IC y SS debería ser enviada en menos de 24 horas después de presentada la solicitud. La Figura 72 muestra una solución basada en G-EEG que verifica el cumplimiento con respecto a esta política. Como se explicó en el Ejemplo 47, `portal` crea el canal `c2`, suscripto por F0, para despachar a F0 las solicitudes de servicio presentadas a través del acceso único. Una vez recibidas las solicitudes por F0, son despachadas a las agencias colaboradoras a través de `c6`. A fin de controlar la política, F0, actuando como Supervisor de Política, habilita la extensión de Puntualidad sobre el canal `c2` y `c6` y configura la extensión para que la política sea controlada. En caso que la política no se cumpla, un mensaje de notificación será enviado a F0 a través de su canal de administración.

Figura 72: Solución Basada en G-EEG para Monitorear la Conformidad de Políticas



6.2.4 Integración de Aplicaciones

Requerimientos: La entrega de servicios integrados requiere integrar aplicaciones de software heterogéneas ejecutadas por varios socios organizacionales. El principal problema técnico es conectar sistemas legados con sistemas construidos con nuevas tecnologías. Un enfoque común, particularmente para las tecnologías de mensajería, se basa en adaptadores. Construidos para productos de mensajería específicos [HW04], los adaptadores son los componentes que permiten a las aplicaciones enviar y recibir mensajes a través de las APIs de mensajería. Por ejemplo, Envoy MQ [Env09] es un adaptador que provee conectividad entre MSMQ y aplicaciones legadas. Dos adaptadores también fueron desarrollados a partir de esta tesis [EDJ09c] para permitir el envío y recepción de mensajes a través del prototipo G-EEG: por sistemas legados construidos en lenguaje Delphi y por sistemas legados capaces de invocar desde el entorno del lenguaje de programación un comando del sistema operativo.

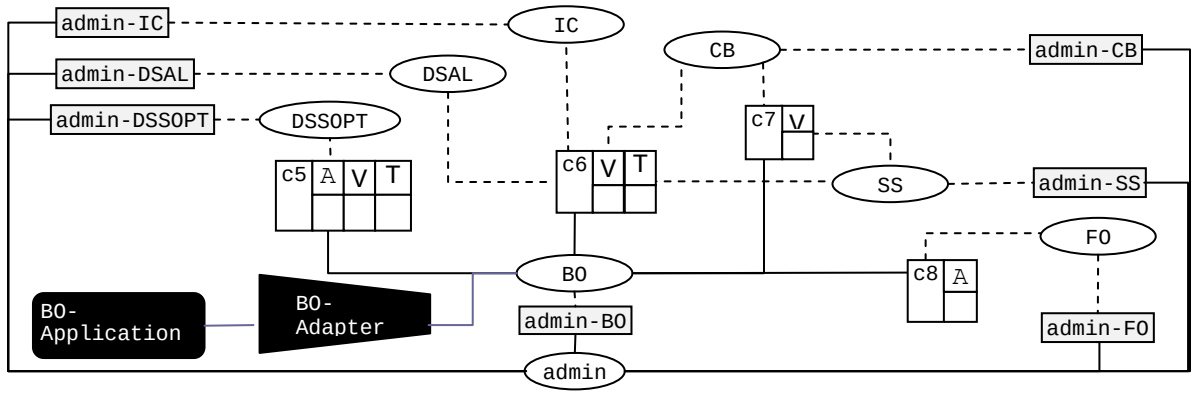
Solución: A fin de construir una solución basada en G-EEG que pueda superar el desafío de integración, se necesitan cuatro pasos. Primero, un miembro es registrado para que actúe en nombre del sistema legado. Segundo, el miembro registrado crea canales o se suscribe a canales existentes para intercambiar mensajes con otras aplicaciones. Tercero, varias extensiones son habilitadas y configuradas en los canales usados para intercambiar mensajes con las aplicaciones legadas, tales como Auditoría, Validación y Transformación. Finalmente, el miembro registrado puede enviar y recibir mensajes directamente a través de G-EEG, o de manera indirecta invocando los servicios provistos por el adaptador, asumiendo que tal adaptador existe y es accesible.

Justificación: Justificamos que la solución basada en G-EEG descripta arriba es válida para enfrentar el desafío de integración de aplicaciones, demostrando como cumple los requerimientos descriptos arriba:

1) Existen adaptadores para sistemas legados [EDJ09c]. Tales adaptadores permiten a estos sistemas, escritos en cualquier lenguaje que provea acceso p. ej. a la línea de comando, a enviar y recibir mensajes a través de G-EEG.
2) Los servicios de G-EEG usados para crear estructuras de comunicación – registrar miembros, suscribir a canales, etc. puede ser invocados por el sistema legado a través del adaptador, sin necesidad de cambiar el sistema legado.
3) La extensión de Auditoría puede mantener un registro histórico de mensajes enviados o recibidos por el sistema legado, si el sistema legado no puede ser modificado para mantener sus propios registros.
4) La extensión de Validación puede garantizar que el sistema legado reciba sólo información válida, ya que puede no ser posible modificar el código fuente del sistema legado para introducir excepciones para administrar fallas.
5) La extensión de Transformación puede garantizar que el sistema legado reciba información en el formato que éste espera, o envíe mensajes en el formato que otros esperan.
6) Servicios para habilitar y configurar las extensiones propuestas pueden ser solicitadas a través de la línea de comando, sin necesidad de modificar los sistemas legados.

A continuación, la Figura 73 presenta una solución basada en G-EEG para el desafío de Integrar Aplicaciones. La solución se explica en el Ejemplo 50.

Figura 73: Solución Basada en G-EEG para Integrar Aplicaciones



Ejemplo 50: Solución Basada en G-EEG para Integrar Aplicaciones

Reconsidere el Ejemplo 26, con la aplicación de back-office legada emitiendo servicios de licencias en IACM. La Figura 73 representa una solución basada en G-EEG para automatizar interacciones entre dicha aplicación legada y otras aplicaciones que toman parte en el proceso de negocios. La solución incluye a siete miembros de G-EEG: BO – actuando en nombre de la aplicación legada, y DSSOPT, DSAL, IC, CB, SS y FO – representando a las aplicaciones involucradas en el proceso de negocios. Los canales c5, c6, c7 y c8 son creados por BO para intercambiar mensajes con las otras organizaciones asociadas como se explicó en el Ejemplo 48. Las siguientes extensiones son habilitadas y configuradas por BO: (1) Auditoría (A) en los canales c5 y c8, ya que los mensajes enviados a DSSOPT y FO pueden tener que ser recuperados para responder a quejas; (2) Validación (V) en los canales c5, c6 y c7 para proteger la aplicación legada de estructura o tipos de mensajes incorrectos recibidos por BO desde esos canales; y (3) Transformación (T) en los canales c5 y c6, ya que los mensajes enviados a

través de tales canales pueden requerir formatos especiales que no pueden ser garantizados por B0. Finalmente, asumiendo que la aplicación legada no puede acceder a las APIs de G-EEG de manera directa, existe un adaptador para que B0 pueda enviar y recibir mensajes a través del prototipo. El adaptador solicita servicios al miembro de G-EEG y envía los mensajes recibidos a la aplicación legada.

6.2.5 Interoperabilidad Sintáctica

Requerimientos: Un problema técnico clave para garantizar la interoperabilidad sintáctica entre aplicaciones es definir y adoptar un estándar común para el intercambio de información entre aplicaciones.

Solución: XML es un estándar común para el intercambio de información que ofrece una parte de la solución para resolver el desafío de interoperabilidad sintáctica. Una notación precisa para escribir datos auto-descriptos, XML define la sintaxis de los documentos y los mensajes intercambiados entre aplicaciones. G-EEG adopta XML como el lenguaje estándar para escribir mensajes, resolviendo directamente el problema de interoperabilidad sintáctica. A través de sus extensiones, G-EEG también utiliza diferentes tecnologías en base a XML: XML Schema para la extensión de Validación y XSLT para la extensión de Transformación.

Justificación: Justificamos que una solución basada en G-EEG es válida para enfrentar el desafío de interoperabilidad sintáctica, demostrando como cumple con los requerimientos descriptos arriba:

1) Todos los mensajes de G-EEG son escritos en XML.
2) XML apoya Unicode [Uni09], un sistema de codificación de caracteres que soporta a la mayoría de los lenguajes escritos contemporáneos.
3) Todos los principales lenguajes de programación proveen soporte para XML, incluyendo C++. Java, Cobol, Perl, etc.

A continuación se presenta una solución basada en G-EEG para el desafío de Interoperabilidad Sintáctica.

Ejemplo 51: Solución Basada en G-EEG para Interoperabilidad Sintáctica

Reconsidere el Ejemplo 48. Ya que todos los socios organizacionales están de acuerdo en intercambiar mensajes escritos en XML como parte de las colaboraciones soportadas por G-EEG, la aplicación de back-office en IACM es capaz de intercambiar mensajes con todos ellos. La solución propuesta basada en G-EEG resuelve el desafío presentado en el Ejemplo 27.

6.2.6 Interoperabilidad Semántica

Requerimientos: El principal desafío técnico para interoperabilidad semántica es establecer un entendimiento común de los conceptos compartidos entre todos los socios involucrados en la producción y entrega de SPE. Como se explicó en la Sección 2.4.2, un enfoque común es el uso de ontologías (Definición 20). Sin embargo, careciendo de un enfoque aplicado a todo lo largo de gobierno para el desarrollo de ontologías, las agencias pueden usar sus propios vocabularios o definir sus propias ontologías. La misma situación surge en el contexto de servicios trans-nacionales entregados por múltiples administraciones públicas, cada una con sus propias ontologías. En tales escenarios, otro problema técnico es mediar el significado entre diferentes vocabularios u ontologías.

Solución: Asumiendo que hay dos socios, A y B, usando diferentes vocabularios `Voc(A)` y `Voc(B)`, aquí se presenta una solución basada en G-EEG para el intercambio de mensajes entre A y B que al mismo tiempo resuelve el desafío de interoperabilidad semántica: (1) cada socio registra un miembro – `mem-A` y `mem-B`; (2) cada miembro registrado crea un canal para enviar mensajes al otro socio – `mem-A` crea `A-B` y `mem-B` crea `B-A`; (3) cada miembro se suscribe al canal creado por el otro miembro – `mem-A` se suscribe a `B-A` y `mem-B` se suscribe a `A-B`; (4) cada propietario de canal habilita la extensión de Transformación en el canal que posee – `mem-A` en `A-B` y `mem-B` en `B-A`; y (5) cada propietario de canal configura la extensión de Transformación con un mapeo desde su propio vocabulario al vocabulario usado

por el receptor del mensaje – mem-A configura la extensión con el mapeo desde Voc(A) a Voc(B), mientras que mem-B configura la extensión con el mapeo desde Voc(B) a Voc(A).

Justificación: Justificamos que la solución basada en G-EEG es válida para enfrentar el desafío de interoperabilidad semántica, demostrando como cumple con los requerimientos descriptos arriba:

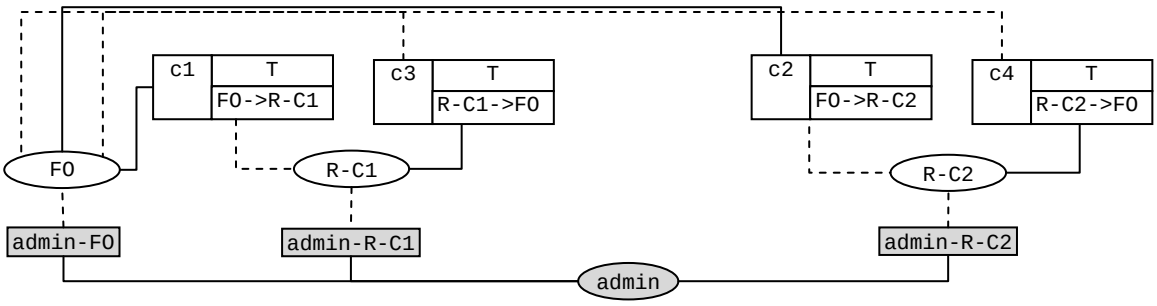
1) Los socios pueden intercambiar mensajes a través de estructuras de comunicación y servicios provistos por G-EEG.
2) Los mensajes intercambiados entre socios son escritos en XML. De este modo, socios colaborativos pueden procesar la sintaxis de los mensajes intercambiados, un prerrequisito para la interoperabilidad semántica.
3) El vocabulario usado por mensajes XML puede ser transformado usando XSLT, con reglas de transformación precisas especificadas por las plantillas XSLT. Habilitando y configurando la extensión de Transformación en un canal, un miembro suscriptor puede recibir mensajes expresados en el vocabulario que es capaz de entender.

A continuación se presenta una solución basada en G-EEG para el desafío de Interoperabilidad Semántica.

Ejemplo 52: Solución Basada en G-EEG para Interoperabilidad Semántica

Reconsidere el Ejemplo 28 en donde la aplicación de front-office (F0) del servicio de licencias solicita la colaboración de los registros centrales en varios países de la región para verificar el registro de los postulantes en sus países de origen. La Figura 74 presenta una solución a este problema basada en G-EEG, asumiendo que: (1) F0 colabora con dos registros ubicados en diferentes países representados por los miembros R-C1 y R-C2, y (2) F0, R-C1 y R-C2 usan los vocabularios Voc(F0), Voc(R-C1) y Voc(R-C2), respectivamente. A los efectos de garantizar la interoperabilidad semántica entre dichas aplicaciones, se crean los siguientes canales: F0 crea canales c1 y c2 para enviar mensajes a R-C1 y R-C2 respectivamente, mientras que R-C1 y R-C2 crea los canales c3 y c4, respectivamente para enviar mensajes a F0. R-C1 está suscripto a c1 y R-C2 está suscripto a c2, mientras que F0 se suscribe a c3 y c4. La extensión de Transformación (T) está habilitada en todos los canales, y tales extensiones son configuradas con los siguientes mapeos: canal c1 - mapeo de Voc(F0) a Voc(R-C1), canal c2 - mapeo de Voc(F0) a Voc(R-C2), canal c3 - mapeo de Voc(R-C1) a Voc(F0), y canal c4 - mapeo de Voc(R-C2) a Voc(F0). En base a esta estructura, cada miembro recibe mensajes expresados en su propio vocabulario.

Figura 74: Solución Basada en G-EEG para Asegurar Interoperabilidad Semántica



6.2.7 Subcontratación Flexible

Requerimientos: A fin de poder soportar la subcontratación flexible para las responsabilidades involucradas en la entrega integrada de SPE, se requiere que la comunicación entre los socios colaborativos pueda ser reconfigurada dinámicamente cuando nuevos socios se unen a la coalición, cuando los socios existentes dejan la coalición, o cuando las responsabilidades de los socios cambian. Con este fin, la solución TIC subyacente debería poder agregar o remover socios a/de la coalición, y crear, configurar y destruir estructuras de comunicación usadas por ellos sin producir trastornos notables en la provisión de servicios.

Solución: En forma nativa, G-EEG resuelve estos requerimientos. Además de los servicios de mensajería básicos de G-EEG-CORE, algunas extensiones de G-EEG podrían también ser usadas. Esto incluye todas las extensiones que apoyan a los procesos inter-organizativos (Sección 6.2.2): Orden, Puntualidad, Composición de Canales y Autenticación. La subcontratación flexible de SPE también puede involucrar: (1) Cifrado - para proteger el intercambio de información sensitiva a través de los límites organizativos, (2) Puntualidad - para controlar el cumplimiento de las políticas que regulan la entrega de servicios subcontratados, y (3) Auditoría - para mantener registros históricos de las solicitudes de servicios procesados por socios no gubernamentales.

Justificación: Justificamos que la solución basada en G-EEG es válida para enfrentar el desafío de la subcontratación flexible, demostrando como cumple los requerimientos descriptos arriba:

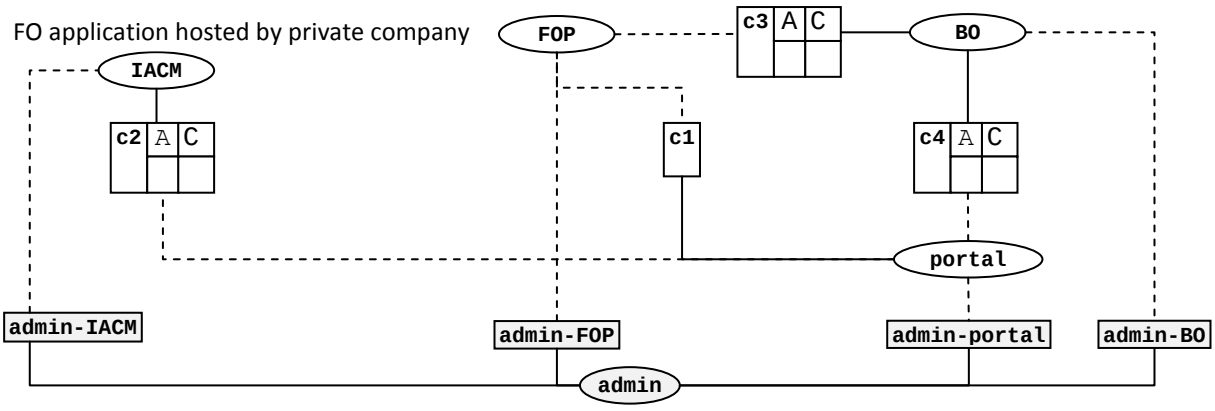
1) Todos los servicios de mensajería centrales – registrar y des-registrar miembros**,** crear y destruir canales, suscribir y des-suscribir miembros a/de canales, y enviar mensajes a miembros - pueden ser invocados dinámicamente en tiempo de ejecución sin afectar las estructuras de comunicación existentes.
2) Todos los servicios de mensajería extendidos – habilitar, configurar y deshabilitar extensiones - puede ser invocados dinámicamente en tiempo de ejecución sin causar interrupciones al sistema.

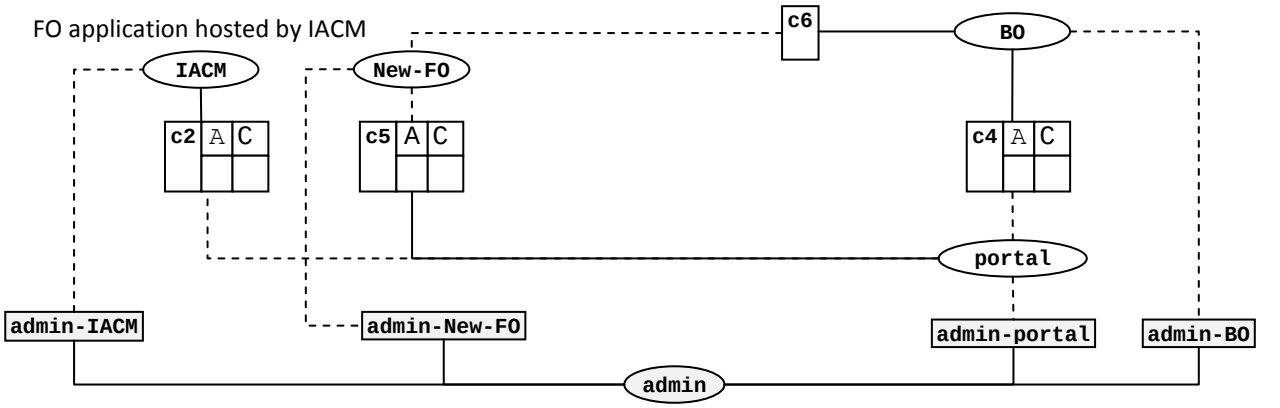
A continuación se presenta una solución basada en G-EEG para el desafío de Subcontratación Flexible.

Ejemplo 53: Solución Basada en G-EEG para Asegurar Subcontratación Flexible

Reconsidere el Ejemplo 29 en donde una compañía privada provee el portal de gobierno y la aplicación de front-office para el servicio de licencias. La Figura 75 debajo presenta dos estructuras de comunicación que soportan este escenario. El primero (arriba) asume que los miembros `Portal` y `FOP` residen en la compañía privada, comunicándose con dos miembros en la agencia de licencias - `IACM` y `BO`. Los miembros residentes en la compañía privada (`Portal` y `FOP`) intercambian mensajes a través de un simple canal `c1` sin extensiones, mientras que todas las comunicaciones que se llevan a cabo a través de limites organizacionales - entre `Portal` y `IACM` a través de `c2`, entre `FOP` y `BO` a través de `c3`, y entre `BO` y `Portal` a través de `c4`, es soportado por las extensiones de Cifrado (`C`) y Auditoría (`A`). El segundo (abajo) muestra la estructura de comunicación modificada luego de que `IACM` asume la responsabilidad de `FO`, reemplazando a la aplicación provista por la compañía privada. Para poder adaptar la estructura de comunicación luego de este cambio, las siguientes acciones son ejecutadas: (1) `FOP` solicita des-suscribirse de `c1` y `c3` y des-registrarse, (2) `portal` destruye `c1` y `BO` destruye `c3`, (3) un nuevo miembro `New-FO` es registrado para representar la nueva aplicación de front-office, (4) `Portal` crea un nuevo canal `c5` para comunicarse con `New-FO` y habilita extensiones de Cifrado y Auditoría en este canal, (5) `BO` crea un nuevo canal `c6` para comunicarse con `New-FO` y (6) `New-FO` se suscribe a `c5` y `c6`. Todas las acciones pueden ser dinámicamente ejecutadas por G-EEG sin afectar otras estructuras existentes.

Figura 75: Solución Basada en G-EEG para Asegurar Subcontratación Flexible





6.2.8 Ecosistema Dinámico

Requerimientos: A fin de entregar servicios públicos integrados a través de una coalición dinámica formada por organizaciones de los sectores públicos, privados y voluntarios (government-enterprise ecosystem), el desafío principal es mantener la estabilidad en la entrega de SPE a pesar de los cambios constantes en la coalición subyacente. Todos los requerimientos técnicos de la Sección 6.2.7 siguen siendo válidos para este desafío. Adicionalmente, se requiere poder descubrir a nuevos miembros para la coalición, mantener un registro de todos los miembros existentes y cómo participan en colaboraciones, etc.

Solución: G-EEG aplica un enfoque común para este desafío, dependiendo en servicios de directorios tales como Páginas Amarillas y Blancas. Las Páginas Amarillas ayudan a adquirir, mantener y proveer información acerca de organizaciones y servicios ofrecidos por ellas, mientras que las Páginas Blancas incluyen información específica acerca de las organizaciones. UDDI provee un ejemplo de cómo se pueden implementar tales servicios. En UDDI, la registración de un negocio comprende: (1) proveer la(s) dirección(es) de la empresa, contactos e identificadores conocidos – Páginas Blancas, (2) categorizaciones industriales de las actividades de la empresa – Páginas Amarillas, y (3) información técnica acerca de los servicios ofrecidos por la empresa – Páginas Verdes. Siguiendo un enfoque similar, G-EEG ofrece la extensión de Localización a través de la cual los miembros pueden consultar sobre: miembros y canales relacionados a ellos a través de la propiedad, suscripción o habilitación; e información específica sobre miembros, canales y extensiones.

Justificación: Justificamos que la solución basada en G-EEG es válida para enfrentar el desafío del ecosistema dinámico, demostrando como cumple con los requerimientos descriptos arriba:

1) Con la extensión de Localización entregada, existe un miembro dedicado (discover) para proveer tales servicios.
2) G-EEG permite a cualquier miembro habilitar la extensión de Localización, resultando en un canal conectando a dicho miembro con el mimebro dedicado de la extensión - discover.
3) Localización permite a los miembros consultar acerca de otros miembros y los canales relacionados a ellos a través de posesión o suscripción; o canales, miembros y extensiones relacionados a ellos a través de posesión, suscripción o habilitación. El servicio de Localización permite a miembros consultar acerca de los detalles de extensiones, canales y miembros específicos, una vez que se conozca la identidad del miembro, canal o extensión.
4) Los miembros pueden solicitar servicios de Localización enviando una solicitud al miembro dedicado de la extensión (discover) a través de un canal creado específicamente para tal fin cuando la extensión es habilitada.

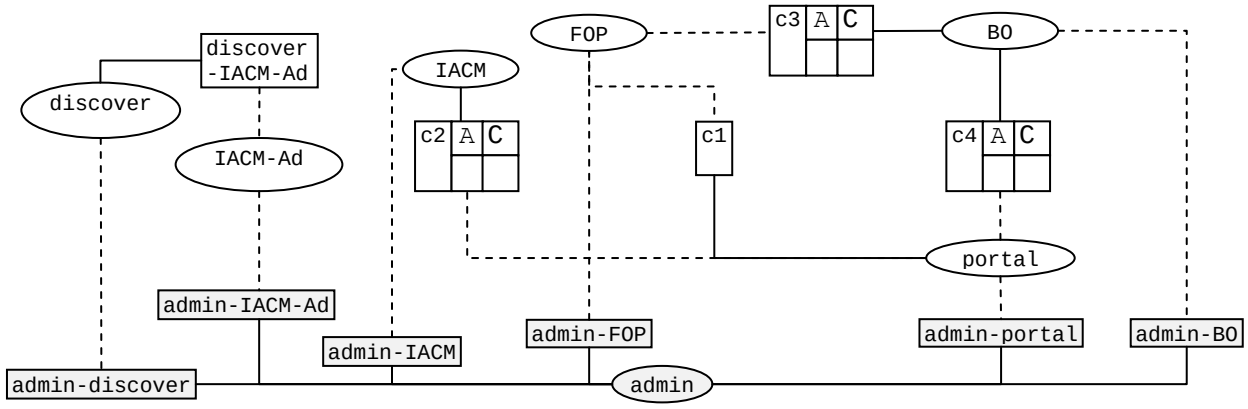
A continuación se presenta una solución basada en G-EEG para el desafío de Ecosistema Dinámico.

Ejemplo 54: Solución Basada en G-EEG para Soportar un Ecosistema Dinámico

Reconsidere el Ejemplo 30. Asuma que la extensión de Localización es entregada en el sistema y que el miembro `discover` es el miembro dedicado de la extensión. La figura debajo presenta una solución basada en G-EEG para soportar el servicio de licencias entregadas por IACM. Para poder acceder a los servicios de Páginas

Amarillas/Blancas, el administrador de servicios de IACM registra un miembro – `IACM-Ad` y habilita la extensión de Localización. Como resultado, el canal `discover-IACM-Ad` es creado por `discover` y `IACM-Ad` es suscripto a él. Posteriormente, `IACM-Ad` puede solicitar información acerca de miembros y canales involucrados en el servicio de licencias. Luego de recuperar información, `discover` puede responderle a `IACM-Ad` con la lista de los miembros o canales involucrados en el servicio de licencias.

Figura 76: Solución Basada en G-EEG para Soportar un Ecosistema Dinámico



6.2.9 Entrega por Múltiples Canales

Requerimientos: Cuando se implementan estrategias para la entrega de servicios públicos integrados a través de múltiples canales, el principal desafío es cómo puede el mecanismo de entrega cumplir con los requerimientos técnicos específicos de cada tipo de canal usado. Un enfoque común consiste en que la solución se base en una arquitectura de software con componentes específicos para cada canal, responsables de transformar información del formato estándar, como XML, al formato entendible por los dispositivos y canales específicos. Por ejemplo, componentes que transformen XML a VoiceXML [W3C04h] para soportar la diseminación a través de teléfono usando navegadores de voz, o componentes que transformen XML a WML [Meh08] para soportar la diseminación a través de dispositivos móviles.

Solución: La solución basada en G-EEG para este desafío se basa en la arquitectura de software descripta arriba. Asumiendo que hay tres canales disponibles – SMS, e-mail y teléfono, la arquitectura comprende un componente responsable de enviar notificaciones a quienes solicitan SPE en base a la información recuperada de la base de datos central conteniendo el estado de tales solicitudes, y tres componentes responsables de transformar mensajes de notificación a los formatos entendidos por cada uno de los canales usados. La solución puede ser construida en cuatro pasos: (1) un miembro es registrado por cada componente de la arquitectura - `Notify`, `SMS`, `e-Mail` y `Phone`; (2) cada miembro registrado crea un canal - `Notify` crea `c1`, SMS el canal `c2`, `e-Mail` el canal `c3` y Phone el canal `c4`; (3) los miembros `SMS`, `e-Mail` y `Phone` habilitan la extensión de Transformación en los canales que poseen y configuran esta extensión para producir el formato adecuado para el canal - SMS habilita y configura la Transformación en `c2`, `e-Mail` en `c3` y `Phone` en `c4`; (4) el miembro `Notify` habilita la extensión de Composición de Canales para dividir el canal `c1` en los canales `c2`, `c3` y `c4`, y configura la extensión para poder distribuir los mensajes según el valor del elemento `channelType` del mensaje.

Justificación: Justificamos que la solución basada en G-EEG es adecuada para enfrentar el desafío de entrega por múltiples canales, demostrando como cumple con los requerimientos descriptos arriba:

1) Las estructuras de comunicación de G-EEG habilitan el intercambio de mensajes entre el componente de notificación y los componentes responsables de administrar diferentes tipos de canales de entrega.
2) La extensión de Composición de Canales con el operador de Ruteo habilita el envío de mensajes de notificación, recibidos a través de un sólo canal, a los tres tipos de canales de entrega soportados por el sistema.

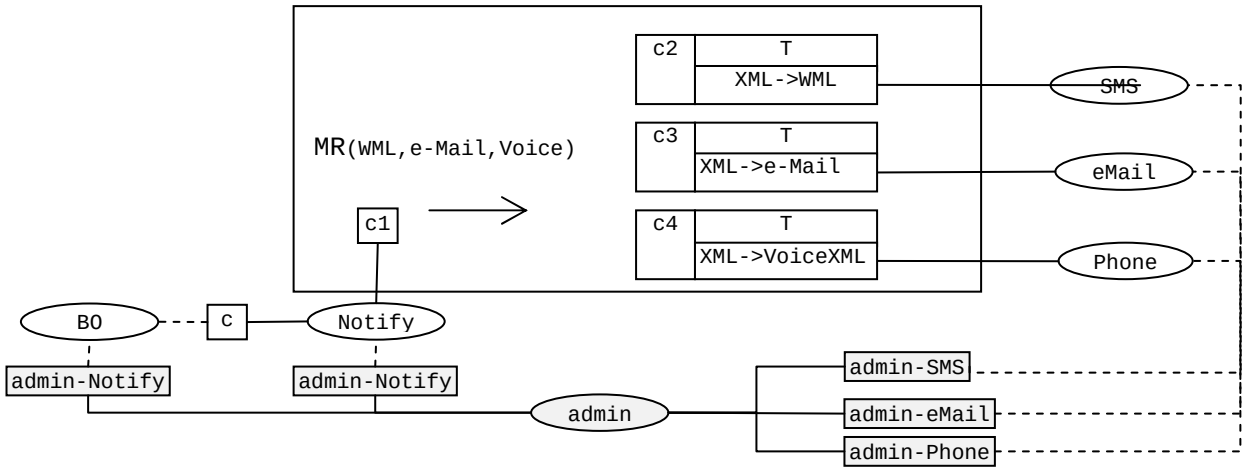
3) La extensión de Transformación configurada para cada tipo de canal de entrega asegura que los mensajes recibidos en el formato estándar XML son transformados al formato entendido por el canal correspondiente.

A continuación se presenta una solución basada en G-EEG para el desafío de Entrega por Múltiples Canales.

Ejemplo 55: Solución Basada en G-EEG para Entrega de Licencias a través de Múltiples Canales

Reconsidere el Ejemplo 31. Suponga que los postulantes pueden seleccionar uno de tres canales - SMS, e-Mail o Teléfono - para recibir notificaciones. La figura debajo representa una solución basada en G-EEG para el desafío de entrega de licencias a través de múltiples canales. La solución comprende cinco miembros representando las siguientes aplicaciones y componentes: (1) BO – la aplicación de back office del servicio de licencias, (2) `Notify` – el componente de notificación, y (3) SMS, `e-Mail` y Phone – los componentes para administrar los canales de SMS, e-Mail y Teléfono. También comprende los siguientes canales: el canal c creado por BO para enviar mensajes de notificación relacionados a las licencias a `Notify`; el canal `c1` creado por `Notify` para enviar mensajes a los postulantes; el canal `c2` creado y configurado por SMS para transformar mensajes de XML al formato WML entendido por los dispositivos de SMS; el canal `c3` creado y configurado por `e-Mail` para transformar mensajes de XML a formatos entendidos por los clientes que usan e-mail; y el canal `c4` creado y configurado por Phone para transformar mensajes de XML a VoiceXML, el formato entendido por dispositivos móviles. Finalmente, `Notify` habilita la extensión de Composición de Canales y la configura para separar el canal `c1` en `c2`, `c3` y `c4`. La solución habilita a `Notify` a enviar mensajes de notificación a un sólo canal, con administradores individuales de canales transformando los mensajes en los formatos entendidos por el dispositivo de destino. La solución enfrenta el desafío presentado en el Ejemplo 31.

Figura 77: Solución Basada en G-EEG para Entrega de Licencias a través de Múltiples Canales



6.2.10 Requerimientos de Dependabilidad

Requerimientos: Los requerimientos de dependabilidad implican siete propiedades no funcionales: (1) Disponibilidad, (2) Confiabilidad, (3) Seguridad, (4) Puntualidad (5) Supervivencia, (6) Recuperabilidad y (7) Mantenibilidad.

Solución: Los requerimientos de dependabilidad deben ser garantizados durante el desarrollo del sistema, la operación y el mantenimiento, y no simplemente agregando partes o capas responsables por requerimientos individuales. Como resultado, las soluciones basadas en G-EEG tratan los requerimientos de dependabilidad durante el diseño, desarrollo y aplicación, como se explica a continuación:

- *Diseño* – El diseño de G-EEG comprende un pequeño, pero totalmente funcional, grupo de servicios centrales, y varias extensiones para llevar a cabo una rica mensajería, habilitados individualmente por usuarios en base a sus

necesidades. Las extensiones pueden ser creadas a través de un proceso riguroso de especificación, diseño y verificación, tratando de este modo los requerimientos de disponibilidad y confiabilidad. Las extensiones también puede ser usadas para implementar el nivel de seguridad requerido, por ejemplo: cifrando mensajes, autenticando miembros, y adoptando políticas específicas con respecto a la suscripción de canales o el intercambio de mensajes.

- *Desarrollo* – G-EEG fue desarrollado siguiendo un proceso riguroso – elicitación de requerimientos, modelado del dominio y casos de uso, arquitectura y diseño detallado, e implementación y entrega, todos documentados con diagramas UML apropiados. Un nivel razonable de confiabilidad ha sido alcanzado como resultado, especialmente en los servicios de mensajería básicos.
- *Aplicaciones* – La infraestructura le da la posibilidad a las agencies de intercambiar mensajes a través de un grupo de canales lógicos cuidadosamente administrados y el uso prudente de extensiones, incluyendo extensiones para implementar seguridad. La confiabilidad en el intercambio de mensajes es asegurada por el núcleo central subyacente y por los propietarios de los canales que realizan una cuidadosa administración de canales y del tránsito de mensajes. La disponibilidad es asegurada por la entrega de mensajes en dos etapas, desde el remitente hasta al propietario del canal y desde el propietario del canal a todos los suscriptores. La seguridad es garantizada por la aplicación de políticas seleccionadas y el uso de extensiones orientadas a seguridad.

Justificación: G-EEG satisface con los siete requerimientos de dependabilidad de la siguiente manera:

1) *Disponibilidad* – Una aplicación externa siempre puede solicitar recuperar a su miembro (requerimiento F6) y de este modo, la aplicación tiene asegurada la disponibilidad de los servicios de mensajería. La solicitud es aprobada por el miembro Administrador quien siempre está disponible (requerimiento NF28).
2) *Confiabilidad* – El mecanismo de G-EEG para el intercambio de mensajes garantiza que cada mensaje sea entregado al menos una vez (requerimiento NF29) y que los mensajes no son reordenados (requerimiento NF30).
3) *Seguridad* – G-EEG entrega tres propiedades de seguridad: (1) Autenticación – a través de la extensión de Autenticación los miembros son identificados antes de solicitar servicios de mensajería; (2) Autorización – diferentes roles de miembros son identificados para acceder a servicios de mensajería, como administrador, propietario de canal o suscriptor; y (3) Confidencialidad – a través de la extensión de Cifrado, los mensajes en tránsito no pueden ser leídos por partes no autorizadas.
4) *Puntualidad* – Evaluada en base a la puntualidad para la entrega de mensajes, la performance del sistema fue cuidadosamente tratada durante el diseño al definir un mínimo conjunto de conceptos y funciones para implementar los servicios de mensajería centrales.
5) *Supervivencia* – Como el administrador está siempre presente en el sistema (requerimiento NF28), cualquier miembro puede ser reiniciado en cualquier momento (requerimiento F6) y, una vez disponible, recibe los mensajes que le fueron enviados cuando estaba inactivo, sus mensajes nunca se pierden (requerimiento NF29). Esto garantiza la existencia de un nivel mínimo de funcionalidad, aún frente a errores y fallas parciales.
6) *Recuperabilidad* – Siempre es posible recuperar a un miembro de un error simplemente reiniciándolo (requerimiento F6), dependiendo de la presencia continua del administrador en el sistema (requerimiento NF28).
7) *Mantenibilidad* – La mantenibilidad de G-EEG está garantizada en dos niveles: (a) aplicación – nueva funcionalidad requerida por las aplicaciones que utilizan la mensajería puede ser entregada, habilitada, configurada y deshabilitada en tiempo de ejecución, sin necesidad de modificar las aplicaciones existentes; y (b) infraestructura – debido al riguroso proceso de desarrollo aplicado, el diseño modular y la documentación del desarrollo, el sistema es relativamente fácil de mantener.

A continuación se presenta un ejemplo de cómo G-EEG trata el desafío de los Requerimientos de Dependabilidad.

Ejemplo 56: Solución Basada en G-EEG para Entrega de Licencias – Requerimientos de Dependabilidad

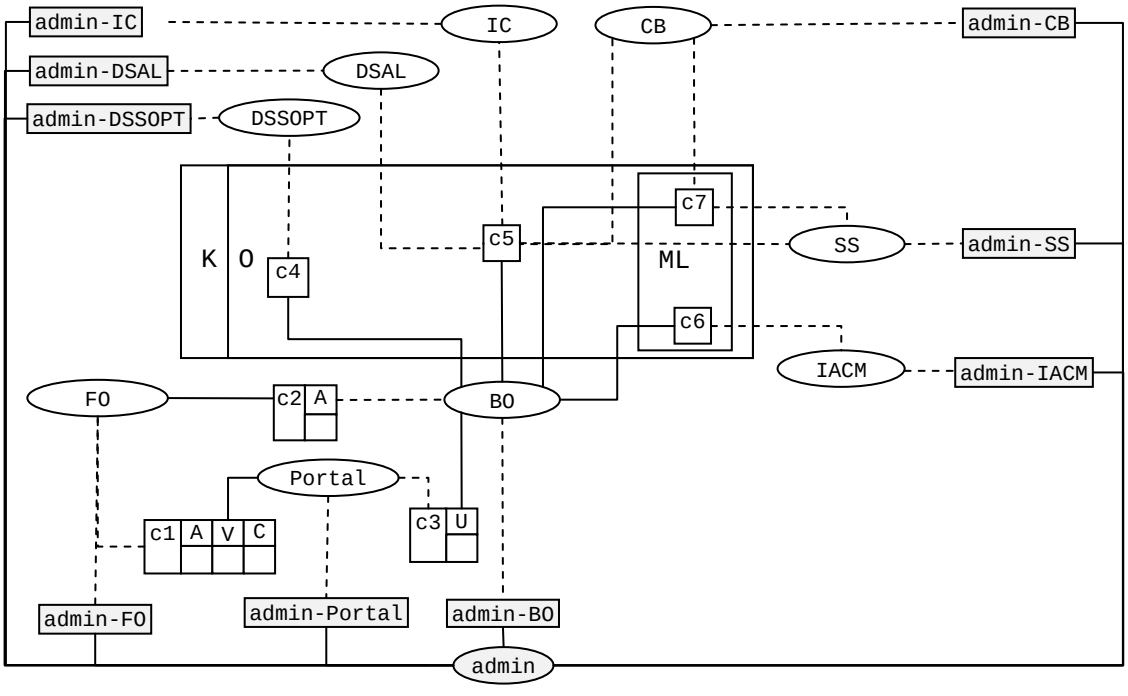
Considere los requerimientos no funcionales especificados en el Ejemplo 32. Los requerimientos pueden ser resueltos mediante una solución basada en G-EEG de la siguiente manera: (1) Disponibilidad – ya que un miembro siempre puede ser reiniciado, cada vez que el portal emite una solicitud de servicio, el miembro que actúa en nombre del portal siempre puede enviar un mensaje a la aplicación en IACM; (2) Confiabilidad – los mensajes intercambiados entre agencias colaboradoras no se pierden y son entregados en el orden en el cual fueron enviados; (3) Seguridad – habilitando extensiones de Autenticación y Cifrado en el canal usado para intercambiar información sensitiva, como por ejemplo entre DSAL e IACM; (4) Puntualidad – la extensión del Puntualidad puede monitorear el tiempo de respuesta cuando se produce intercambio de mensajes entre socios; (5) Supervivencia – las agencias

siempre pueden enviar mensajes a través de G-EEG ya que sus miembros pueden ser reiniciados en cualquier momento, dependiendo de la continua disponibilidad del miembro Administrador; (6) Recuperabilidad – las extensiones de Validación y Auditoría pueden garantizar que sólo mensajes válidos sean entregados, y que los mensajes auditados puedan ser recuperados y reenviados en caso de errores pasajeros; y (7) Mantenibilidad – nuevas agencias y otros socios pueden ser agregados al sistema en tiempo de ejecución al poder registrar miembros para comunicarse en su nombre, y sus opiniones pueden ser solicitadas por medio de intercambio de mensajes a través de nuevos canales creados, sin interrumpir los mecanismos de entrega de servicios ya existentes.

6.3 Caso de Estudio

En esta sección explicamos cómo G-EEG puede soportar la entrega integrada del servicio de licencias según el estudio descripto en la Sección 1.3. La Figura 78 representa una arquitectura basada en G-EEG que puede implementar el proceso de negocios para emitir licencias de negocios en la Figura 3. La inicialización básica para que G-EEG pueda soportar el proceso de licencias es ofrecer un portal de acceso único a través del cual los ciudadanos pueden comunicarse con el gobierno y sus varias agencias para solicitar información, postularse para servicios, realizar el seguimiento del proceso, concertar citas, etc. Para poder habilitar esto, un miembro es registrado en G-EEG para representar al portal – `Portal`. Dos miembros más son registrados para representar las dos partes del servicio de licencias, la de front office (`FO`) y la de back office (`BO`), ambas administradas por IACM. El front office, back office y el portal de acceso único están conectados a través de los canales `c1`, `c2` y `c3`. Las varias agencias involucradas con el proceso – `DSSOPT`, `DSAL`, `IC`, `CB`, `SS` e `IACM` – también son representadas en G-EEG a través de sus miembros, con canales dedicados conectándolos al back office: `c4` conecta a `DSSOPT`; `c5` a `DSAL`, `IC`, `CB` y `SS`; y `c6` conecta a `IACM`.

Figura 78: Arquitectura Basada en G-EEG para el Servicio de Licencias



Aquí vemos como la arquitectura soporta las diferentes etapas del proceso:

1) *Pre-Solicitud* – En esta etapa un postulante solicita una cita con IACM a través de un portal de acceso único que envía la solicitud a la aplicación front office (`FO`) usando el canal `c1`.
2) *Solicitud* – El canal `c1` también puede ser usado para enviar formularios de postulación y los documentos necesarios recibidos a través del portal a `FO`, con las siguientes extensiones horizontales habilitadas: Auditoría (`A`) para evaluar la cantidad de postulaciones presentadas, Validación (`V`) para garantizar que `FO` reciba información válida, y Cifrado (`C`) para proteger la información.

3) *Evaluación de Completitud* – Una vez completada, la aplicación es enviada desde el front office (F0) al back office (B0) a través de c2. El servicio de Auditoría (A) es habilitado en c2 para realizar el seguimiento de las postulaciones enviadas. Si faltan documentos necesarios, el postulante es informado acerca de esto a través del portal, actualizado usando c1.
4) *Evaluación* – En esta etapa se requieren las opiniones de otras agencias: c4 es usado para solicitar opiniones técnicas de DSSOPT; c5 es usado para solicitar opiniones de DSAL, IC, CB y SS; c6 es usado para comunicarse con IACM; y c7 ayuda a notificar a CB y SS acerca de fechas de inspección confirmadas por el postulante y a recibir resultados de inspección. Además, c3 es usado por B0 para actualizar el estado del proceso en el Portal, en respuesta a las solicitudes de seguimiento. La extensión de Autenticación (U) es habilitada en c3 para garantizar que el estado de los procesos sea sólo confirmado por B0.
5) *Toma de Decisión* – No se requiere ningún servicio de mensajería en esta etapa.
6) *Seguimiento* – Las notificaciones son enviadas a los postulantes a través del portal, actualizado usando c1 o c3.

Para poder soportar la ejecución del proceso, las siguientes extensiones verticales son habilitadas: Orden (O) para garantizar que la secuencia de mensajes intercambiados a través de los canales c4-c7 se ajusta al proceso de negocios; Seguimiento (K) a través de c4-c7 para habilitar a B0 a realizar el seguimiento de la ejecución del proceso; y Composición de Canales con el operador de Vincluación (ML) para unir c6 y c7.

6.4 Soluciones Relacionadas

Esta sección evalúa a G-EEG con respecto a soluciones organizacionales, tecnológicas y fundacionales ya existentes para Gobierno Integrado, como fue presentado en el Capítulo 3. La evaluación se lleva a cabo en dos etapas. Primero, aplicamos el marco de evaluación introducido en el Capítulo 3 para los tipos de soluciones organizacionales, tecnológicas y fundacionales, y evaluamos a G-EEG a través de las tres dimensiones. Segundo, comparamos a G-EEG con las soluciones existentes.

Considerada como una solución organizacional, G-EEG posee las siguientes características:

- *Propósito* – Habilitar la colaboración entre organizaciones de los sectores público, privado y voluntario para poder soportar la entrega integrada de Servicios Públicos Electrónicos.

- *Vistas* – G-EEG provee elementos para el modelado de sistemas con respecto a las Vistas de Información, Proceso y Organizacional. Soporta parcialmente la Vista de la Información a través de descripciones de meta datos de mensajes y formatos de variables, y el uso de XML Schemas para la extensión de Validación; tales descripciones son usadas para definir los datos producidos y consumidos por los procesos. La definición de procesos administrados por la extensión de Orden soporta parcialmente la Vista de Procesos. La Vista Organizacional está soportada por la información mantenida por el miembro Administrador en el repositorio central, que también puede ser usado para modelar redes colaborativas. Sin embargo, la Visión Tecnológica no es soportada por G-EEG.

- *Interoperabilidad* – G-EEG soporta la interoperabilidad técnica a través del uso de software de código abierto y de tecnologías open standards, a través de la adopción de principios de diseño como identificadores asignados por el miembro Administrador, y a través del conjunto de APIs independientes de la plataforma [EDJ09c] para garantizar la integración de sistemas legados. Soporta la interoperabilidad semántica: parcialmente a través de la extensión de Transformación que habilita la modificación del formato y el vocabulario de los mensajes, y totalmente por tres extensiones semánticas – validación, mediación y descubrimiento, basadas en el uso de ontologías [San08]. Finalmente, G-EEG soporta parcialmente la interoperabilidad organizacional a través de estructuras de comunicación creadas y configuradas dinámicamente, y varias extensiones que soportan la ejecución de procesos inter-organizacionales – Orden, Puntualidad, Seguimiento, Composición de Canales, etc.

La Tabla 19, al final de este capítulo, compara a G-EGG con otras soluciones organizacionales. La comparación se explica a continuación.

Organizacionalmente, G-EEG puede ser considerado como un Marco de Interoperabilidad que se enfoca en soportar la ejecución de procesos de negocios colaborativos. A pesar de que los objetivos de G-EEG y de e-GIF son similares, difieren en el tipo de mecanismo aplicado para cumplir con sus objetivos. G-EEG se focaliza en el soporte tecnológico para facilitar el flujo de información entre las organizaciones de los sectores público, privado y voluntario, abarcando

varias soluciones organizacionales. Por otro lado, e-GIF abarca un amplio marco de políticas y un registro de activos organizacionales que permiten el flujo de la información entre las organizaciones públicas. Limitando el alcance a las soluciones organizacionales que apoyan la ejecución de procesos de negocios colaborativos, G-EEG se posiciona por encima ya que provee soporte para el modelado de las Vistas de la Información, Procesos y Organizacional, soporta totalmente la interoperabilidad técnica y semántica, y soporta en parte la interoperabilidad organizacional. Sin embargo, las Arquitecturas Empresariales son en general más completas que los Marcos de Interoperabilidad, y G-EEG, entre todas las soluciones organizacionales, trata un problema específico de los Marcos de Interoperabilidad.

Como solución tecnológica, G-EEG posee las siguientes características:

- *Arquitectura* - Basado en una arquitectura completamente distribuida, G-EEG comprende tres componentes principales: (1) Miembro - tres tipos de miembros son usuario, visitante y administrador, (2) Canal - dos tipos de canales son el canal administrador y el canal definido por usuarios; (3) Extensiones - se definen extensiones orientadas a miembros u orientadas a canales. La arquitectura de G-EEG está detallada en los Capítulos 4 y 5.

- *Atributos No Funcionales* - G-EEG tiene los siguientes atributos no funcionales: (1) Confiabilidad - el software G-EEG [DEJ08] entrega mensajes a las colas de destino almacenadas en discos locales, garantizando la entrega confiable de mensajes a pesar de la no disponibilidad temporal de los miembros receptores; (2) Portabilidad - G-EEG puede ser ejecutado en cualquier plataforma de TI, ya que su implementación se basa en Java y estándares abiertos; (3) Mensajes Auto-descriptos - como los mensajes de G-EEG están escritos en XML, están auto-definidos a través de etiquetas XML; (4) Paradigma de Comunicaciones - G-EEG apoya de manera nativa la comunicación asincrónica, con intercambio sincrónico simulado a través de extensiones G-EEG; (5) Mecanismo de Distribución - G-EEG soporta tanto los los mecanismos punto-a-punto como el de publicar-suscribir para la entrega de mensajes, al permitir suscribir uno o varios miembros a un canal; y (6) Limitaciones – las restricciones de G-EEG, como por ejemplo el tamaño del mensaje, el número máximo de canales suscriptos, el número máximo de extensiones habilitadas, etc. son definidos en un archivo de configuración de G-EEG y son determinadas específicamente para cada nodo dependiendo de la capacidad del hardware.

- *Atributos Funcionales* - G-EEG satisface los atributos funcionales de la siguiente manera: (a) Servicio de Recupero puede ser provisto implementando un extensión orientada a miembros que ayude al miembro a recuperar mensajes en base a su tipo, prioridad y el canal recibido; (b) Servicio de Auditoría es ofrecido a través de la extensión de Auditoría; (c) Servicio de Validación es ofrecido por la extensión de Validación; (d) Servicio de Transformación es ofrecido a través de la extensión de Transformación; (e) Servicio de Autenticación es ofrecido a través de la extensión de Autenticación; (f) Servicios de Nombrado es provisto ya que G-EEG permite asignar nombres a miembros y canales; (g) Servicio de Localización es ofrecido a través de la extensión de Localización; (h) Servicio de Transacción puede ser implementado por una extensión orientada a miembros; (i) Servicio de Composición es soportado ya que varias extensiones orientadas a miembro pueden ser habilitadas por un miembro y varias extensiones orientadas a canales pueden ser habilitadas en un canal; (j) Servicio de Desarrollo es provisto por G-EEG-DEVELOP.

La Tabla 20, al final de este capítulo, compara G-EEG con otras soluciones tecnológicas. Tecnológicamente, G-EEG es un MOM programable, desarrollado como una solución de dominio específico para Gobierno Integrado. Excepto por BEA MessageQ e IAMS, todas las soluciones, incluido G-EEG, ofrecen una arquitectura distribuida. G-EEG, como la mayoría de las soluciones, garantiza que los mensajes no se pierden y son entregados una sola vez. Mientras que la mayoría de las soluciones están disponibles para plataformas específicas de TI, SOA, JADE y G-EEG son independientes de la plataforma. En G-EEG, como en MSMQ, IAMS y SOA, los mensajes son escritos en XML. Todas las soluciones soportan el intercambio sincrónico y asincrónico de mensajes. Sin embargo, G-EEG y WebSphere MQ apoyan de manera nativa las comunicaciones asincrónicas y pueden simular comunicaciones sincrónicas. Excepto por Hermes2 y JADE, todas las soluciones aceptan la entrega de mensajes punto-a-punto y publicar-suscribir. La mayoría de las soluciones operan bajo ciertas limitaciones, basadas en el hardware y software que las soportan. En cuanto a los requerimientos funcionales, las soluciones existentes implementan algunos de ellos, mientras que G-EEG provee o puede proveer todas las características definidas por medio de extensiones. Más aún, mientras que SOA y JADE apoyan la composición de servicios a través de herramientas adicionales, la característica es soportada de manera nativa por las especificaciones de G-EEG. Finalmente, solo JADE y G-EEG pueden agregar nuevas funcionalidades a través del desarrollo. En resumen, como una solución tecnológica, G-EEG cumple con los requerimientos no funcionales como lo hacen las otras soluciones, pero para requerimientos funcionales, G-EEG es la única solución tecnológica que provee o puede proveerlos todos, incluyendo el desarrollo de una nueva funcionalidad.

Como solución fundacional, G-EEG posee las siguientes características:

- *Estilo de Especificación* – El modelo de RSL provisto por G-EEG permite especificar extensiones. Se pueden escribir tanto especificaciones basadas en estado como basadas en acciones, usando las expresiones del lenguaje provistas por el modelo RSL.

- *Semántica* – Las operaciones de G-EEG asignan semántica operacional, con el significado de las operaciones definido en término de cambios de estado, tanto local como global, causado por las mismas.

- *Refinamiento* – Debido a que G-EEG fue modelado usando RSL, G-EEG aplica el mismo principio de refinamiento (Sección 3.4.1).

- *Herramientas* – Dos herramientas fueron desarrolladas para soportar la version de calidad de producción del prototipo de G-EEG [DEJ08]: (1) una consola gráfica para invocar y monitorear la ejecución de los servicios de G-EEG y (2) una aplicación web para consultar información sobre los miembros y canales de G-EEG, y para administrar los recursos de conocimiento usados por G-EEG, como XML Schemas usados para validar mensajes, plantillas XSLT usadas para transformar mensajes, etc.

La Tabla 21, al final de este capítulo, compara a G-EEG con otras soluciones fundacionales. Como solución fundacional, G-EEG puede ser considerado un lenguaje de especificación para mensajería programable. Como REO que se focaliza en especificar conectores y su composición en sistemas basados en componentes, G-EEG se concentra en la especificación de funciones de mensajería que pueden soportar la ejecución colaborativa de procesos. Formalizado en RSL, G-EEG puede soportar tanto especificaciones basadas en estado como especificaciones basadas en acciones, siguiendo el principio de refinamiento de RAISE. Semántica operacional es definida para las operaciones de G-EEG, y las extensiones también pueden ser definidas mediante axiomas. Como otros lenguajes, se pueden desarrollar herramientas para G-EEG, con dos herramientas actualmente disponibles: (1) una aplicación de consola para solicitar y monitorear los servicios de G-EEG y (2) una aplicación web para administrar los recursos y para obtener información sobre el estado de G-EEG y sus miembros.

Finalmente, cuatro características distintivas de G-EEG comparado con las soluciones existentes son:

1) G-EEG cruza las dimensiones organizacionales, tecnológicas y fundacionales, si bien se focaliza principalmente en las dimensiones tecnológica y fundacional;
2) G-EEG ofrece servicios de mensajería básicos y enriquecidos, todos dentro del mismo ambiente de desarrollo y producción y con una sola herramienta proveyendo toda la funcionalidad;
3) G-EEG permite crear y configurar dinámicamente las estructuras de comunicación para soportar mensajería básica y extendida, y para entregar nueva funcionalidad bajo una provisión de servicio continuo; y
4) G-EEG fue particularmente diseñado para tratar los requerimientos de Gobierno Integrado.

Tabla 19: Comparación de Soluciones Organizacionales

		e-GIF	NZ e-GIF	EIF	Zachman's Framework	SAGA	FEA	G-EEG
Nombre		e-Government Interoperability Framework	New Zealand e-Government Interoperability Framework	European Interoperability Framework	The Zachman Framework	Standards and Architectures for e-Government Applications	Federal Enterprise Architecture	Government-Enterprise Ecosystem Gateway
Tipo		Marco de Interoperabilidad	Marco de Interoperabilidad	Marco de Interoperabilidad	Arquitectura Empresarial	Arquitectura Empresarial	Arquitectura Empresarial	Parte de un Marco de Interoperabilidad
Origen		Unidad de Gobierno Electrónico de la Oficina del Gabinete del Reino Unido	State Services Commission del Gobierno de Nueva Zelanda	Interoperable Delivery of European e-Government Services to public Administrations, Businesses and Citizens (IDABC)	Originalmente desarrollado por J.A. Zachman	Coordination and Advisory Board for IT in the Adminis-tration (KBSt unit) , Federal Ministry of the Interior, Germany, AG, and Fraunhofer-Institut für Soft ware- und Systemtechnik (IIST)	FEA Program Management Office de Office of Management and Budget (OMB), Estados Unidos de América	Solución propuesta en esta tesis
Descripción		Incluye dos componentes principales: el marco de e-GIF y el registro de e-GIF.	Comprende tres documentos: Estandares, Políticas y Recursos.	Incluye tres componentes: (1) recomendaciones para EPS, (2) recomendaciones para interoperabilidad, y (3) definiciones de interacciones pan-europeas.	El framework está estructurado como una matriz de 5 filas - Planificador, Dueño, Diseñador, Constructor y Subcontratista; y 6 columnas - Datos, Funciones, Red, Gente, Tiempo y Motivación.	Modela aplicaciones de Gobierno Electrónico siguiendo 5 perspectivas: Empresa, Información, Computacional, Ingeniería y Tecnología.	Comprende cinco modelos interrelacionados: Business Reference Model, Performance Reference Model, Service Component Reference Model, Data Reference Model and Technical Reference Model.	G-EEG incluye tres componentes: G-EEG-CORE, G-EEG-EXTEND y G-EEG-DEVELOP
Propósito		Habilitar flujo continuo de información a través de organizaciones públicas del Reino Unido.	Proveer un framework técnico en un modelo de capas para clasificar estándares de TI.	Suplementar, y no reemplazar, las recomenda-ciones de interoperabilidad nacionales agregando la dimensión pan-europea.	Proveer una estructura lógica para clasificar y organizar las representaciones descriptivas de una empresa.	Proveer un conjunto de estándares para hacer possible el desarrollo de Gobierno Electrónico en Alemania.	Asistir en el desarrollo y mantenimiento de arquitecturas empresariales para todas las agencias de gobierno.	Facilitar la colaboración entre organizaciones públicas y privadas para soportar la entrega integrada de SPE.
Vistas	Información	Conceptos, modelos y datos son definidos en e-GMS, GCL, GDSC y XML Schemas.	Se definen estándares para representación e integración de datos, descubrimiento de meta-datos, nombre de espacios y direcciones.	Se definen acuerdos para diccionarios de datos, tesauros multi-lenguajes, y vocabulario XML relacionado a servicios pan-Europeos.	Se incluyen varias represen-taciones en la Perspectiva de Datos – entidades de negocios y de datos y definiciones de datos.	Se incluyen elementos en la Vista de la Información	Se cubren elementos en el Modelo de Referencia de Datos.	Parcialmente soportada por meta-datos usados para definir estructuras de mensajes, y XML Schemas usados para validación.
	Proceso	Estándares para areas verticals son incluídos en el TSC.	Se incluyen recomendaciones y estándares en las capas de Acceso y Presentación.	Los procesos son modelados identificando tres tipos de interacciones entre actores principales.	Los procesos son modelados en las Perspectivas de Función, Tiempo y Motivación.	Se incluyen elementos en la Vista de la Computación.	Se cubren elementos conjuntamente por el Modelo de Referencia de Negocios, el Modelo de Referencia de Servicios y el Modelo de Referencia de Performance.	Parcialmente soportada por definiciones de procesos manejados por la extensión de Orden.

		e-GIF	NZ e-GIF	EIF	Arquitectura de Zachman	SAGA	FEA	G-EEG
	Tecnología	Estándares relacionados con TI están considerados en el TSC.	Algunos estándares están definidos en la capa de Red.	Estándares relacionados con IT son considerados en el framework.	Los aspectos de tecnología son modelados en la perspectiva Donde.	Se incluyen elementos en las Vistas de la Ingeniería y de la Tecnología.	Se cubren elementos en el Modelo de Referencia Técnico.	No es soportada por G-EEG.
	Organizacional	No está considerado por e-GIF.	El documento de políticas incluye elementos sobre el gerenciamiento y la gobernabilidad del marco.	Se incluyen algunas recomendaciones organizacionales.	Algunos elementos son cubiertos en la Perspectiva Gente.	Se incluyen elementos en la Vista de la Ingeniería.	Se cubren elementos en el Modelo de Referencia de Negocios.	Es parcialmente soportada por información almacenada en el repositorio central por el miembro Administrador.
Interoperabilidad	Técnica	Está tratado por políticas técnicas y especificaciones que gobiernan los flujos de información.	Estándares técnicos se incluyen en las capas de Red, Integración de Datos y Seguridad.	Se incluyen recomendaciones técnicas sobre el intercambio de datos y aplicaciones de front-office.	Soportada parcialmente proveyendo plantillas para agregar aspectos técnicos a la arquitectura.	Se incluyen estándares en las Vistas de la Tecnología y de la Ingeniería.	Soportada por el Modelo de Referencia Técnica.	Es soportada por adopción de estándares abiertos, principios de diseño, y un conjunto de APIs independientes de la plataforma.
	Semántcia	Está parcialmente soportado a través de e-GMS.	Está parcialmente soportada por estándares provistos para definir meta-datos y estructuración y modelamiento de datos.	Se identifican importantes cuestiones semánticas y se proveen guías para tratarlas.	Modelos semánticos pueden ser definidos en la Perspectiva de Datos. Alguna información sobre procesos y recursos humanos se incluyen en las Perspectivas Propietario y Gente.	Se soporta a través de la Vista de la Información.	Se soporta con la inclusión de esquemas de clasificación estándar y taxonomías en todos los modelos. Se desarrolló una ontología para soportar el intercambio efectivo de datos entre agencias.	Es parcialmente soportada por la extensión de Transformación, y totalmente soportada por tres extensiones semánticas – Validación, Mediación y Descubrimiento.
	Organizacional	Está parcialmente soportada para areas específicas. Provee guías para entrega de SPE por multiples canales.	No está tratada por NZ e-GIF.	Está soportada por recomendaciones para las interacciones para la entrega de servicios Pan-Europeos.	Soporta conceptos principales como participantes y procesos de negocios internos; mientras otros están cubiertos parcialmente (partes externas) o no cubiertos en absoluto (interacciones con otras organizaciones)	Está soportada por aspectos considerados en la Vista de la Empresa.	Parcialmente soportada por el Modelo de Referencia de Negocios y el Modelo de Referencia de los Componentes de Servicio.	Parcialmente soportada por estructuras de comunicación que pueden ser creadas y configuradas dinámicamente y extensiones que soportan la ejecución de procesos inter-organizacionales.

Tabla 20: Comparación de Soluciones Tecnológicas

		BEA Message Q	WebSphere MQ	MSMQ	IAMS	Hermes2	SOA	JADE	G-EEG
Nombre		Bea MessageQ	WebSphere MQ	Microsoft Message Queue Server	Inter Agency Messaging Service	Hermes Messaging Gateway	Service-Oriented Architecture	Java Agent Development Framework	Government-Enterprise Eco-system Gateway
Tipo		Middleware Orien-tado a Mensajes	Middleware Orientado a Mensajes	Middleware Orientado a Mensajes	Middleware Orientado a Mensajes	Middleware Orientado a Mensajes	Arquitectura de software	Framework de desarrollo basado en agentes	Middleware programmable orien tado a mensajes
Origen		BEA Systems	IBM	Microsoft	Reach Government Agency, Irish Government	Center for e-Commerce Infrastructure Develop-ment (CECID) at Hong Kong University	No clear ownership	Telecom Italia Lab	Solution proposed in the thesis
Descripción		Solución middleware de propósito general	Solución middleware de propósito general	Solución middleware de propósito general	Solución de dominio específico	Solución de dominio específico	Arquitectura de software de propósito general	Solución middleware de propósito general	Solución de dominio específico
Arquitectura		Centralizada, incluye 3 componentes: Message Queue, Message Queuing Group y Message Queuing Bus	Distribuída, incluye 3 componentes: Message Queue, MQ Queue Manager y Channel	Distribuída, incluye 3 componentes: Application Queue, System Queue, y MSMQ Server	Centralizada, comprende un component principal: IAMS-Core	Distribuída, incluye 4 componentes: Corvus-Core Kernel, Registry, ebMS-plug-in, y ADS2-plug-in.	Distribuída, incluye 3 roles: Proveedor de Servicio, Solicitante de Servicio y Registro de Servicio; y 3 operaciones: Publicar, Buscar y Ligar	Distribuída, incluye 3 componentes: Agent, Container, y JADE Platform	Distribuída, incluye 3 componentes principales: Miembro, Canal y Extensión.
No Funcional	Confiabi-lidad	Los mensajes no se pierden usando modo recuperable.	Los mensajes no se pierden y se entregan sólo una vez.	Los mensajes no se pierden usando modo recuperable. Los mensajes se entregan solo una vez, si se son parte de una transacción.	Los mensajes no se pierden. Una sola entrega no está garantizada.	Los mensajes no se pierden y se entregan sólo una vez.	Los mensajes no se pierden y se entregan sólo una vez siguiendo WS-Reliability.	Los mensajes no se pierden y se entregan sólo una vez.	Los mensajes no se pierden y se entregan sólo una vez.
	Portabi-lidad	Puede usarse en plataformas de TI bajo varios sistemas operativos.	Puede usarse en plataformas de TI bajo varios sistemas operativos.	Disponible en todas las plataformas de Microsoft.	Windows 2000 Server y posiblemente disponible en plataformas Microsoft	Se puede usar en plataformas corriendo Windows XP, Linux y Solaris.	Es una arquitectura independiente de la plataforma.	Los agentes pueden residir en diferentes plataformas de TI.	Puede ejecutarse en cualquier plataforma
	Mensaje	Auto-descriptos usando Field Manipulation Language.	Auto-descriptos usando un descriptor de mensaje para cada mensaje.	Usa XML como default. Binary Message y ActiveX Message Formatters, para serializar objetos en formato binario.	Los mensajes son escritos en XML.	Adopta dos estándares para mensajes: ebMS y AS2	Los mensajes son escritos en XML.	Adopta FIPA ACL Message Structure Specification	Los mensajes son escritos en XML.
	Comunicación	Soporta comunicaciones sincrónicas y asincrónicas.	Soporta comunicacio- nes asincrónicas en forma nativa, mientras que las sincrónicas las puede simular.	Soporta comunicaciones asincrónicas. Adicional-mente, puede leer mensajes de manera sincrónica.	Soporta comunicacio- nes asincrónicas. Comunicaciones sincró-nicas pueden soportar-se a través de RMI	Soporta comunicacio- nes asincrónicas.	Soporta comunicaciones sincrónicas y asincrónicas.	Soporta comunicaciones sincrónicas y asincrónicas.	Por definición soporta comunicaciones asin-crónicas. Comunica-ciones sincrónicas pueden ser simuladas por una extensión.

		BEA Message Q	WebSphere MQ	MSMQ	IAMS	Hermes2	SOA	JADE	G-EEG
No Funcional	Distribución	Soporta entrega punto-a-punto y publicar-suscribir.	Soporta entrega punto-a-punto y publicar-suscribir.	Soporta entrega punto-a-punto y publicar-suscribir.	Soporta entrega punto-a-punto y publicar-suscribir.	Soporta entrega punto-a-punto.	Soporta entrega punto-a-punto. La entrega publicar-suscribir está soportada usando las especificaciones de WS-Notification.	Soporta entrega punto-a-punto.	Soporta entrega punto-a-punto y publicar-suscribir.
	Limitaciones	Mensajes hasta 4Mb Message Queue Groups hasta 32,000, cada Grupo hasta 999 Message Queues.	El tamaño de mensaje está limitado por un paramétro. El tamaño máximo es de 100Mb o 2Gb, si se usa DB2.	Mensajes hasta 4 Mb. La capacididad de las Message Queue entre 1.4 y 1.6Gb.	No hay información disponible.	No hay información disponible.	SOAP no define limitación defines no limitation for message size, depends on the communication protocol.	JADE defines no restrictions. Limitations are defined by under-lying technologies.	Bounds are defined in a configuration file, and depend on hardware capacity.
Funcional	Recu-pero	Provisto	Provisto	No provisto	No hay información disponible	Provisto	Puede ser provisto por un servicio web intermedio.	Provisto	Puede ser provisto por una extensión orientada a miembro.
	Audi-toría	Provisto	No provisto	Provisto	Provisto	No provisto	Puede ser provisto por un servicio web intermedio.	Provisto	Provisto por una extension.
	Valida-ción	No provisto	No provisto	No provisto	Provisto	No hay información disponible	Puede ser provisto por un servicio web intermedio.	Provisto	Provisto por una extension.
	Trans-formac	No provisto	No provisto	No provisto	No hay información disponible	Provisto	Puede ser provisto por un servicio web intermedio.	No provisto	Provisto por una extension.
	Auten-ticac.	No provisto	Provisto	Provisto	Provisto	No provisto	Provisto siguiendo WS-Security.	Provisto	Provisto por una extension.
	Nom-brado	Provisto	Provisto	Provisto	Provisto	Provisto	Provisto por UDDI.	Provisto	Provisto
	Localiz.	Provisto	Provisto	Provisto	Provisto (asumido)	Provisto	Provisto por UDDI.	Provisto	Provisto por una extension.
	Transac.	No provisto	No provisto	Provisto	No hay información disponible	No hay información disponible	Puede ser provisto por WS-Coordination y WS-AtomicTransaction	No provisto	Puede ser provisto por una extensión orientada a miembro.
	Composición	No provisto	No provisto	No provisto	No provisto	No provisto	Provisto por lenguajes para modelar procesos de negocio y motores de workflows usados para ejecutar tales procesos.	Provisto a través de tecnologías construidas sobre JADE	Provisto por las especificaciones de G-EEG.
	Desa-rrollo	No provisto	No provisto	No provisto	No provisto	No provisto	No provisto	Provisto a través del framework de desarrollo de JADE	Provisto por G-EEG-DEVELOP.

Tabla 21: Comparación de Soluciones Fundacionales

	RSL	CSP	Reo	Pi-Calculus	G-EEG
Nombre	RAISE Specification Language	Communicating Sequential Processes	Reo	Pi-Calculus	Government-Enterprise Ecosystem Gateway
Tipo	Wide-spectrum specification language	Lenguaje de especificación para sistemas concurrentes	Un lenguaje de coordinación de componentes basado en canales	Un cálculo de procesos – una evolución de CCS (Calculus of Communicating Systems)	Un lenguaje de especificación para mensajería programable
Origen	The RAISE Language Group	Propuesto por Tony Hoare	Propuesto por Arbab y Mavaddat	Propuesto por Robin Milner, Joachin Parrow y David Walker	Solución propuesta en la tesis
Descripción	Provee 3 estilos de especificación: aplicativo, imperative y concurrente. Una especificación está compuesta por un conjunto de módulos, cada uno conteniendo definiciones de tipos, valores, variables, canales, axiomas y otros módulos.	Un sistema concurrente es modelado como un conjunto de procesos, cada uno con su estado privado comprendiendo un conjunto de variables que interactúan entre ellas y con su entorno por medio del intercambio de mensajes via comunicaciones sincronizadas.	Se basa en 4 conceptos importan- tes: Instancia de Componente, Canal, Nodo y Conector. Un sistema es modelado como un conjunto de instancias de compo-nentes ejecutadas en varias ubicaciones, comunicándose a través de conectores. Las instan-cias de componentes y conectores son móviles. Se focaliza en los conectores y sus composiciones.	Pi-calculus es un lenguaje para especificar y analizar propiedades de systemas que consisten de agentes que interactúan entre ellos, y cuyas configuraciones o vecindades están continuamente cambiando.	Se basa en 4 conceptos principales: miembro, canal, mensaje y extensión.
Estilo de Especificación	Soporta especificaciones basadas en estados y en acciones	Soporta especificaciones basadas en acciones	Soporta hasta cierto grado especificaciones basadas en estado.	Soporta especificaciones basadas en acciones	Soporta especificaciones basadas en estados y en acciones
Semántica	Operacional Denotacional Axiomática	Operacional Denotacional Algebraica	Operacional	Operacional Denotacional Coalgebraica	Operacional Axiomática – para algunas extensiones.
Refinamientos	Refinamiento basado en preservación de propiedades y en substitutividad.	Dependiendo del modelo semantico, se pueden definir 3 tipos de refinamiento: refinamiento de trazas, refinamiento de fallas, y refinamiento de fallas/divergencias	Se puede aplicar refinamiento a los conectores de Reo, en base a conceptos de bisimulación.	Bisimulación	Refinamiento está basado en la preservación de propiedades y en substitutividad.
Herramientas	Herramientas para escribir (emacs) y documentar especific.(RSL pretty printer), checkear especificaciones – type checker y generador de con diciones de confianza, verificar especificac. – traductores de RSL a demostradores de teoremas SAL y PVS, ejecutar especificaciones – traductores de RSL a C++ and SML, traducción de diagramas UML a RSL	Herramientas para escribir especificaciones (viM), chequear condiciones de correctitud (FDR), simular el comportamiento de un sistema (ProBE), y producir descripciones CSP para protocolos de seguridad (Casper)	La herramienta Eclipse Coordination provee un conjunto de plug-ins para el entorno de desarrollo de Eclipse, que comprendes herramientas visuals integradas para soportar el modelado y la construcción de circuitos Reo. La herramienta permite transformar el modelo creado en un autómata con restricciones y luego en código Java.	Mobility Workbench (MWB) – para verificar equivalencias de bisimulación, MIHDA – para controlar bisimulación temprana; y Pi-Calculus para SOA para modelar coreografía de servicios web.	Herramientas que proveen una interface de usuario para solicitar servicios de mensajería. Una aplicación web para administrar recursos de G-EEF y para obtener información del estado del sistema.

Capítulo 7

Conclusiones

Luego de presentar los fundamentos (Capítulo 4), la implementación (Capítulo 5) y la evaluación (Capítulo 6) de G-EEG como una realización concreta de Mensajería Programable y una solución para Gobierno Integrado, este capítulo concluye la tesis. Estructurado en tres secciones, el capítulo provee un resumen de los resultados obtenidos (Sección 7.1), esboza la contribución (Sección 7.2) y discute direcciones para investigaciones futuras (Sección 7.3).

7.1 Resumen

La tesis presentó una solución tecnológica que soporta la colaboración entre agencias gubernamentales comprometidas con el desarrollo y la operación de Gobierno Integrado, llamado Government-Enterprise Ecosystem Gateway (G-EEG). G-EEG es un marco de comunicación y coordinación que habilita a procesos y aplicaciones inter-organizacionales a construir y desarrollar estructuras de comunicación complejas en base a mensajería asincrónica. Una realización concreta de Mensajería Programable, G-EEG puede ser considerado una simple, pero poderosa, solución para la integración de procesos y de información, particularmente para Gobierno Integrado.

El Capítulo 1 introdujo el concepto de Gobierno Electrónico – transformación facilitada por tecnología de organizaciones gubernamentales para responder mejor a las necesidades de sus electores - ciudadanos, empresas y otras ramas de gobierno, seguido por Gobierno Integrado – transformación de estructuras jerárquicas a redes de organizaciones conectadas a través de áreas y niveles de gobierno y a través de sectores público, privado y voluntario, con colaboración y entrega colaborativa de servicios integrados como los principios claves de organización. A título ilustrativo fue presentado un caso de estudio de un servicio público integrado real para emitir licencias de negocios a postulantes, en base a la colaboración y al intercambio de información entre varias agencias gubernamentales. Si bien la entrega de servicios integrados ofrece beneficios concretos al gobierno y a sus clientes, su implementación plantea varios tipos de desafíos legales, financieros, sociales, organizacionales y tecnológicos que las agencias deben superar. Enfocándose en los desafíos tecnológicos y organizacionales, hasta el grado en el cual tales desafíos pueden ser solucionados por TIC, el problema tratado en la tesis fue formulado - construir una plataforma de comunicación y de coordinación, con el modelo y la teoría subyacente, para facilitar el establecimiento, la operación y la evolución de redes organizacionales, capaces de entregar servicios públicos de manera integrada. Fue seguido por la presentación de G-EEG como la solución propuesta. Luego de explicar el diseño, su estructura y comportamiento, G-EEG, fue evaluado en base al problema presentado, el conjunto de desafíos tecnológicos y el caso de estudio de las licencias. Finalmente, la contribución y estructura de la tesis fueron delineadas.

El Capítulo 2 presentó los conceptos principales para el dominio de Gobierno Integrado: Agencias, Resultados, Capacidades, Recursos y Desafíos. Agencia es la unidad organizativa clave de las administraciones públicas y el motor principal para la entrega de servicios públicos. Resultados, definidos a través de los conceptos de Clientes, Servicios y Canales, comprende los efectos tangibles o los beneficios del trabajo llevado a cabo por las agencias. Un tipo de resultado particular, el servicio público integrado es ofrecido proactivamente a Clientes a través de contactos únicos, y entregado colaborativamente por agencias, empresas y otras organizaciones que trabajan en conjunto. Los Servicios son entregados a través de varios canales, con canales tradicionales cada vez más integrados a los electrónicos. Las Capacidades representan las habilidades de las agencias para entregar resultados, particularmente para entregar servicios integrados. Incluyen: Colaboración – capacidad para ejecutar procesos de negocios que entregan servicios integrados; Asociación – capacidad para compartir responsabilidades y riesgos con otras organizaciones; Integración – capacidad para conectar procesos de negocios y aplicaciones de software de diferentes socios de servicios; y Coordinación – capacidad para usar recursos compartidos provistos por socios de servicios y para controlar la calidad de la entrega de servicios. Los Recursos comprenden varios tipos de activos - financieros, humanos, organizacionales y tecnológicos – requeridos por las agencias para entregar servicios. Los recursos organizacionales incluyen procesos – actividades que cada socio necesita llevar a cabo para poder producir y entregar un servicio. Mientras que en el

pasado el proceso de automatización era la mayor preocupación, Gobierno Integrado requiere una reingeniería radical de tales procesos a través del uso de las TIC. Recursos tecnológicos comprenden todos los recursos relacionados a las TIC que las agencias necesitan para entregar servicios. La tesis se enfoca en recursos tecnológicos y herramientas que permiten la colaboración entre agencias que entregan servicios integrados de manera conjunta. Por último, Desafíos constituyen varios tipos de barreras que las agencias deben superar para entregar servicios integrados, incluyendo barreras legales, financieras, sociales, organizacionales y tecnológicas. En particular, un número de desafíos tecnológicos fueron introducidos para evaluar la aptitud de cualquier solución tecnológica de Gobierno Integrado.

El Capítulo 3 presentó algunos trabajos relacionados, considerando tres categorías: soluciones organizacionales, tecnológicas y fundacionales para Gobierno Integrado. Antes de presentar las soluciones, un marco de evaluación fue introducido para cada categoría. Entre las soluciones organizacionales, el capítulo presentó tres Arquitecturas Empresariales – Zachman, SAGA y FEA; y tres Marcos de Interoperabilidad – eGIF, NF-eGIF y EIF. Entre las soluciones tecnológicas, el capítulo explicó tres soluciones de propósito general – BEA, WebShpere MQ y MSMQ; dos soluciones específicas de Gobierno Electrónico – IAMS y Hermes; una arquitectura de software – SOA, y un marco de desarrollo – JADE. Entre las soluciones fundacionales, el capítulo introdujo cuatro lenguajes de especificación formal – RSL, CSP, Reo y Pi-Calculus. Finalmente, todas las soluciones fueron evaluadas en cada categoría, según el marco de evaluación.

Los fundamentos de G-EEG fueron presentados en el Capítulo 4. Primero, los conceptos principales - miembros, canales, mensajes y extensiones, fueron introducidos y relacionados, resultando en un modelo conceptual aplicado para formalizar G-EEG y para llevar a cabo la implementación del prototipo de G-EEG. Segundo, el capítulo presentó la estructura de G-EEG en términos del framework de mensajería básico, un sistema de extensiones horizontales y verticales habilitado dinámicamente sobre el framework básico, y un marco de desarrollo para construir rigurosamente nuevas extensiones. Luego, el funcionamiento de G-EEG fue explicado, usando una simple notación gráfica para describir las estructuras de comunicación basadas en G-EEG, cómo tales estructuras pueden ser usadas para intercambiar mensajes, y cómo ciertos tipos de mensaje particulares pueden causar que las estructuras subyacentes cambien. Usando la notación introducida, varios escenarios para el intercambio de mensajes han sido explicados, desde el simple intercambio de mensajes a través de un canal plano, pasando por mensajes intercambiados a través de un canal con extensiones horizontales, hasta el intercambio de mensajes coordinado a través de varios canales con extensión vertical habilitada. Luego, el capítulo presenta un conjunto representativo de extensiones horizontales y verticales, describiendo para cada una de ellas las estructuras de comunicación creadas y el comportamiento de mensajería resultante. Finalmente, el capítulo refiere a la formalización de G-EEG en RSL, comenzando con XML como la estructura de datos subyacente para mensajes y variables; a través de una familia de lenguajes XML para representar varios tipos de expresiones sobre estructuras XML, cada uno con una sintaxis y una semántica definida; hasta la mensajería programable formalmente descripta con tales lenguajes XML y llevada a cabo por G-EEG. Finalmente, se presentó un análisis de la solución. El Apéndice A contiene las especificaciones formales.

El Capítulo 5 presentó el prototipo de software que implementa parcialmente G-EEG y el modelo introducido en el Capítulo 4. Intentando ilustrar la factibilidad de implementar G-EEG y aprender del desarrollo y del uso del prototipo, el prototipo fue construido a través de un proceso de desarrollo riguroso basado en UML. Comenzando con el modelo conceptual de G-EEG, los requerimientos funcionales y no funcionales fueron elaborados para capturar servicios específicos de mensajería y atributos de dependabilidad esperados, en base a los desafíos para Gobierno Integrado. El modelo de casos de uso fue creado como una base para implementar los requerimientos funcionales, con mapeo entre tales requerimientos y los caso de uso definidos. La arquitectura para el prototipo de G-EEG fue definida, incluyendo vistas estáticas y dinámicas, seguida por un diseño detallado, incluyendo diagramas de clases de diseño y ejemplos de diagramas de secuencia para capturar el comportamiento básico de mensajería. El prototipo implementa servicios básicos de mensajería – registrar y des-registrar miembros, crear y destruir canales, suscribir y des-suscribir miembros a y de canales, y enviar y recibir mensajes. También implementa tres extensiones horizontales auditoría, validación y transformación. El capítulo presenta detalles de las tecnologías usadas para implementar el prototipo, basadas en código abierto y estándares abiertos. Asimismo, se ilustró un posible escenario de entrega. El Apéndice C contiene artefactos de desarrollo, mientras que intefaces de programas de aplicación se presentan en el Apéndice D.

El Capítulo 6 presentó la evaluación de G-EEG en cuatro etapas. Primero, G-EEG fue evaluado con respecto a la declaración del problema en el Capítulo 1, explicando cómo soporta el establecimiento, la operación y evolución de redes organizacionales. Segundo, fue evaluado a través de desafíos tecnológicos a Gobierno Integrado introducidos en el Capítulo 2. Por cada desafío, los principales requerimientos técnicos fueron explicados, y una solución basada en G-EEG fue presentada, justificando como cumple con los requerimientos. Tercero, G-EEG fue evaluado con respecto al

caso de estudio del servicio de licencias. Luego de explicar el servicio y el proceso de negocios para producir y entregar tipos de servicios similares, una solución basada en G-EEG fue descripta para apoyar el proceso y, a través de ella, la entrega integrada del servicio de licencias. Cuarto, G-EEG fue evaluado con respecto a soluciones organizacionales, tecnológicas y fundacionales para Gobierno Integrado presentadas en el Capítulo 3. Basado en el marco de evaluación original definido para este propósito, G-EEG fue comparado con todas las soluciones presentadas, a través de las tres dimensiones.

7.2 Contribuciones

Esta tesis ha hecho las siguientes cinco contribuciones:

1) Introducción del concepto de Mensajería Programable – un paradigma para el intercambio automatizado de mensajes entre entidades colaborativas para responder a las diversas necesidades de comunicación de ambientes colaborativos complejos y dinámicos, como los ambientes que caracterizan a Gobierno Integrado.

2) Definición, formalización e implementación de Government-Enterprise Ecosystem Gateway (G-EEG) como una realización concreta de Mensajería Programable. Una plataforma de comunicación y coordinación de alto nivel, G-EEG permite a sus miembros, actuando en nombre de software, personas u organizaciones, llevar a cabo el intercambio automatizado de mensajes como base para la colaboración, intercambio de información o ejecución de procesos de negocios inter-organizacionales. Además de simple mensajería, G-EEG permite a sus miembros evolucionar las estructuras de comunicación subyacentes así como también permitir una rica funcionalidad de mensajería sobre tales estructuras para responder a sus complejas y cambiantes necesidades de comunicación. G-EEG fue diseñado para ser: minimalista – construido con el menor número posible de conceptos; extensible – construido sobre un marco básico simple pero proveyendo un mecanismo de extensión para proveer a sus miembros de mayor funcionalidad, dependiendo de sus necesidades; dinámico – capaz de responder a las cambiantes necesidades de comunicación y de computación de sus miembros con estructuras de comunicación y extensiones funcionales evolucionando a lo largo del tiempo; y confiable – construido sobre un modelo formal preciso para garantizar el comportamiento predecible y escalable con respecto a sus miembros.

   El valor y la originalidad de G-EEG han sido confirmados a través de cuatro tipos de evaluaciones:

   a) G-EEG fue evaluado con respecto a la declaración del problema formulado en el Capítulo 1 – una plataforma de comunicación y coordinación, con la teoría y el modelo subyacente, para facilitar el establecimiento, la operación y evolución de redes organizacionales capaces de entregar servicios públicos de manera integrada.
   b) G-EEG fue evaluado con respecto a diez desafíos tecnológicos que caracterizan a Gobierno Integrado y a Servicios Públicos Integrados: acceso único, procesos inter-organizacionales, monitoreo de cumplimiento de políticas, integración de aplicaciones, interoperabilidad sintáctica, interoperabilidad semántica, sub-contratación flexible, ecosistema dinámico, entrega por múltiples canales y requerimientos de dependabilidad.
   c) G-EEG fue evaluado con respecto al caso de estudio de la vida real basado en la entrega integrada de servicios de licencias, originado por Municipal Affairs Bureau del Gobierno de Macao RAE. G-EEG se mostró capaz de soportar las necesidades de comunicación y coordinación de este caso de estudio.
   d) Basado en un modelo formal e implementado en un prototipo a prueba de concepto, G-EEG muestra las características de soluciones organizacionales, tecnológicas y fundacionales. Como tal, fue comparado con soluciones organizacionales, tecnológicas y fundacionales presentadas en el Capítulo 3.

3) Un modelo formal para XML y computaciones basadas en XML. El modelo fue el resultado de los esfuerzos para formalizar G-EEG en RSL, conduciendo al modelo de estructuras XML básicas para representar nodos de elementos, elementos de texto y documentos XML, el conjunto de predicados y operaciones definido en tales estructuras, y la familia de lenguajes interrelacionados para escribir varios tipos de expresiones sobre las estructuras – Booleanas, números, textos, caminos, nodos, listas de nodos y árboles. Todos los lenguajes tienen su sintaxis y su semántica formalmente definidas. A su vez, el modelo fue validado al describir G-EEG, cuyos mensajes y variables están todos basados en estructuras XML, usando expresiones formuladas en los lenguajes.

4) Un exhaustivo estudio de conceptos y desafíos de Gobierno Integrado fue realizado. Los conceptos principales que definen el dominio de Gobierno Integrado – Agencias, Resultados, Capacidades, Recursos y Desafíos fueron identificados, elaborados con conceptos más específicos como Cliente, Servicio y Canal para Resultados, y relacionados (Capítulo 2). Por cada concepto, un modelo conceptual fue propuesto para llegar a comprender sus atributos y relaciones con otros conceptos en el dominio.

5) Un marco para evaluar soluciones organizacionales, tecnológicas y fundacionales para Gobierno Integrado (Capítulo 3). El marco comprende un conjunto de atributos generales aplicable a todas las soluciones, sin importar la categoría, y un conjunto de atributos específicos aplicable a las soluciones en categorías particulares. El marco fue validado a través de la aplicación al conjunto de siete soluciones organizacionales, ocho soluciones tecnológicas y cinco soluciones fundacionales para Gobierno Integrado, con G-EEG evaluado en las tres categorías.

Estas contribuciones se reflejaron en las siguientes publicaciones:

1) *Messaging Infrastructure for Electronic Government: Background, Rationale, Objectives* [EJF06] – El trabajo explica la necesidad de una infraestructura de software confiable que habilite el intercambio de información en soporte de Gobierno Electrónico, y presenta líneas para futuras investigaciones.
2) *Extensible Message Gateway for e-Government - Development Document* [EJ06a] – Este informe describe el desarrollo de un prototipo de software proveyendo servicios de mensajería a agencias, como se mostró en el Capítulo 5. El prototipo implementa los servicios básicos de G-EEG y tres ejemplos de extensiones horizontales.
3) *Infrastructure for Electronic Government - A Prototype for Messaging Services* [EJ06b] – Basado en [EJ06a], el trabajo describe la infraestructura técnica que soporta la comunicación entre agencias gubernamentales durante la entrega de servicios integrados – motivación, requerimientos funcionales, modelo conceptual, vistas estáticas y dinámicas de la arquitectura, y posibles escenarios para la entrega de servicios de mensajería.
4) *Government-Enterprise Ecosystem Gateway (G-EEG) for Seamless e-Government* [EJ07a] – El trabajo presenta los desafíos para el desarrollo de Gobierno Integrado y propone una solución técnica – Government-Enterprise Ecosystem Gateway (G-EEG) para enfrentarlos, con componentes, comportamiento y un número de extensiones de G-EEG definidas. El trabajo fue nominado para recibir el premio al mejor trabajo.
5) *Building a Dependable Messaging Infrastructure for Electronic Government* [EJ07b] – El trabajo presenta el desarrollo de una infraestructura de mensajería confiable para Gobierno Electrónico. Describe los requerimientos de dependabilidad para aplicaciones de gobierno en general y explica cómo la infraestructura de G-EEG enfrenta tales requerimientos en los niveles de diseño, desarrollo y aplicación.
6) *Programmable Messaging for Electronic Government – Building a Foundation* [EJ07c] – El trabajo presenta un fragmento de un modelo formal para G-EEG, enfocado particularmente en los servicio de mensajería básicos.

El prototipo de G-EEG del Capítulo 5 fue publicado y demostrado como sigue:

7) *Rapid Development of Electronic Public Services – A Case Study in Electronic Licensing Services* [OJE07d] – La demostración de software del Servicio Electrónico de Licencias (e-Licensing) desarrollado sobre el prototipo de la Infraestructura de Software para SPE. La demostración muestra el proceso de desarrollo de SPE sobre la infraestructura, usando el prototipo G-EEG del Capítulo 5 como el servicio de mensajería subyacente.

A su vez, los resultados obtenidos fueron usados en las siguientes publicaciones:

1) *Infrastructure Support for e-Government - An Overview* [OJOE06] – El reporte presenta una visión general de la infraestructura de software para Gobierno Electrónico, desarrollada para el Gobierno de Macao RAE. Describe los componentes de la infraestructura, y provee pautas de cómo los elementos de la infraestructura pueden ser usados para construir Servicios Públicos Electrónicos. El prototipo G-EEG (Capítulo 5) es un componente de esta arquitectura.
2) *Rapid Development of Electronic Public Services – Software Infrastructure and Software Process* [OJE07a] – El trabajo presenta una infraestructura de software para apoyar el rápido desarrollo de Servicios Públicos Electrónicos (SPE). La infraestructura provee un marco de diseño y componentes y servicios de infraestructura de tiempo de ejecución para ayudar al diseño, a la implementación y a la ejecución de SPE. También describe un proceso de software para el desarrollo de SPE en base a tal infraestructura. El prototipo G-EEG (Capítulo 5) es usado como un servicio de infraestructura.

3) *A Composite Domain Framework for Developing Electronic Public Services* [OJE07b] – El trabajo presenta un framework de dominio compuesto para construir las partes de Front Office y Back Office de un SPE. El framework soporta un conjunto de requerimientos de dominio obtenidos a través de un análisis detallado de más de 30 servicios públicos concretos. Los servicios de mensajería, descritos en la tesis, son parte del Back Office de SPE.
4) *Domain Models and Enterprise Application Framework for Developing Electronic Public Services* [OJE07c] – El trabajo presenta un modelo de dominio genérico para apuntalar el desarrollo de SPE – desde modelos conceptuales, a través de requerimientos y arquitectura, hasta la implementación. Los servicios de mensajería [EJ06a] son parte de esos modelos.
5) *Extensible Message Gateway, Development Report* [DEJ08] – El informe presenta el desarrollo del producto de software de calidad de producción para el Gobierno de Macao RAE en base al prototipo documentado en [EJ06a].
6) *E-Queuing Service, Development Report* [EDJ09a] – El reporte presenta el desarrollo de un servicio electrónico de colas de acceso único, automatizando la administracipon de colas de clientes en agencias gubernamentales. La comunicación entre el portal de acceso único y varios proveedores de servicios se realiza usando G-EEG y soportado por [DEJ08].
7) *E-Appointment Service, Development Report* [EDJ09b] – Este informe presenta un servicio electrónico de citas, que usa el servicio de mensajería de G-EEG y es técnicamente soportado por [DEJ08].

El seminario *Government-Enterprise Ecosystem Gateway – A Communication Framework for Electronic Government* fue preparado para introducir el concepto de Gobierno Integrado, para presentar algunos desafíos para su desarrollo, y para proponer a G-EEG como una posible solución a tales desafíos. Basado en los contenidos de los Capítulos 4 y 5, el seminario incluyó una demostración del prototipo G-EEG. El seminario fue presentado en: UNU-IIST, Macao, en julio 2006; Hawaii International Conference on System, enero 2007; Universidad de Hong Kong, enero 2007; y Universidad de Albany, Estados Unidos, en marzo 2008.

Los resultados obtenidos por esta tesis fueron usados para desarrollar los siguientes materiales de capacitación: (1) *UNU-CNSA School on Electronic Governance – How to Implement Electronic Governance* [EDO09a] – El curso fue parte de una escuela de capacitación dictada la autora y otros académicos de UNU a los representantes de China National School of Administration; y (2) *E-Macao Strategy School – Implementation Module* [EDOD09b] – El curso fue parte de una capacitación brindada por la autora y otros conferencistas de UNU a los oficiales del Gobierno de Macao RAE.

El trabajo presentado en esta tesis también fue usado como base para la tesis de MSc – Interoperabilidad Semántica para el *Middleware de Mensajería Programable – Extensiones de Mediación, Validación y Descubrimiento* [San08]. Basado en el prototipo y la especificación de G-EEG, la tesis especificó y desarrolló tres extensiones para apoyar la interoperabilidad semántica: Validación – validar mensajes expresados en diferentes sintaxis según una cierta ontología; Mediación – mediar los contenidos de los mensajes entre diferentes ontologías; y Descubrimiento – descubrir los recursos web que satisfacen una cierta consulta semántica, basado en una ontología y un razonador.

Adicionalmente, el trabajo en esta tesis asentó las bases para dos proyectos de investigación. El primer proyecto es "*Un Marco para el Gobierno Electrónico a través de Servicios Web en la Web Semántica".* El objetivo de este proyecto es desarrollar un marco que mejore los servicios ofrecidos por G-EEG a través de agentes inteligentes que sean capaces de administrar protocolos y formatos de la Web Semántica. El proyecto se lleva a cabo en el Departamento de Ciencias e Ingeniería de la Computación, UNS, Argentina. El segundo proyecto es "G*overnment Information Sharing".* El objetivo del proyecto es establecer las necesidades de intercambio de información de las agencias gubernamentales de Macao, para prescribir estrategias de intercambio de información basadas en la investigación de mejores prácticas, y para proponer la infraestructura técnica y organizacional para implementar la estrategia. Un objetivo del proyecto es entregar una infraestructura técnica, basada en G-EEG, en agencias piloto para permitir el intercambio de información entre ellas. El proyecto se lleva a cabo en el UNU-IIST Center for Electronic Governance.

7.3 Trabajo Futuro

Como se mencionó en la Sección 7.2, dos proyectos de investigación han sido ya iniciados basados en el trabajo llevado a cabo en esta tesis, mostrando algunas direcciones para futuros trabajos: (1) Un *Marco para el Gobierno Electrónico a través de Servicios Web en la Web Semántica* y (2) G*overnment Information Sharing*. El objetivo del primero, enfocado en investigación técnica, es explorar la aplicación de agentes inteligentes y tecnologías de la Web Semántica en G-EEG. El segundo incluye investigación y desarrollo a través de cuestiones técnicas y organizacionales,

con el objetivo de desarrollar estrategias para el intercambio de información y de infraestructuras técnicas y organizacionales que puedan apoyar la implementación de tales estrategias.

A partir de los resultados obtenidos en esta tesis es posible definir tres áreas de trabajos futuros: (1) la aplicación de métodos formales al paradigma de Mensajería Programable, (2) temas específicos de Ingeniería de Software aplicados a la Mensajería Programable, y (3) la futura utilización de los servicios de Mensajería Programable. Las posibles líneas de investigación de cada una de estas áreas se explican a continuación.

Como aplicación de los métodos formales de desarrollo al paradigma de Mensajería Programable se definen las siguientes líneas: especificación de la totalidad de las extensiones propuestas para validar y mejorar el método de desarrollo propuesto por G-EEG-DEVELOP; verificación de las propiedades de comportamiento de los protocolos de mensajería usados por G-EEG-CORE; uso de especificaciones combinadas basadas en estado y en acciones para describir a G-EEG en diferentes niveles de abstracciones; especificación y verificación de las propiedades de comportamiento para la composición de extensiones; aplicación de distintos lenguajes de especificación formal, como CSP, para la especificación de los servicios de mensajería y la comparación con las especificaciones en RSL; verificación de la tolerancia a fallas de los servicios de mensajería.

Cuatro líneas de investigación se definen dentro del área de temas específicos de Ingeniería de Software aplicados a la Mensajería Programable: la definición de patrones de diseño de alto nivel para los servicios de mensajería básicos y extendidos; la abstracción para utilizar los servicios de Mensajería Programable como procesos primitivos en arquitecturas de más alto nivel, como las del tipo Enterprise Service Bus; definición y aplicación de técnicas de empaquetamiento para el reuso de los servicios de Mensajería Programable en diferentes dominios de aplicación; y la aplicación de procedimientos y herramientas de calidad que permitan certificar los servicios de Mensajería Programable que puedan proveerse bajo entornos de comunidades de práctica.

Dentro del área de la futura utilización de los servicios de Mensajería Programable se identifican las siguientes líneas de investigación: desarrollo de las restantes extensiones descriptas en la tesis; construcción de un repositorio de extensiones para G-EEG-EXTEND; validación del entorno de desarrollo a través de nuevas extensiones desarrolladas, como por ejemplo, extensiones para incluir firma digital o una estampilla de fecha y hora de envío, o una extensión que permita la implementación de mensajes sincrónicos, etc; enriquecer la funcionalidad actual de alguna de las extensiones, como por ejemplo, enriquecer el lenguaje para especificar los procesos de negocios usado por la extensión de Orden a fin que permita controlar otras características del proceso; y empaquetar las especificaciones y el código de G-EEG para futuros desarrollos por comunidades de práctica.

Bibliografía

[AG02] ANDRULIS, J. and GRESHAM, M., *Using Hybrid Funding Strategies to Support the State of Arizona*. IBM, 2002, p.11, available at http://www-935.ibm.com/services/ (visited July 29, 2008).

[AIHW08] AUSTRALIAN INSTITUTE OF HEALTH AND WELFARE (AIHW), *Recurrent Expenditure*. AIHW, 2008, Metadata Online Registry (METeOR) (Ed.), available at http://meteor.aihw.gov.au/content/index. phtml/itemId/269132 (visited August 1, 2008).

[AL93] ABADI, M. and LAMPORT, L., *Composing Specifications*. ACM Transactions of Programming Languages and Systems (TOPLAS), vol. 15, 1993, pp. 73-132, ISSN 0164-0925, available at http://citeseerx.ist.psu.edu/ viewdoc/summary?doi=10.1.1.41.5272 (visited August 9, 2008).

[AM02] ARBAB, F. and MAVADDAT, F., *Coordination through Channel Composition*. In Coordination Languages and Models: Proceedings of Coordination 2002, Talcott, F. and Arbab C. (ed.), Lecture Notes in Computer Sciences, Springer Verlag, vol. 2315, 2002, pp. 21-38.

[Apa09] THE APACHE XML PROJECT, XMLBeans, http://xmlbeans.apache.org, visited 14 February 2009.

[AR02] ARBAB, F. and RUTTEN, J.J.M.M., *A Co-inductive Calculus of Component Connectors*. In Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT'02), 24-27 September 2002, Frauenchiemsee, Germany, Springer-Verlag, 2002, pp. 34-55, available at http://citeseerx.ist.psu.edu /viewdoc/summary?doi=10.1.1.19.8617 (visited October 26, 2008).

[Arb02] ARBAB, F., *A Channel-based Coordination Model for Component Composition*. Software ENginneering (SEN). s.l. : Centrum voor Wiskunde en Informatica, Technical Report SEN-R0203, 2002, pp. 37, ISBN 1386-369X, available at ftp.cwi.nl/CWIreports/SEN/SEN-R0203.pdf (visited October 7, 2008).

[Ari02] ARIZONA GOVERNMENT, *Arizona@Your Service*. Government of the State of Arizona, United States of America, 2002, available at http://www.az.gov (visited July 29, 2008).

[Aus06] AUSTRALIA GOVERNMENT, *Introductory Guide to Public Private Partnership - FMG 16*. Department of Finance and Regulation, Government of Australia, 2006, ISSN 9757595-7, available at http://www.finance.gov.au/publications/fmg-series/16-intro-guide-to-public-private-partnerships.html (visited July 29, 2008).

[Bar01] BARZELAY, M., *The New Public Management: Improving Research and Policy Dialogue*. eBooks.com, The Digital Bookstore, 2001, ISSN 1597347744.

[BCDF06] BALDONI, R., CIMPIAN, E., DIMITROV, M., FRAGIS, T., DE GIACOMO, G., KIRYAKOV, A., LOUTAS, N., MECCELA, A., MOCAN, S., NAZIR, S., v.OVEREEM, A., PERISTERAS, V., RUSSO, R., TARABANIS, K., TRIANTAFYLLOU, E., VITVAR, T., WATERFELD, W., WINKLER, K., WITTERS, J., *SemanticGov, State of the Art Report* (*WP01 – Overall Conceptual Analysis – D1.2 – State of the Art Report)*. Center for Research and Technology Hellas (CERTH), 2006, pp.143, available at http://www.enterprise-arc.info/Images/EIF/semanticgov_ITI_D012_Final_State-of-the-art_ Report.pdf (visited July 29, 2008).

[BCPR07] BELLIFEMINE, F., CAIRE, G., POGGI, A., RIMASSA, G. *JADE - A White Paper*. Tilab, 2007, pp. 19, available at http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf (visited August 19, 2009).

[BD00] BAUM and DI MAIO, *Gartner's Four Phases of E-Government Model*. Gartner Group, 2000.

[BD93] BOLIGNANO, D. and DEBBABI, M., *A Denotational Model for the Integration of Concurrent, Functional, and Imperative Programming*. In Proceedings of the Fifth International Conference on Computing and Information, ACM, 1993, pp. 244-250, ISBN 0-8186-4212-2.

[BEA00] BEA SYSTEMS, INC, *BEA MessageQ - Introduction to Message Queuing*. BEA Systems, Inc., 2000, pp. 118, available at http://e-docs.bea.com/tuxedo/msgq/pdf/mqintro.pdf (visited October 7, 2008).

[BEA08] BEA SYSTEMS, *Introduction to FML Programming*. BEA Systems, 2008, available at http://edocs.bea.com/ tuxedo/tux90/fml/fml01.htm (visited August 14, 2008).

[Bel08] BELGIAN FEDERAL GOVERNMENT, *Belgium Social Security Portal*. Belgian Federal Government, available at http://socialsecurity.be (visited October 25, 2008).

[BGN04] BURBECK, K., GARPE, D., NADJM-TEHRANI, S., *Scale-up and Performance Studies of Three Agent Platforms*. Department of Computer and Information Science, LinkOpojng University, Sweden, 2004, pp. 7, available at http://www.ida.liu.se/~rtslab/publications/2004/Burbeck04platform.pdf (visited August 20, 2008).

[BHL01] BERNERS-LEE, T., HENDLER, J. and LASSILA, O., *The Semantic Web*. Scientific American, May 17, 2001.

[BM02] BUSCEMI, M., and MONTANARI, U., *Pi-Calculus Early Observational Equivalence: A First Order Coalgebraic Model*. University of Pisa, Technical Report: TR-02-14, 2002, available at http://portal.acm.org/citation. cfm?id=898072&coll=GUIDE&dl=GUIDE (visited October 26, 2008).

[BSAR06] BAIER, C., SIRJANIS, M., ARBAB, F., and RUTTEN, J.J.M.M., *Modeling Component Connectors in Reo by Constraint Automata*. Special issue on Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03), Elsevier North-Holland, Inc., vol. 61, issue 2, July 2006, pp. 75-113, available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.7419 (visited October 26, 2008).

[Cab08a] CABINET OFFICE, *Customer Service Excellence*. Government of United Kingdom, February 2008, available at http://www.cse.cabinetoffice.gov.uk/homeCSE.do? (visited June 21, 2008).

[Cab08b] CABINET OFFICE, *Leadership for Reform and Customer Focus*. Government of United Kingdom, June 2008, available at http://www.cabinetoffice.gov.uk/workforcematters/leadership/reform.aspx (visited June 20, 2008).

[Cai04] CAINE, A., *E-Government: Legal and Administrative Obstacles to Sharing Data held by Australian Government Agencies – Discussion Paper Number 13*. Australian Government Information Management Office (AGIMO), Government of Australia, 2004, pp.12, available at www.finance.gov.au/publications/future-challenges-for-egovernment/docs/AGIMO-FC-no13.pdf (visited August 1, 2008).

[Can07] CANADA GOVERNMENT, *We've Gone Mobile!*. Canada Government Portal, December 20, 2007, available at http://www.canada.gc.ca/mobile/wireless-eng.html (visited June 21, 2008).

[Can08a] THE CANADIAN COUNCIL FOR PUBLIC-PRIVATE PARTNERSHIP (CCPPP), *About PPP - Definition and Models*. CCPPP, 2008, available at http://www.pppcouncil.ca/aboutPPP_definition.asp (visited July 29, 2008).

[Can08b] THE CANADIAN COUNCIL FOR PUBLIC-PRIVATE PARTNERSHIP (CCPPP), *About PPP – Why Public-Private Partnerships?* CCPPP, 2008, available at http://www.pppcouncil.ca/aboutPPP_why.asp (visited July 29, 2008).

[CCL03] CURRY, E., CHAMBERS, D., and LYONS, G., *A JMS Message Transport Protocol for the JADE Platform*. In Proceedings of IEEE/WIC International Conference of Intelligent Agent Technology (IAT'03), IEEE, 2003, pp. 593-603, available at http://doi.ieeecomputersociety.org/10.1109/IAT.2003.1241153 (visited August 20, 2008).

[CEC07] CENTER FOR E-COMMERCE INFRASTRUCTURE DEVELOPMENT (CECID), *Hermes Messaging Gateway. CECID - Hermes*. CECID, The Univeristy of Hong Kong, Hong Kong SAR, China, 2007, available at http://www.cecid.hku.hk/hermes.php (visited October 7, 2008).

[CEC08] CENTER FOR e-COMMERCE INFRASTRUCTURE DEVELOPMENT (CECID), *CECID – About Us*. CECID, 2008, available at http://www.cecid.hku.hk/overview.php (visited August 20, 2008).

[CGI98] CGI GROUP INC., *CGI Case Studies – Virginia Department of Taxation (VATAX)*. CGI – Outsourcing, System Integration and Consulting, CGI, 1998, available at http://www.cgi.com/web/en/ library/case_studies/71155.htm (visited July 29, 2008).

[CK04] CIMANDER, R. and KUBICEK, H., eGovernment *Interoperability at Local and Regional Level, Good Practice Case, Social Security Benefits for Citizens in Belgium*. Crossroad Bank for Social Security

and National Office for Social Security, eGovernment Unit DG Information Society and Media European Commission, Brussels, Belgium, 2004, pp. 19, available at http://unpan1.un.org/intradoc/groups/public/documents/other/ unpan022035.pdf (visited October 25, 2008).

[Col03] COLLINS, P., *Implementing a Messaging Infrastructure for Intra-Governmental Cooperation.* 3rd European Conference on E-Government (EGOV 2003), Prague, Czech Republic, September 1-5, 2003, pp. 69-80.

[CU81] COLUMBIA UNIVERSITY, *The US ASCII Character Set*. The Kermit Project, 1981, available at http://www.columbia.edu/kermit/ascii.html (visited August 12, 2008).

[CUI06] CERTH/ITI, UNIROMA, IBERMATICA, ALTEC, NATH, ICE, UNTC, *EU.Publi.com Project*. 2006, available at http://unpan1.un.org/intradoc/groups/public/documents/UNTC/UNPAN007037.pdf (visited July 27, 2008).

[DAML07] DARPA AGENT MARKUP LANGUAGE (DAML), *OWL-based Web Services Ontology (OWL-S; formerly DAML-S)*. DARPA Agent Markup Language Program DAML, 2007, available at http://www.daml.org/services/owl-s/ (visited October 25, 2008).

[DB05] DAVIES, S. and BROADHURST, P., *WebSphere MQ V6 Fundamentals*. IBM RedBooks, 2005, pp. 446, available at http://www.redbooks.ibm.com/redpieces/pdfs/sq246878.pdf (visited August 14, 2008).

[DCGP05] DAVIES, S., COWEN, L., GIDDINGS, C., PARKER, H., *WebSphere Message Broker Basics*. IBM RedBooks, 2005, available at http://www.redbooks.ibm.com/abstracts/sg247137.html (visited August 14, 2008).

[DCMI08] DOUBLIN CORE METADATA INITIATIVE (DCMI), *Dublin Core Metadata Element Set Version 1.1*. DCMI, January 14, 2008, available at http://dublincore.org/documents/dces/#ISO15836 (visited July 29, 2008).

[DD02] Denhardt, J. V., and Denhardt, R. B., *The New Public Service: Serving not Steering*. M.E. Sharpe, 2002, ISBN 0765608464.

[DEJ08] DOUWE, V., ESTEVEZ, E., JANOWSKI, T. *Extensible Message Gateway, Development Report*, Software Infrastructure for Electronic Government Project, e-Macao Program, April 2008.

[DGJM02] DANG VAN, H., GEORGE, C., JANOWSKI, T., MOORE, R., *Specification Case Studies in RAISE*. Springer, 2002, ISSN 1-85233-359-6., available at http://www.iist.unu.edu/RAISE_Case_Studies/ (visited August 10, 2008).

[Dic98] DICKMAN, A., *Designing Applications with MSMQ,* Addison-Wesley, 1998.

[Dru05] DRUMMOND GROUP INC., *MIME-Bases Secure Peer-to-Peer Business Data Interchange Using HTTP, Applicability Statement 2 (AS2)*. Internet Engineering Task Force, 2005, pp. 33, available at http://www.ietf.org/rfc/ rfc4130.txt (visited August 20, 2008).

[Dub06] DUBOIS, P., *MySQL Cookbook*, O'Reilly, November 2006.

[EC05] EUROPEAN COMMISSION, *e-Procurement Optimised System for the Healthcare Marketplace, e-Pos Project*. European Commission, 2005, available at http://www.eposproject.com/index.php (visited July 29, 2008).

[EDJ07] ESTEVEZ, E., DOUWE, V., JANOWSKI, T. *Extensible Message Gateway, User Manual*, Software Infrastructure for Electronic Government Project, e-Macao Program, July 2008.

[EDJ09a] ESTEVEZ, E., DOUWE, V., JANOWSKI, T. *E-Queuing Service, Development Report*, Software Infrastructure for Electronic Government Project, e-Macao Program, February 2009.

[EDJ09b] ESTEVEZ, E., DOUWE, V., JANOWSKI, T. *E-Appointment Service, Development Report*, Software Infrastructure for Electronic Government Project, e-Macao Program, February 2009.

[EDJ09c] ESTEVEZ, E., DOUWE, V., JANOWSKI, T. *Extensible Messaging Gateway – Enhanced Services*, Technical Report, Software Infrastructure for Electronic Government Project, e-Macao Program, February 2009.

[EDO09b] ESTEVEZ, E., DOUWE, V., OJO, A., DZUSUPOVA, Z., and JANOWSKI, T. *UNU-CNSA School on Electronic Governance – How to Implement Electronic Governance,* UNU – China National School of Administration, UNU-IIST Macao, 19-21 January 2009, available http://www.egov.iist.unu.edu/index.php?/cegov/Events/Events-2009/UNU-CNSA-School-and-Workshop-on-Electronic-Government-in-China, visited April 26, 2009.

[EDO09b] ESTEVEZ, E., DOUWE, V., OJO, A., DZUSUPOVA, Z., and JANOWSKI, T. *e-Macao Strategy School – Implementation Module,* e-Macao School, UNU-IIST Macao, 12-15 January 2009, available at http://www.egov.iist.unu.edu/index.php?/cegov/Events/Events-2009/e-Macao-Strategy-School-How-can-Macao-utilize-ICT-for-Public-Sector-Modernization visited April 26, 2009.

[EIS07] EUROPE'S INFORMATION SOCIETY, *Thematic Portal - Project Factsheets, eTen, ePOS, Technical Details*. Europe's Information Society, 2007, available at http://ec.europa.eu/information_society/activities/eten/cf/opdb/cf/project/index.cfm?mode=desc&project_ref=ETEN-517390 (visited July 29, 2008).

[EJ06a] ESTEVEZ, E. and JANOWSKI, T., *Extensible Message Gateway for e-Government - Development Document*. e-Macao Project, Technical Report No. 11, August 2006. Available at http://www.emacao.gov.mo/documents/06/report11.pdf, visited 26 April 2009.

[EJ06b] ESTEVEZ, E. and JANOWSKI, T., *Infrastructure for Electronic Government - A Prototype for Messaging Services*. In proceedings of the 1st Iberoamerican Congress on e-Government, Santiago de Chile, October 2006.

[EJ07a] ESTEVEZ, E. and JANOWSKI, T., *Government-Enterprise Ecosystem Gateway (G-EEG) for Seamless e-Government*. In Proceedings of The 40th Hawaii International Conference on System Sciences (HICSS 2007), Waikoloa, Big Island, Hawaii, United States of America, January 2007, IEEE Computer Society Press, ISBN 978-0-7695-2755-0, pp. 101-110, available at http://portal.acm.org/citation.cfm?id=1255848 (visited October 25, 2008).

[EJ07b] ESTEVEZ, E. and JANOWSKI, T., *Building a Dependable Messaging Infrastructure for Electronic Government*. In proceedings of the 2nd International Workshop "Dependability and Security in e-Governmet (DeSeGov 2007), part of the International Conference on Availability, Reliability and Security, Vienna, Austria, April 2007, IEEE Computer Society, IEEE, ISBN 0-7695-27775-2, ISBN 978-0-7695-27775-8, pp. 948—955.

[EJ07c] ESTEVEZ, E. and JANOWSKI, T., *Programmable Messaging for Electronic Government – Building a Foundation*. In proceedings of the Festchrift Simposium for Dines Bjorner and Zhou Chaochen in Macao, September 2007. Lecture Notes in Computer Science, Springer.

[EJF06] ESTEVEZ, E., JANOWSKI, T., and FILLOTTRANI, P., *Messaging Infrastructure for Electronic Government: Background, Rationale, Objectives*. In proceedings of the 8th Workshop of Researchers in Computer Sciences (WICC), Moron, Argentina, June 2006, ISBN 950-9474-34-7.

[EJOK07] ESTEVEZ, E., JANOWSKI, T., OJO, A. and KHAN, I., *Coordination Offices for E-Government*. United Nations University – International Institute for Software Technology, Technical Report 363, April 2007, pp. 18, available at http://iist.unu.edu (visited October 25, 2008).

[Env09] ENVOY TECHNOLOGIES, Envoy Connect, available at http://www4.envoytech.com/envoy/ navigation.do? iid=NET&page=Products/NET_interoperability.html, visited May, 4 2009.

[Erl05] ERL, T., *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Pearson Education, 2005, ISBN 0-13-185858-0.

[Est09] ESTEVEZ, E., *Programmable Messaging for Electronic Government, PhD Thesis,* Center for Electronic Governance at United Nations University, International Institute for Software Technology, Macao SAR, China, August 2009.

[EU08] ePRACTICE.EU, *e-Government Factsheet – Estonia – National Infrastructure*. ePractice.eu, April 10, 2008, available at http://www.epractice.eu/document/3332 (visited October 25, 2008).

[Fab06] FABRICI, D., *Defining Ontologies in a Multi Agent Scenario Using the JADE Framework*. In Proceedings of the 4th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence,

Herlany, Slovakia, 2006, available at www.bmf.hu/conferences/sami2006/Fabrici.pdf (visited August 20, 2008).

[FEA05] THE FEDERAL ENTERPRISE ARCHITECTURE (FEA) PROGRAM MANAGEMENT OFFICE, GOVERNMENT OF UNITED STATES OF AMERICA, *The Data Reference Model*, Version 2.0. FEA Program Management Office, 2005, pp. 104, available at http://www.whitehouse.gov/omb/egov/documents/DRM_2_0_Final.pdf (visited August 11, 2008).

[FEA07] THE FEDERAL ENTERPRISE ARCHITECTURE (FEA) PROGRAM MANAGEMENT OFFICE, GOVERNMENT OF UNITED STATES OF AMERICA. *FEA Consolidated Reference Model Document Version 2.3*. Executive Office of the President of the United States, 2007, pp. 90, available at http://www.whitehouse.gov/omb/egov/documents/FEA_CRM_v23_Final_Oct_2007.pdf (visited October 7, 2008).

[FGW06] FEDOROWICZ, J., GOGAN, J.L. and WILLIAMS, C. B., *The e-Government Collaboration Challenge: Lessons from Five Case Studies*. IBM Center for the Business of Government, Network and Partnership Series, 2006, pp. 54, available at http://www.businessofgovernment.org/pdfs/FedorowiczReport.pdf (visited July 28, 2008).

[FIPA02] THE FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS (FIPA), *FIPA ACL Message Structure Specification*. FIPA 2002, available at http://www.fipa.org/specs/fipa00061/ (visited August 19, 2008).

[FIPA08] THE FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS (FIPA), *About FIPA*, FIPA 2008, available at http://www.fipa.org (visited August 19, 2008).

[FML03] FIELD, T., MULLER, E. and LAW, E., *The e-Government Imperative*. Organization for Economic Co-operation and Development, ISBN 92-64-10117-9, Paris, France, 2003, pp. 199, available at http:213.253.134.43/oecd/pdfs/browseit/4203071E.PDF (visited October 7, 2008).

[GAO06] GENERAL ACCOUNTABILITY OFFICE (GAO), GOVERNMENT OF UNITED STATES OF AMERICA, *Enterprise Architecture - Leadership Remains Key to Establishing and Leveraging Architectures for Organizational Transformation*. GAO, Government of the United States of America. 2006. pp. 182, available at http://www.gao.gov/ new.items/d06831.pdf (visited October 7, 2008).

[GGH05] GARDINER, P., GOLDSMITH, M., HULANCE, J., JACKSON, D., ROSCOE, B., SCATTERGOOD, B. and ARMSTRONG, P., *FDR2 Manual*. Formal Systems (Europe) Ltd., 2005, available at http://www.fsel.com/ documentation/fdr2/html/fdr2manual_40.html#SEC40 (visited August 20, 2008).

[Gru92] GRUBER, T.R., *A Translation Approach to Portable Ontology Specifications*. Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, California, United States, 1992, pp. 1-24, Technical Report KSL 92-71, available at http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html (visited July 27, 2008).

[Gui04] GUIJARRO, L., *Analysis of the Interoperability Frameworks in e-Government Initiatives*. In Electronic Government, Proceedings of the Third International Conference EGOV 2004, Zaragoza, Spain, August/September 2004, Roland Traünmuller (Ed.), Lecture Notes in Computer Science, 3183, pp. 36-39.

[HC06] HAMMER, M. and CHAMPY, J., *Reengineering the Corporation: A Manifesto for Business Revolution*. New York, United States of America, Harper Business Books, ISBN 0-06-662112-7.

[HE05] HULTGREN, G., and ERIKSSON, O. (Eds), *The Concept of e-Service from a Social Interaction Perspective*. University of Limerick, Limerick, Ireland, in Proceedings of the Third International Conference on Action in Language, ALOIS 2005, Organizations and Information Systems, pp. 65-80, 2005, available at http://www.alois2005.ul.ie/ALOIS_2005_Proceedings_web.pdf (visited June 06, 2008).

[Hib09] HIBERNATE, *Relational Persistence for Java and .Net*, http://www.hibernate.org, visited 14 February 2009.

[Hoa78] HOARE, T., *Communicating Sequential Processes*. Communications of the ACM, ACM, 1978, vol. 21, pp. 666-677, available at http://portal.acm.org/citation.cfm?id=359585 (visited August 24, 2008).

[Hoh01] HOHBACH, B., *Accenture Newsroom: Accenture Signs-In-Savings Contract with the U.S. Department of Education*. Consulting, Technology and Outsourcing Services at Accenture, Accenture, December 21, 2001, available at http://accenture.tekgroup.com/article_display.cfm?article_id =3816 (visited July 29, 2008).

[HW04] HOHPE. G. and WOOLF, B., *Enterprise Integration Patterns – Designing, Building and Deploying Messaging Solutions*, Addison Wesley, 2004, ISBN: 0321200683.

[IACM04] INSTITUTO PARA OS ASSUNTOS CIVICOS E MUNICIPAIS (IACM), *One-Stop Licensing Service for Food and Beverage Establishments, Guidelines for Applications.* IACM, The Government of Macao SAR, China, 2004.

[IBM08a] IBM, *Websphere Software*, IBM, available at http://www-306.ibm.com/software/websphere/ (visited October 7, 2008).

[IBM08b] IBM CORPORATION, *WebSphere MQ Queue Managers*. IBM, 2008, available at http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.express.doc/concepts/cjc0011_.html (visited August 14, 2008).

[IBM08c] IBM CORPORATION, *List of Supported Software for WebSphere MQ V6.0*. IBM, 2008, available at http://www-1.ibm.com/support/docview.wss?rs=171&uid=swg27006402 (visited August 14, 2008).

[IDA04] INTERCHANGE of DATA BETWEEN ORGANIZATIONS (IDA), *Multi-Channel Delivery of e-Government Services*. IDA, 2004, available at http://www.cisco.com/web/DE/pdfs/publicsector/ida_07_04.pdf (visited June 21, 2008).

[IDA04] INTEROPERABLE DELIVERY OF EUROPEAN E-GOVERNMENT SERVICES TO PUBLIC ADMINISTRATIONS, BUSINESS AND CITIZENS (IDABC), *European Interoperability Framework for Pan-European e-Government Services*. IDABC, EIF - European Interoperability Framework. Luxembourg, Belgium, European Communities, 2004. ISBN 92-894-8389-X, available at http://europa.eu/int/idabc/en/document/3761 (visited October 7, 2008).

[IDA05] INTEROPERABLE DELIVERY OF EUROPEAN E-GOVERNMENT SERVICES TO PUBLIC ADMINISTRATIONS, BUSINESSES AND CITIZENS (IDABC), IDABC – Web site. IDABC, 2005, available at http://ec.europa.eu/idabc/ (visited August 9, 2008).

[IEEE98] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE), *IEEE Posix*. IEEE Standards Association, 1998, available at http://standards.ieee.org/regauth/posix/ visited (July 29, 2008).

[IETF98] INTERNET ENGINEERING TASK FORCE (IETF), *UTF-8, A Transformation Format of ISO 10646*. IETF, 1998, available at http://www.ietf.org/rfc/rfc2279.txt (visited August 12, 2008).

[JOE05] JANOWSKI, T., OJO, A., and ESTEVEZ, E. *The State of Electronic Government in Macao, Volume 1: Survey*. Macao e-Government Project (e-Macao), April 2005, available at http://www.emaco.gov.mo/documents/01/ report1.pdf (visited October 7, 2008).

[Jon04] JONES, N., *Service Design and Delivery Guide*. The Cabinet Office, Government of United Kingdom, 2004, available at http://www.epractice.eu/resource/523 (visited June 21, 2008).

[KBST06] KBST UNIT AND AG AND FRAUNHOFER-INSTITUT FÜR SOFTWARE- UND SYSTEMTECHNIK (IIST), *Standards and Applications for e-Government Applications (SAGA), Version 3.0.* Federal Government's Coordination and Advisory Board for IT in the Administration (KBSt unit) , Federal Ministry of the Interior of Government of Germany, Berlin, Germany, Federal Ministry of Interior, Germany, 2006, pp. 125, Technical Report, available at http://www.kbst.bund.de/cln_028/nn_837392/SharedDocs/Anlagenkbst/Saga/saga__3__0__en.html__nn n=true (visited October 7, 2008).

[KGT95] KETTINGER, W. J., GUHA, S. and TENG, J. T.C., *The Process Reengineering Life Cycle Methodology: A Case Study*. Kettinger, in Business Process Change: Reegineering Concepts, Methods and Technologies, V.Grover and W.J. (Eds), Idea Publishing Group, pp. 210-244, 1995.

[Kot96] KOTTER, J. P. *Leading Change*. Library of Congress Cataloging-in-Publication Data, 1996. ISBN 0-87584-747-1.

[LACS06] LAFFER, A., ARRISON, S., COORS, A., STEIN, G., VASQUEZ, V., WINEGARDEN, W., *Deregulation Lessons for Telecommunications in California from the Airline and Natural Gas Industries*. Pacific Research Institute, San Francisco, United States of America, 2006, pp. 23, available at http://jobfunctions.bnet.com/abstract. aspx?docid=288048 (visited August 1, 2008).

[LASK05] LEITNER, C., ALABAU, A., SOTO MORA, G., KREUZEDER, M., *Organizational Changes, Skills and the Role of Leadership required by eGovernment*. European Institute for Public Administration, Luxembourg, 2005, Ministere de la Fonction Publique et de la Reforme Administrative, European Institute for Public Administration, pp. 182, available at http://ec.europa.eu/idabc/servlets/Doc?id=21547 (visited October 24, 2008).

[Law03] LAW, E., *Challenges for e-Government Development*. United Nations Public Administration Network (UNPAN), Mexico, DC, Mexico, 2003, pp. 18, available at http://unpan1.un.org/intradoc/ groups/public/documents/UN/UNPAN012241.pdf (visited August 1, 2008).

[LDL08] LIVERPOOL DIRECT LIMITED (LDL), *Liverpool Direct*, LDL, 2008, available at http://www.liverpooldirectlimited. co.uk/ index.asp (visited October 25, 2008).

[LKM06] LIU, S., KUNGAS, P. and MATSKIN, M., Agent-Based Web Service Composition with JADE and JXTA. In Proceedings of the 2006 International Conference on Semantic Web & Web Services, SWWS 2006, Las Vegas, United States of America, 2006, pp. 110-116, ISBN 1-60132-016-7, available at www.idi.ntnu.no/~peep/ papers/SWWS2006_LiKM.pdf (visited August 20, 2008).

[Mat05] MATS, W., *Swedish Single eApplication for Exports*. ePractice.eu Portal, European Commission, 2005, available at http://www.epractice.eu/cases/1960 (visited June 21, 2008).

[MD02] MARGETTS, H. and DUNLEAVY, P., *Better Public Services through e-Government: Academic Article in Support of Better Public Services through e-Government - Cultural Barriers to E-Government*. Better Public Services through e-Government: Academic Article in Support of Better Public Services through e-Government - Cultural Barriers to E-Government, London, 2002, available at http://www.governmentontheweb. co.uk/downloads/papers/Cultural_Barriers.pdf (visited August 1, 2008).

[Meh08] MEHTA, N., *Mobile Web Development: Building Mobile Websites, SMS and MMS Messaging, Mobile Payments, and Automated Voice Call Systems with XHTML MP, WCSS, and Mobile AJAX.* Packt Publishing Ltd. Birmingham, United Kingdom. ISBN 978-1-847193-53-8, February 2008.

[Mer08] MERRIAN-WEBSTER, *Definition from the Merrian-Webster Online Dictionary*. Dictionary and Thesaurus – Merrian-Webster Online, 2008, available at http://www.merriam-webster.com (visited October 25, 2008).

[Mic07] MICROSOFT CORPORATION, *Connected Government Framework*, 2007. available at http://www.microsoft. com/interop/govt/ cgf/default.mspx (visited July 27, 2008).

[Mic08a] MICROSOFT CORPORATION, *MSMQ Overview*. Microsoft Development Network (MSDN), 2008, available at http://msdn.microsoft.com/en-us/library/ms811051.aspx (visited August 15, 2008).

[Mic08b] MICROSOFT CORPORATION, *Understanding MSMQ – Chapter 1*. Microsoft Technet: Resources for IT Professionals, Microsoft Corporation, 2008, available at http://technet.microsoft.com/en-us/library/cc723251.aspx (visited August 15, 2008).

[Mic08c] MICROSOFT CORPORATION, *Resource Management in MSMQ Applications*. Microsoft Development Network (MSDN), 2008, available at http://msdn.microsoft.com/en-us/library/ms811056.aspx (visited August 15, 2008).

[Mic08d] MICROSOFT CORPORATION, *Message Queuing (MSMQ): Directory Service Discovery Protocol Specification*. Microsoft Corporation, 2008, available at http://download.microsoft.com/download /9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5BMS-MQSD%5D.pdf (visited August 15, 2008).

[Mic08e] MICROSOFT CORPORATION, *Microsoft Biz Talk Server Product Overview*. Microsoft Corporation, 2008, available at http://www.microsoft.com/biztalk/en/us/overview.aspx (visited August 15, 2008).

[Mil80] MILNER, R., *A Calculus for Communicating Systems*. Lecture Notes in Computer Science, 92, Springer, 1980.

[Mil81] MILLER, L.J., *The ISO Reference Model of Open Systems Interconnection: A First Tutorial*. Association for Computing Machinery (ACM), ACM Annual Conference/Annual Meeting, Proceedings of the ACM'81 Conference, 1981, pp. 6, ISSN 0-89791-049-4, available at http://portal.acm.org/citation.cfm?id= 800175.809901&coll=GUIDE&dl=ACM&%20type=series&idx=800175&part=series&WantType=series&title=ACM%2FCSCER.

[Mil91] MILNER, R., *The Polyadic-Calculus: A Tutorial*. Logic and Algebra of Specification, 1991, available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.5962 (visited October 26, 2008).

[Mil93] MILNE, R., *The Formal Basis for the RAISE Specification Language*. In Proceedings of the International Workshop on Semantics of Specification Languages (SoSL), Springer-Verlag, London, 1993, pp. 23-50, ISBN 3-540-19854-7.

[Mil99] MILNER, R., *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999, ISBN 0-521-65869-1.

[MPS04] MARQUES PEREIRA, C. and SOUSA, P., *A Method to Define an Enterprise Architecture using the Zachman Framework*. In Proceedings of the 2004 ACM symposium on Applied Computing, 2004, pp. 1366-1371, available at http://portal.acm.org/citation.cfm?id=967900.968175 (visited August 8, 2008).

[MPW89] MILNER, R., PARROW, J., AND WALKER, D., *A Calculus of Mobile Processes, Parts I and II*, June 1989 (Revised September 1990), Information and Computation, Vol. 100, pp. 1-40 and 41-77, available at http://www.lfcs.informatics.ed.ac.uk/reports/89/ECS-LFCS-89-85/ (visited October 7, 2008).

[MWC99] MUTHU, S., WHITMAN, L., and CHERAGHI, S. H., *Business Process Re-engineering: A Consolidated Methodology*. In Proceedings of the 4th Annual International Conference on Industrial Engineering Theory, Applications and Practice, San Antonio, Texas, United States of America, 1999, pp. 1-5, available at http://webs.twsu.edu/whitman/papers/ijii99muthu.pdf (visited 25/06/2008).

[MZ95] MOWBRAY, T. J. and ZAHAVI, R., *The Essential CORBA*. John Wiles & Sons, 1995, ISBN 978-0-4711-0611-1.

[NAO08] NATIONAL AUDIT OFFICE (NAO), *UK National Audit Office*. Government of United Kingdom, 2008, available at http://www.nao.org.uk/ (visited August 1, 2008).

[NN99] NIELSON, H.R. and NIELSON, F., *Semantics with Applications: A Formal Introduction, 2nd Edition*. Wiley Professional Computing, 1999, p. 240, ISSN 0-471-92980-8, available at http://www.daimi.au.dk/ ~bra8130/Wiley_book/wiley.html (visited August 9, 2008).

[NZ08a] NEW ZEALAND GOVERNMENT, STATE SERVICES COMMISSION, *New Zealand e-Government Interoperability Framework (NZ e-GIF) Version 3.3, Introduction*. State Services Commission, The Government of New Zealand, 2008, pp. 91, Technical Report, available at http://www.e.govt.nz/ standards/e-gif/e-gif-v-3-3/e-gif-v-3-3-complete.pdf (visited October 7, 2008).

[NZ08b] NEW ZEALAND GOVERNMENT, STATE SERVICES COMMISSION, *NZGLS Metadata Element Set Version 2.1*. State Services Commission, The Government of New Zealand, 2008, E-Government in New Zealand - New Zealand E-Government Programme, 2004, available at http://www.e.govt.nz/ standards/nzgls/standard (visited August 8, 2008).

[NZ08c] NEW ZEALAND GOVERNMENT, STATE SERVICES COMMISSION, *NZGLS Thesauri*. State Services Commission, The Government of New Zealand, 2008, E-Government in New Zealand - New Zealand E-Government Programme, 2004, available at http://www.e.govt.nz/standards/nzgls/ thesauri/downloads.html (visited August 8, 2008).

[OAS01] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Relax NG*. OASIS Relax NG Technical Committee, 2001, available at http://www.oasis-open.org/committees/tc_ home.php?wg_ abbrev=relax-ng (visited August 11, 2008).

[OAS02] OASIS ebXML MESSAGING SERVICES TECHNICAL COMMITTEE, *Message Service Specification Version 2.0*. OASIS, 2002, pp. 70, available at http://www.oasis-open.org/committees/download.php/272/ebMS_v2_0.pdf (visited August 20, 2008).

[OAS03] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Business Process Execution Language (BPEL)*. OASIS, May 2003, available at http://www.oasis-open.org/committees/ download.php/2046/BPEL%20V1-1%20May%2005%202003%20.Final.pdf (visited July 27, 2008).

[OAS04] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *WS Reliability 1.1*. OASIS Web Services Reliable Messaging Technical Committee, 2004, pp. 72, available at docs.oasis-open.org/wsrm/ws-reliability/ v1.1/wsrm-ws_reliability-1.1-spec-os.pdf (visited August 21, 2008).

[OAS06a] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Web Services Security 1.1*. OASIS, February, 2006, available at http://www.oasis-open.org/committees/tc_ home.php?_wg_abbrev=wss (visited July 27, 2008).

[OAS06b] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Web Services Base Notification 1.3 (WS-Base Notification).* OASIS, 2006, available at docs.oasis-open.org/wsn/wsn-ws_ base_notification-1.3-spec-os.pdf (visited August 21, 2008).

[OAS07a] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Universal Description, Discovery and Integration (UDDI)*. OASIS, 2007, available at http://uddi.xml.org (visited July 27, 2008).

[OAS07b] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Web Services Coordination (WS Coordination) Version 1.1*. OASIS, 2007, available at http://docs.oasis-open.org/ws-tx/wscoor/2006/06 (visited August 21, 2008).

[OAS07b] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Web Services Atomic Transaction (WS-Atomic Transaction), Version 1.1*. OASIS, 2007, available at http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec/wstx-wsat-1.1-spec.html (visited August 21, 2008).

[OAS07c] ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS), *Web Services Business Process Execution Languages (WSBPEL) Version 2.0*. OASIS, 2007, available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf (visited August 21, 2008).

[OJOE06] OJO, A., JANOWSKI, T., OTENIYA, G., ESTEVEZ, E., *Infrastructure Support for e-Government – An Overview.* E-Macao Report N. 7, August 2006, available at http://www.emacao.gov.mo/documents/06/report7.pdf (visited April 26, 2009).

[OJE07a] OJO, A., JANOWSKI, T., ESTEVEZ, E., *Rapid Development of Electronic Public Services – Software Infrastructure and Software Process.* In proceedings of 8th Annual International Conference on Digital Government Research, dg.o 2007, Philadelphia, USA, May 2007. Digital Government Research Centre.

[OJE07b] OJO, A., JANOWSKI, T., ESTEVEZ, E., *A Composite Domain Framework for Developing Electronic Public Services*. UNU-IIST Technical Report 370, April 2007. Presented and published in the proceedings of the International Conference on Software Engineering Theory and Practice (SETP-07), Orlando, USA, July 2007, available at http://www.iist.unu.edu (visited 26 April, 2009).

[OJE07c] OJO, A., JANOWSKI, T., ESTEVEZ, E., *Domain Models and Enterprise Application Framework for Developing Electronic Public Services*. UNU-IIST, Technical Report 369, April 2007. Presented and published in the proceedings of the 6th International EGOV Conference 2007, Regensburg, Germany, September 2007, Trauner Druck, available at http://www.iist.unu.edu (visited 26 April, 2009).

[OJE07d] OJO, A., JANOWSKI, T., ESTEVEZ, E., *Rapid Development of Electronic Public Services – A Case Study in Electronic Licensing Services*. In proceedings of 8th Annual International Conference on Digital Government Research, dg.o 2007, Philadelphia, USA, May 2007. Digital Government Research Centre.

[OPS04] OFFICE OF PUBLIC SERVICE, MERIT AND EQUITY. *Seamless Government: Improving Outcomes for Queenslanders, Now...and in the Future*. Queensland Government, 2004, available at http://www.opsc.qld.gov.au/ library/docs/resources/publications/Notices/SeamlessGovernment.pdf (visited June 20, 2008).

[Par03] PARTNERSHIPS BRITISH COLUMBIA, *An Introduction to Public-Private Partnership*. Partnerships British Columbia, 2003, pp.7, available at http://www.partnershipsbc.ca/pdf/An%20Introduction%20 to%20P3%20-June03.pdf (visited July 29, 2008).

[Pic00] PICKETT, J. et al (Eds), *The American Heritage Dictionary of the English Language*. Houghton Mifflin Company, Boston, 2000, ISBN 0-395-82517-2.

[PT04] PERISTERAS, V. and TARABANIS, K., *Governance Enterprise Architecture (GEA): Domain Models for e-Governance*. In Proceedings of the 6th International Conference on Electronic Commerce, Delft, The Netherlands, ACM International Conference Proceeding Series, Vol 60, pp. 471-479, ISBN: 1-58113-930-6, 2004.

[PTF08] Pi4SOA TECHNOLOGIES FOUNDATION, *Pi-Calculus for SOA*. Pi4SOA Project wiki, 2008, available at http://pi4soa.wiki.sourceforge.net/ (visited October 26, 2008).

[Ray01] RAY, E., Learning XML, O'Reilly, 2001.

[Ray93] RAYMOND, K., *Reference Model of Open Distributed Processing: A Tutorial*. In Proceedings of the IFIP TC6 International Conference on Open Distributed Processing, 1993, pp. 3-14, available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.105.6373 (visited August 8, 2008).

[Rea02] REACH, *IAMS Briefing Document - Inter Agency Messaging Service*. Reach, 2002, available at http://www.reach.ie/iams/IAMS_Overview_v1-1.pdf (visited October 7, 2008).

[Rea08] REACH AGENCY – GOVERNMENT OF IRELAND, *Reach Services – Connecting People and Public Services*. Reach Agency, 2008, available at http://www.reachservices.ie/portalapp/index.htm (visited August 15, 2008).

[RH08] RED HAT, *JBoss Business Process Management* (jBoss jBPM). Red Hat, 2008, available at http://www.jboss.com/products/jbpm (visited August 19, 2008).

[Ril04] RILEY, T., *E-Government - The Digital Divide and Information Sharing: Examining the Issues*. Commonwealth Centre for E-Governance and Riley Information Services Inc., Ottawa, Canada, 2004, pp. 37, available at http://www.rileyis.com/publications/research_papers/track04/ Divide1Vs.3.pdf (visited August 1, 2008).

[RLG92] THE RAISE LANGUAGE GROUP, *The RAISE Specification Language,* Prentice Hall, 1992, ISBN 0-13-752833-7, out of print, available at: ftp://ftp.iist.unu.edu/pub/RAISE/method_book/ (visited October 7, 2008).

[RMG96] THE RAISE METHOD GROUP, *The RAISE Development Method*. BCS Practitioner Series, Prentice Hall, 1996, available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book (visited August 24, 2008).

[Rog97] ROGERSON, S., *Data Matching*. Institute for the Management of Information Systems (IMIS) Journal, vol. 7, 1997, available at http://www.ccsr.cse.dmu.ac.uk/resources/general/ethicol/ Ecv7no1.pdf (visited August 1, 2008).

[Ron02] RONAGHAN, S., *Benchmarking e-Government: A Global Perspective Assessing the UN Member States*. United Nations Division for Public Economics and Public Administration (UN-DPEPA) and American Society for Public Administration (ASPA), pp. 81, 2002, available at http://unpan1.un.org/intradoc/groups/public/documents/ UN/UNPAN021547.pdf (visited June 21, 2008).

[Ros05] ROSCOE, A. W., *The Theory and Practice of Concurrency.* Prentice-Hall, 2005, available at web.comlab.ox.ac.uk/people/Bill.Roscoe/publications/68b.pdf (visited October 7, 2008).

[SAG08] SOFTWARE AG, *WebMethods*, 2008, available at http://www.softwareag.com/Corporate/products/ wm/default.asp (visited October 25, 2008).

[Sak02] SAKOWICZ, M., *How to Evaluate e-Government? Different Methodologies and Methods*. Warsaw School of Economic, Department of Public Administration, 2002, available at intradoc/groups/public/ documents/NISPAcee/UNPAN009486.pdf (visited June 06, 2008).

[San93] SANGIORGI, D., A Theory of Bisimulation for the Pi-Calculus, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, United Kingdom, May 1993.

[San08] SANCHEZ, A. *Semantic Interoperability for Programmable Messaging Middleware – Mediation, Validation and Discovery Extensions,* MSc thesis presented and defended at National University of San Luis, 18 September 2008.

[Sch00] SCHACTER, M., *Public Sector Reform in Developing Countries - Issues, Lessons and Future Directions*. Institute on Governance, Ottawa, Canada, Canadian International Development Agency, 2000, pp. 17, available at www.iog.ca/publications/ps_reform.PDF (visited October 25, 2008).

[SDC06] SOMMERVILLE, I., DEWSBURY, G., CLARKE, K. and ROUNCEFIELD, M., *Dependability and Trust in Organisational and Domestic Computer Systems*. In Trust in Technology: A Socio-Technical Perspective, Springer Netherlands, 2006, vol. 36, chapter 8, pp. 169-193, ISBN 978-1-4020-4257-7 (Printed version) and 978-1-4020-4258-4 (Online version), available at http://www.springerlink.com/content/g540335r6uw836j5/ (visited August 2, 2008).

[SEI07] SOFTWARE ENGINEERING INSTITUTE (SEI). *Message-Oriented Middleware*. Carnegie Mellon University, 2007, available at http://www.sei.cmu.edu/str/descriptions/momt_body.html (visited October 7, 2008).

[Sen08] SEN3 RESEARCH GROUP, *Eclipse Coordination Tools*. Centrum Wiskunde & Informatica, Amsterdam, The Netherlands, available at http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools (visited October 26, 2008).

[Ser02] SERAIN, D., *Middleware and Enterprise Application Integration, The Architecture of e-Business Solutions, 2nd Edition*. Springer-Verlag, 2002, ISBN 978-1-85233-570-0, available at http://www.springer.com/computer/ communications/book/978-1-85233-570-0 (visited August 14, 2008).

[SGC06] SEMANTIC GOV CONSORTIUM, *D012: State of the Art Report*. August 2006, available at www.semantic-gov.org/index.php?name=UpDownload&req=getit&lid=215 (visited July 27, 2008).

[Sha00] SHARPE, R., *Citizens' Preferences, Measuring the Acceptability of Channels*. Kable, Ltd, London, England, pp. 21, 2000, available at http://www.egov.vic.gov.au/pdfs/Citizen%2BPreferences.pdf (visited June 21, 2008).

[Sob07] SOBOCIŃSKI, P., *A Well-behaved LTS for the Pi-calculus*. In Electronic Notes in Theoretical Computer Science (ENTCS), Amsterdam, The Netherlands, Elsevier Science Publishers B.V., 2007, vol. 192, issue 1, pp. 5-11, ISSN:1571-0661, available at http://www.google.com/search?client=opera &rls=en&q=A+Well-behaved+LTS+for+the+Pi-calculus&sourceid=opera&ie=utf-8&oe=utf-8 (visited October 26, 2008).

[Sou84] SOUNDARARAJAN, N., *Axiomatic Semantics of Communicating Sequential Processes*. ACM Transactions on Programming Languages and Systems (TOPLAS), ACM, 1984, vol. 6, issue 4, pp. 647-662, ISBN 0164-0925 available at http://portal.acm.org/citation.cfm?id=1805&coll=portal &dl=ACM (visited August 24, 2008).

[Sta96] STARK, I., *A Fully Abstract Domain Model for the Pi-Calculus*. In Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1996, pp. 36-42, http://www.inf.ed.ac.uk/~stark/fuladm.html (visited October 26, 2008).

[Sun08a] SUN MICROSYSTEMS, *Java Message Service*. 2008, available at http://java.sun.com/products/jms (visited July 27, 2008).

[Sun08b] SUN MICROSYSTEMS, *JXTA Technology*. Sun Microsystem, 2008, available at http://www.sun.com/ software/jxta/ (visited August 20, 2008).

[Sun09] SUN MICROSYSTEMS, MySQL, http://www.mysql.com, visited 14 February 2009.

[TI08a] TELECOM ITALY, *Java Agent DEvelopment Framework – JADE Documentation*. Telecom Italy, 2008, available at http://jade.tilab.com/ (visited October 7, 2008).

[TI08b] TELECOM ITALY, *Telecom Italy Lab*. Telecom Italy, 2008, available at http://www.telecomitalia.it/ (visited August 19, 2008).

[TN99] TURNER, E. and NICOLL, P., *Electronic Service Delivery, including Internet Use by Commonweath Government Agencies*. Australian National Audit Office, Commonweath of Australia, 1999, pp. 87, ISSN 1036-7632, ISBN 0 644 39194 4, available at http://www.anao.gov.au/uploads/documents/1999-00_Audit_Report_18.pdf, (visited June 21, 2008).

[Tre08a] HM TREASURY, *Public-Private Partnerships – The Government's Approach*. Technical Report, HM Treasury, Government of United Kingdom, London, England, The Stationary Office, 2008, pp. 27, available at http://www.hm-treasury.gov.uk/mediastore/otherfiles/PPP2000.pdf (visited July 29, 2008).

[Tre08b] HM TREASURY, *Public-Private Partnerships – The Private Finance Initiative*. HM Treasury, Government of United Kingdom, London, England, 2008, available at http://www.hm-treasury.gov.uk/documents/ public_private_partnerships/ppp_index.cfm (visited July 29, 2008).

[TT05] TAMBOURIS, E. and TARABANIS, K, *e-Government and Interoperability*. In Proceedings of 5th European Conference on E-Government (ECEG 2005), University of Antwerp, Belgium, 16-17 June 2005, pp. 399-407.

[UK07] UNITED KINGDOM GOVERNMENT, E-GOVERNMENT UNIT, *e-Government Interoperability Framework*. Cabinet Office, United Kingdom, 2007, available at www.govtalk.gov.uk/interoperability/egif.asp (visited October 7, 2008).

[UNE06] UNITED NATIONS ECONOMIC AND SOCIAL COUNCIL, *Definition of Basic Concepts and Terminologies in Governance and Public Administration.* United Nations Economic and Social Council, Committee of Experts in Public Administration, 2006, pp. 15, available at http://unpan1.un.org/intradoc/groups/public/documents/unpan022332.pdf (visited October 7, 2008).

[Uni09] UNICODE, Inc., *About the Unicode Standard*, available at http://www.unicode.org/standard/standard.html, visited May 4, 2009.

[USDT08] UNITED STATES DEPARTMENT OF TRANSPORTATION (USDOT), *BTA - Capital Expenditures*. Research and Innovative Technology Administration (RITA), USDOT, 2008, available at http://www.bts.gov/publications/government_transportation_financial_statistics/2003/html/capital_expenditures.html (visited August 1, 2008).

[USP08] UNITED STATES POSTAL SERVICE (USPS), *USPS – The Official Change of Address Form*. USPS, Imagitas, 2008, available at https://moversguide.usps.com/?referral=USPS (visited July 29, 2008).

[Vit04] VITAGLIONE, G., *Mutual-authenticated SSL IMTP connections*. Telecom Italia Lab (TILAB), 2004, pp. 6, available at jade.tilab.com/doc/tutorials/SSL-IMTP/SSL-IMTP.doc (visited August 20, 2008).

[VMDE07] VICTOR, B., MOLLER, F., DAM, M., and ERIKSSON, L-H., *The Mobility Workbench*. Department of Information Technology, Uppsala University, 2007, available at http://www.it.uu.se/research/group/mobility/mwb (visited October 26, 2008).

[W3C00] WORLD WIDE WEB CONSORTIUM (W3C), *SOAP Messages with Attachments.* W3C, 2000, available at http://www.w3.org/TR/SOAP-attachments (visited August 21, 2008).

[W3C01] WORLD WIDE WEB CONSORTIUM (W3C), *Web Services Description Language*. W3C, March 2001, available at http://www.w3.org/TR/wsdl (visited July 27, 2008).

[W3C03] WORLD WIDE WEB CONSORTIUM (W3C), *Simple Object Access Protocol (SOAP)*. W3C, June 2003, available at http://www.w3.org/TR/soap/ (visited July 27, 2008).

[W3C04a] WORLD WIDE WEB CONSORTIUM (W3C), *Web Services Architectures*. W3C, 2004, available at www.w3.org/TR/ws-arch/#service_oriented_architecture (visited July 26, 2008).

[W3C04b] WORLD WIDE WEB CONSORTIUM (W3C), *Web Services Glossary*. W3C, 2004, available at http://www.w3.org/TR/ws-gloss/ (visited July 27, 2008).

[W3C04c] WORLD WIDE WEB CONSORTIUM (W3C), *XML Schema*. W3C, October 2004, available at http://www.w3.org/XML/Schema (visited July 27, 2008).

[W3C04d] WORLD WIDE WEB CONSORTIUM (W3C), *Web Services Choreography Description Language (WS-CDL)*, Working Draft. W3C, November 2004, available at http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/ (visited july 27, 2008).

[W3C04e] WORLD WIDE WEB CONSORTIUM (W3C), *Resource Description Framework (RDF)*. W3C, 2004, available at http://www.w3.org/RDF/ (visited July 27, 2008).

[W3C04f] WORLD WIDE WEB CONSORTIUM (W3C), *Web Ontology Language (OWL)*. W3C, February 2004, available at http://www.w3.org/TR/owl-features (visited July 27, 2008).

[W3C04g] WORLD WIDE WEB CONSORTIUM (W3C), *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C, February 2004, available at http://www.w3c.org/TR/rdf-schema/ (visited July 27, 2008).

[W3C04h] WORLD WIDE WEB CONSORTIUM (W3C), *Voice Extensible Markup Language (VoiceXML) Version 2.0,* W3C, March 2004, available at http://www.w3.org/TR/voicexml20/ (visited May 7, 2009)

[W3C05] WORLD WIDE WEB CONSORTIUM (W3C), *Web Service Modeling Ontology (WSMO)*. W3C, June 2005, available at http://www.w3.org/Submission/WSMO/ (visited July 27, 2008).

[W3C08] WORLD WIDE WEB CONSORTIUM (W3C), *Extensible Markup Language (XML).* W3C, October 14, 2008, available at http://www.w3.org/XML/ (visited October 26, 2008).

[W3C99a] WORLD WIDE WEB CONSORTIUM (W3C), *XSL Transformations (XSLT)*. W3C, November 16, 1999, available at http://www.w3.org/TR/xslt (visited July 27, 2008).

[W3C99b] WORLD WIDE WEB CONSORTIUM (W3C), *Document Type Definition (DTD)*, W3C, 1999, available at http://www.w3.org/TR/html4/sgml/dtd.html (visited July 27, 2008).

[W3C99c] WORLD WIDE WEB CONSORTIUM (W3C), *Document Type Definition (DTD)*. W3C, 1999, available at http://www.w3.org/TR/html4/sgml/dtd.html (visited July 27, 2008).

[War08] WARD, S., *Capital Investment – Capital Investment Definitions*. Welcome to About.com, 2008, available at http://sbinfocanada.about.com/od/financing/g/capinvestment.htm (visited August 1, 2008).

[WD04] WAUTERS, P., VAN DURME, P., *Online Availability of Public Services: How is Europe Progressing? - Web Bases Survey on Electronic Public Services*. Report of the Fifth Measurement, October 2004, Capgemini Ernst and Young, 2005, available at http://ec.europa.eu/information_society/ eeurope/i2010/docs/benchmarking/ online_availability_2006.pdf (visited June 21, 2008).

[Win02] WING, J. *FAQ on Pi-Calculus*.2002, available at www.cs.cmu.edu/~wing/publications/Wing02a.pdf (visited October 26, 2008).

[Yus04] YUSUF, K., *Enterprise Messaging Using JMS and IBM WebSphere*. Prentice Hall, 2004, ISBN 0-13-146863-4.

[Zac87] ZACHMAN, J. A., *A Framework for Information Systems Architecture*. IBM, 1987, IBM Systems Journal, Vol. 276, pp. 276-291, available at http://www.research.ibm.com/journal/sj/263/ ibmsj2603E.pdf (visited October 7, 2008).

[ZC08] ZOPE CORPORATION, *What is Zope*? ZOPE Corporation, 2008, available at http://www.zope.org/ WhatIsZope (visited August 19, 2008).

Apéndices

Apéndice A – Especificaciones en RSL

A.1. MODEL 1 – Direct Messaging

Scheme 1: MODEL 1 - ID

```
scheme ID = class

type
  MemberId

end
```

Scheme 2: MODEL 1 - MESSAGE

```
ID

scheme MESSAGE(I:ID) = class

type
  Message

value
  sender: Message -> I.MemberId,
  recepients: Message -> I.MemberId-set

end
```

Scheme 3: MODEL 1 - QUEUE

```
ID, MESSAGE

scheme QUEUE(I:ID, M:MESSAGE(I)) = class

type
  Queue = M.Message-list

value
  isEmpty: Queue -> Bool
  isEmpty(q) is q = <..>

value
  init: Queue = <..>,

  enqueue: M.Message >< Queue -> Queue
  enqueue(m, q) is q ^ <.m.>,

  dequeue: Queue -~-> M.Message >< Queue
  dequeue(q) is (hd q, tl q) pre ~isEmpty(q)

end
```

Scheme 4: MODEL 1 - MEMBER

```
ID, MESSAGE, QUEUE
```

```
scheme MEMBER(I:ID, M:MESSAGE(I)) = extend QUEUE(I, M) with class

type
  Member
value
  inbox: Member -> Queue,
  outbox: Member -> Queue

value
  isEmptyInbox: Member -> Bool
  isEmptyInbox(e) is isEmpty(inbox(e)),

  isEmptyOutbox: Member -> Bool
  isEmptyOutbox(e) is isEmpty(outbox(e)),

  isInInbox: M.Message >< Member -> Bool
  isInInbox(m, e) is m isin elems inbox(e),

  isInOutbox: M.Message >< Member -> Bool
  isInOutbox(m, e) is m isin elems outbox(e)

value
  init: Member :- inbox(init) = init /\ outbox(init) = init

value
  enqueueInbox: M.Message >< Member -> Member
  enqueueInbox(m, e) as e' post
    inbox(e') = enqueue(m, inbox(e)) /\ outbox(e') = outbox(e),

  dequeueInbox: Member -~-> (M.Message >< Member)
  dequeueInbox(e) as (m, e') post
    let (m', q) = dequeue(inbox(e)) in
      m = m' /\ inbox(e') = q /\ outbox(e') = outbox(e)
    end
    pre ~isEmptyInbox(e)

value
  enqueueOutbox: M.Message >< Member -> Member
  enqueueOutbox(m, e) as e' post
    inbox(e') = inbox(e) /\ outbox(e') = enqueue(m, outbox(e)),

  dequeueOutbox: Member -~-> (M.Message >< Member)
  dequeueOutbox(e) as (m, e') post
    let (m', q) = dequeue(outbox(e)) in
      m = m' /\ inbox(e') = inbox(e) /\ outbox(e') = q
    end
    pre ~isEmptyOutbox(e)

end
```

Scheme 5: MODEL 1 - STATE

```
ID, MESSAGE, MEMBER

scheme STATE(I:ID, M:MESSAGE(I), E:MEMBER(I,M)) = class

type
  State’ = I.MemberId -m-> E.Member,
  State = {| s:State’ :- iswf(s) |}

value
  init: State’ = []

value
```

```
  memberExists: I.MemberId >< State' -> Bool
  memberExists(i, s) is i isin dom s,

  membersExist: I.MemberId-set >< State' -> Bool
  membersExist(ids, s) is ids <<= dom s,

  getMember: I.MemberId >< State' -~-> E.Member
  getMember(i, s) is s(i)
    pre memberExists(i, s)

value
  iswfMessage: M.Message >< State' -> Bool
  iswfMessage(m, s) is
    memberExists(M.sender(m), s) /\
    (all i: I.MemberId :- i isin M.recepients(m) => memberExists(i, s)),

  iswfQueue: E.Queue >< State' -> Bool
  iswfQueue(q, s) is
    q = <..> \/ iswfMessage(hd q, s) /\ iswfQueue(tl q, s),

  iswfMember: I.MemberId >< State' -> Bool
  iswfMember(i, s) is
    let e = getMember(i, s) in
      iswfQueue(E.inbox(e), s) /\
      iswfQueue(E.outbox(e), s) /\
      (all m: M.Message :-
        (E.isInInbox(m, e) => M.sender(m) = i) /\
        (E.isInOutbox(m, e) => i isin M.recepients(m))
      )
    end,

  iswf: State' -> Bool
  iswf(s) is
    (all i: I.MemberId :-
      i isin dom s => iswfMember(i, s)
    )

value
  addMember: I.MemberId >< E.Member >< State -~-> State
  addMember(i, e, s) is s union [ i +> e ]
    pre ~memberExists(i, s),

  deleteMember: I.MemberId >< State -~-> State
  deleteMember(i, s) is s \ {i}
    pre memberExists(i, s),

  modifyMember: I.MemberId >< E.Member >< State -~-> State
  modifyMember(i, e, s) is s !! [ i +> e ]
    pre memberExists(i, s)

end
```

Scheme 6: MODEL 1 - GATEWAY

```
ID, MESSAGE, MEMBER, STATE

scheme GATEWAY(I:ID, M:MESSAGE(I), E:MEMBER(I,M), S:STATE(I,M,E)) = class

value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is
    S.addMember(i, E.init, s)
    pre canRegister(i, s),

  canRegister: I.MemberId >< S.State -> Bool
```

```
  canRegister(i, s) is ~S.memberExists(i, s)

value
  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is
    S.deleteMember(i, s)
    pre canUnregister(i, s),

  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is
    S.memberExists(i, s)

value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let
      i = M.sender(m),
      e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end
    pre canSend(m, s),

  canSend: M.Message >< S.State -> Bool
  canSend(m, s) is S.iswfMessage(m, s)

value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let (m, e) = E.dequeueOutbox(S.getMember(i, s)) in
      (m, S.modifyMember(i, e, s))
    end
    pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is ~E.isEmptyOutbox(S.getMember(i, s))

value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s)
    in transferAll(m, M.recepients(m), s') end
    pre canTransfer(i, s),

  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let
          e = E.enqueueOutbox(m, S.getMember(i, s)),
          s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is S.membersExist(ids, s) /\ ids <<= M.recepients(m)

end
```

A.2. MODEL 2 – Messaging Through Channels

Scheme 7: MODEL 2 - ID

```
scheme ID = class

type
  MemberId, ChannelId

end
```

Scheme 8: MODEL 2 - MESSAGE

```
ID

scheme MESSAGE(I:ID) = class

type
  Message
value
  sender: Message -> I.MemberId,
  recepients: Message -> I.ChannelId

end
```

Scheme 9: MODEL 2 - CHANNEL

```
ID

scheme CHANNEL(I:ID) = class

type
  Channel
value
  owner: Channel -> I.MemberId,
  subscribers: Channel -> I.MemberId-set
axiom
  (all c: Channel :- owner(c) ~isin subscribers(c))

value
  isOwner: I.MemberId >< Channel -> Bool
  isOwner(i, c) is i = owner(c),

  isSubscriber: I.MemberId >< Channel -> Bool
  isSubscriber(i, c) is i isin subscribers(c),

  noSubscribers: Channel -> Bool
  noSubscribers(c) is subscribers(c) = {},

  init: I.MemberId -> Channel
  init(i) as c post owner(c) = i /\ subscribers(c) = {},

  subscribe: I.MemberId >< Channel -~-> Channel
  subscribe(i, c) as c' post
    owner(c') = owner(c) /\ subscribers(c') = subscribers(c) union {i}
    pre ~isSubscriber(i, c),

  unsubscribe: I.MemberId >< Channel -~-> Channel
  unsubscribe(i, c) as c' post
    owner(c') = owner(c) /\ subscribers(c') = subscribers(c) \ {i}
    pre isSubscriber(i, c)

end
```

Scheme 10: MODEL 2 - STATE

```
ID, MESSAGE, MEMBER, CHANNEL

scheme STATE(I:ID, M:MESSAGE(I), E:MEMBER(I,M), C:CHANNEL(I)) = class

type
  Members = I.MemberId -m-> E.Member,
  Channels = I.ChannelId -m-> C.Channel,
  State'::
    members: Members <-> re_members
    channels: Channels <-> re_channels,
  State = {| s:State' :- iswf(s) |}

value
  init: Members = [],
  init: Channels = [],
  init: State' = mk_State'(init, init)

value
  memberExists: I.MemberId >< State' -> Bool
  memberExists(i, s) is i isin dom members(s),

  membersExist: I.MemberId-set >< State' -> Bool
  membersExist(ids, s) is ids <<= dom members(s),

  channelExists: I.ChannelId >< State' -> Bool
  channelExists(i, s) is i isin dom channels(s),

  channelsExist: I.ChannelId-set >< State' -> Bool
  channelsExist(ids, s) is ids <<= dom channels(s),

  getMember: I.MemberId >< State' -~-> E.Member
  getMember(i, s) is members(s)(i) pre memberExists(i, s),

  getChannel: I.ChannelId >< State' -~-> C.Channel
  getChannel(i, s) is channels(s)(i) pre channelExists(i, s),

  channelsOwned: I.MemberId >< State' -~-> I.ChannelId-set
  channelsOwned(im, s) is
    { ic | ic: I.ChannelId :- C.owner(getChannel(ic, s)) = im }
    pre memberExists(im, s),

  channelsSubscribed: I.MemberId >< State' -~-> I.ChannelId-set
  channelsSubscribed(im, s) is
    { ic | ic: I.ChannelId :- im isin C.subscribers(getChannel(ic, s)) }
    pre memberExists(im, s)

value
  iswfMessage: M.Message >< State' -> Bool
  iswfMessage(m, s) is
    let ms = M.sender(m), mr = getChannel(M.recepients(m), s) in
      memberExists(ms, s) /\
      channelExists(M.recepients(m), s) /\
      (C.isOwner(ms, mr) \/ C.isSubscriber(ms, mr))
    end,

  iswfQueue: E.Queue >< State' -> Bool
  iswfQueue(q, s) is
    E.isEmpty(q) \/ iswfMessage(hd q, s) /\ iswfQueue(tl q, s),

  iswfMember: I.MemberId >< State' -~-> Bool
  iswfMember(i, s) is
    let e = getMember(i, s) in
      iswfQueue(E.inbox(e), s) /\ iswfQueue(E.outbox(e), s) /\
      (all m: M.Message :-
        let ms = M.sender(m), mr = getChannel(M.recepients(m), s) in
```

```
          (E.isInInbox(m, e) => ms = i) /\
          (E.isInOutbox(m, e) => (C.isOwner(i, mr) \/ C.isSubscriber(i, mr)))
        end
      )
    end pre memberExists(i, s),

  iswfChannel: C.Channel >< State' -> Bool
  iswfChannel(c, s) is
    memberExists(C.owner(c), s) /\
    membersExist(C.subscribers(c), s),

  iswfMembers: Members >< State' -> Bool
  iswfMembers(ms, s) is
    (all i: I.MemberId :-
      memberExists(i, s) => iswfMember(i, s)
    ),

  iswfChannels: Channels >< State' -> Bool
  iswfChannels(cs, s) is
    (all i: I.ChannelId :-
      channelExists(i, s) => iswfChannel(getChannel(i, s), s)
    ),

  iswf: State' -> Bool
  iswf(s) is
    iswfMembers(members(s), s) /\
    iswfChannels(channels(s), s)

value
  addMember: I.MemberId >< E.Member >< State -~-> State
  addMember(i, e, s) is
    re_members(members(s) union [ i +> e ], s)
    pre ~memberExists(i, s),

  deleteMember: I.MemberId >< State -~-> State
  deleteMember(i, s) is
    re_members(members(s) \ {i}, s)
    pre memberExists(i, s),

  modifyMember: I.MemberId >< E.Member >< State -~-> State
  modifyMember(i, e, s) is
    re_members(members(s) !! [ i +> e ], s)
    pre memberExists(i, s)

value
  addChannel: I.ChannelId >< C.Channel >< State -~-> State
  addChannel(i, c, s) is
    re_channels(channels(s) union [ i +> c ], s)
    pre ~channelExists(i, s),

  deleteChannel: I.ChannelId >< State -~-> State
  deleteChannel(i, s) is
    re_channels(channels(s) \ {i}, s)
    pre channelExists(i, s),

  modifyChannel: I.ChannelId >< C.Channel >< State -~-> State
  modifyChannel(i, c, s) is
    re_channels(channels(s) !! [ i +> c ], s)
    pre channelExists(i, s)

end
```

Scheme 11: MODEL 2 - GATEWAY

```
ID, MESSAGE, MEMBER, CHANNEL, STATE
```

```
scheme GATEWAY(I:ID, M:MESSAGE(I), E:MEMBER(I,M), C:CHANNEL(I), S:STATE(I,M,E,C)) =
class

value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is
    S.addMember(i, E.init, s)
    pre canRegister(i, s),

  canRegister: I.MemberId >< S.State -> Bool
  canRegister(i, s) is ~S.memberExists(i, s)

value
  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is
    S.deleteMember(i, s) pre canUnregister(i, s),

  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is
    S.memberExists(i, s) /\ S.channelsOwned(i, s) = {} /\
    S.channelsSubscribed(i, s) = {}

value
  create: I.ChannelId >< I.MemberId >< S.State -~-> S.State
  create(ic, im, s) is
    S.addChannel(ic, C.init(im), s)
    pre canCreate(ic, im, s),

  canCreate: I.ChannelId >< I.MemberId >< S.State -> Bool
  canCreate(ic, im, s) is
    ~S.channelExists(ic, s) /\ S.memberExists(im, s)

value
  destroy: I.ChannelId >< S.State -~-> S.State
  destroy(ic, s) is
    S.deleteChannel(ic, s) pre canDestroy(ic, s),

  canDestroy: I.ChannelId >< S.State -> Bool
  canDestroy(ic, s) is
    S.channelExists(ic, s) /\ C.noSubscribers(S.channels(s)(ic))

value
  subscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribe(im, ic, s) is
    let c = C.subscribe(im, S.getChannel(ic, s))
    in  S.modifyChannel(ic, c, s) end
    pre canSubscribe(im, ic, s),

  canSubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canSubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    ~C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~C.isOwner(im, S.getChannel(ic, s))

value
  unsubscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  unsubscribe(im, ic, s) is
    let c = C.unsubscribe(im, S.getChannel(ic, s))
    in S.modifyChannel(ic, c, s) end
    pre canUnsubscribe(im, ic, s),

  canUnsubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canUnsubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    C.isSubscriber(im, S.getChannel(ic, s))
```

```
value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let i = M.sender(m), e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end
    pre canSend(m, s),

  canSend: M.Message >< S.State -> Bool
  canSend(m, s) is S.iswfMessage(m, s)

value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let (m, e) = E.dequeueOutbox(S.getMember(i, s))
    in (m, S.modifyMember(i, e, s)) end
    pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is
    ~E.isEmptyOutbox(S.getMember(i, s))

value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      ids = C.subscribers(S.getChannel(M.recepients(m), s))
    in transferAll(m, ids, s') end
    pre canTransfer(i, s),

  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is
    ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let e = E.enqueueOutbox(m, S.getMember(i, s)), s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end
    pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is
    S.membersExist(ids, s) /\
    ids <<= C.subscribers(S.getChannel(M.recepients(m), s))

end
```

A.3. MODEL 3 – Introducing Member and Channel Roles

Scheme 12: MODEL 3 - ID

```
scheme ID = class

type
  MemberId
value
  admin, visitor: MemberId
axiom
  admin ~= visitor

type
  ChannelId
value
  adminChannel: MemberId -> ChannelId
axiom
  (all im1, im2: MemberId :-
    im1 ~= im2 => adminChannel(im1) ~= adminChannel(im2)
  )
value
  isAdminChannel: ChannelId -> Bool
  isAdminChannel(ic) is (exists im: MemberId :- adminChannel(im) = ic)

end
```

Scheme 13: MODEL 3 - STATE

```
ID, MESSAGE, MEMBER, CHANNEL

scheme STATE(I:ID, M:MESSAGE(I), E:MEMBER(I,M), C:CHANNEL(I)) = class

type
  Members = I.MemberId -m-> E.Member,
  Channels = I.ChannelId -m-> C.Channel,
  State'::
    members: Members <-> re_members
    channels: Channels <-> re_channels,
  State = {| s:State' :- iswf(s) |}

value
  initAdmin: I.MemberId -> C.Channel
  initAdmin(i) as c post C.owner(c) = I.admin /\ C.subscribers(c) = {i}

value
  init: Members = [I.admin +> E.init, I.visitor +> E.init],
  init: Channels = [ I.adminChannel(I.visitor) +> initAdmin(I.visitor) ],
  init: State' = mk_State'(init, init)

value
  memberExists: I.MemberId >< State' -> Bool
  memberExists(i, s) is i isin dom members(s),

  membersExist: I.MemberId-set >< State' -> Bool
  membersExist(ids, s) is ids <<= dom members(s),

  channelExists: I.ChannelId >< State' -> Bool
  channelExists(i, s) is i isin dom channels(s),
```

```
  channelsExist: I.ChannelId-set >< State' -> Bool
  channelsExist(ids, s) is ids <<= dom channels(s),

  getMember: I.MemberId >< State' -~-> E.Member
  getMember(i, s) is members(s)(i) pre memberExists(i, s),

  getChannel: I.ChannelId >< State' -~-> C.Channel
  getChannel(i, s) is channels(s)(i) pre channelExists(i, s),

  channelsOwned: I.MemberId >< State' -~-> I.ChannelId-set
  channelsOwned(im, s) is
    { ic | ic: I.ChannelId :- C.owner(getChannel(ic, s)) = im }
    pre memberExists(im, s),

  channelsSubscribed: I.MemberId >< State' -~-> I.ChannelId-set
  channelsSubscribed(im, s) is
    { ic | ic: I.ChannelId :- im isin C.subscribers(getChannel(ic, s)) }
    pre memberExists(im, s)

value
  iswfMessage: M.Message >< State' -> Bool
  iswfMessage(m, s) is
    let ms = M.sender(m), mr = getChannel(M.recepients(m), s) in
      memberExists(ms, s) /\
      channelExists(M.recepients(m), s) /\
      (C.isOwner(ms, mr) \/ C.isSubscriber(ms, mr))
    end,

  iswfQueue: E.Queue >< State' -> Bool
  iswfQueue(q, s) is
    E.isEmpty(q) \/ iswfMessage(hd q, s) /\ iswfQueue(tl q, s),

  iswfMember: I.MemberId >< State' -~-> Bool
  iswfMember(i, s) is
    let e = getMember(i, s) in
      iswfQueue(E.inbox(e), s) /\ iswfQueue(E.outbox(e), s) /\
      (all m: M.Message :-
        let ms = M.sender(m), mr = getChannel(M.recepients(m), s) in
          (E.isInInbox(m, e) => ms = i) /\
          (E.isInOutbox(m, e) => (C.isOwner(i, mr) \/ C.isSubscriber(i, mr)))
        end
      )
    end
    pre memberExists(i, s),

  iswfChannel: C.Channel >< State' -> Bool
  iswfChannel(c, s) is
    memberExists(C.owner(c), s) /\ membersExist(C.subscribers(c), s),

  iswfMembers: Members >< State' -> Bool
  iswfMembers(ms, s) is
    (all i: I.MemberId :- memberExists(i, s) => iswfMember(i, s)),

  iswfChannels: Channels >< State' -> Bool
  iswfChannels(cs, s) is
    (all i: I.ChannelId :- channelExists(i, s) => iswfChannel(getChannel(i, s), s)),

  iswfAdmin: State' -> Bool
  iswfAdmin(s) is
    memberExists(I.admin, s) /\
    (all mi: I.MemberId :-
      memberExists(mi, s) =>
      channelExists(I.adminChannel(mi), s) /\
      C.owner(getChannel(I.adminChannel(mi), s)) = I.admin /\
      C.subscribers(getChannel(I.adminChannel(mi), s)) = {mi}
    ),
```

```
  iswfVisitor: State' -> Bool
  iswfVisitor(s) is
    memberExists(I.visitor, s) /\
    (all ci: I.ChannelId :-
      channelExists(ci, s) =>
      I.visitor ~= C.owner(getChannel(ci, s)) /\
      (I.visitor isin C.subscribers(getChannel(ci, s))
      => ci = I.adminChannel(I.visitor))
    ),

  iswf: State' -> Bool
  iswf(s) is
    iswfMembers(members(s), s) /\ iswfChannels(channels(s), s) /\
    iswfAdmin(s) /\ iswfVisitor(s)

value
  addMember: I.MemberId >< E.Member >< State -~-> State
  addMember(i, e, s) is re_members(members(s) union [ i +> e ], s)
    pre ~memberExists(i, s),

  deleteMember: I.MemberId >< State -~-> State
  deleteMember(i, s) is re_members(members(s) \ {i}, s)
    pre memberExists(i, s),

  modifyMember: I.MemberId >< E.Member >< State -~-> State
  modifyMember(i, e, s) is re_members(members(s) !! [ i +> e ], s)
    pre memberExists(i, s)

value
  addChannel: I.ChannelId >< C.Channel >< State -~-> State
  addChannel(i, c, s) is re_channels(channels(s) union [ i +> c ], s)
    pre ~channelExists(i, s),

  deleteChannel: I.ChannelId >< State -~-> State
  deleteChannel(i, s) is re_channels(channels(s) \ {i}, s)
    pre channelExists(i, s),

  modifyChannel: I.ChannelId >< C.Channel >< State -~-> State
  modifyChannel(i, c, s) is re_channels(channels(s) !! [ i +> c ], s)
    pre channelExists(i, s)

end
```

Scheme 14: MODEL 3 - GATEWAY

```
ID, MESSAGE, MEMBER, CHANNEL, STATE

scheme GATEWAY(I:ID, M:MESSAGE(I), E:MEMBER(I,M), C:CHANNEL(I), S:STATE(I,M,E,C)) =
class

value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is
    let
      s1 = S.addMember(i, E.init, s),
      s2 = S.addChannel(I.adminChannel(i), S.initAdmin(i), s1)
    in s2 end pre canRegister(i, s),

  canRegister: I.MemberId >< S.State -> Bool
  canRegister(i, s) is ~S.memberExists(i, s)

value
  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is
    let s1 = S.deleteMember(i, s), s2 = S.deleteChannel(I.adminChannel(i), s1)
```

```
    in s2 end pre canUnregister(i, s),

  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is
    S.memberExists(i, s) /\ i ~= I.admin /\ i ~= I.visitor /\
    S.channelsOwned(i, s) = {} /\
    S.channelsSubscribed(i, s) = {I.adminChannel(i)}

value
  create: I.ChannelId >< I.MemberId >< S.State -~-> S.State
  create(ic, im, s) is S.addChannel(ic, C.init(im), s) pre canCreate(ic, im, s),

  canCreate: I.ChannelId >< I.MemberId >< S.State -> Bool
  canCreate(ic, im, s) is
    ~S.channelExists(ic, s) /\ S.memberExists(im, s) /\ im ~= I.visitor

value
  destroy: I.ChannelId >< S.State -~-> S.State
  destroy(ic, s) is S.deleteChannel(ic, s) pre canDestroy(ic, s),

  canDestroy: I.ChannelId >< S.State -> Bool
  canDestroy(ic, s) is
    S.channelExists(ic, s) /\ ~I.isAdminChannel(ic) /\
    C.noSubscribers(S.getChannel(ic, s))

value
  subscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribe(im, ic, s) is
    let c = C.subscribe(im, S.getChannel(ic, s))
    in S.modifyChannel(ic, c, s) end
    pre canSubscribe(im, ic, s),

  canSubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canSubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    ~C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~C.isOwner(im, S.getChannel(ic, s)) /\ im ~= I.visitor

value
  unsubscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  unsubscribe(im, ic, s) is
    let c = C.unsubscribe(im, S.getChannel(ic, s))
    in S.modifyChannel(ic, c, s) end
    pre canUnsubscribe(im, ic, s),

  canUnsubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canUnsubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~I.isAdminChannel(ic)

value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let i = M.sender(m), e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end
    pre canSend(m, s),

  canSend: M.Message >< S.State -> Bool
  canSend(m, s) is S.iswfMessage(m, s)

value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let (m, e) = E.dequeueOutbox(S.getMember(i, s)) in
      (m, S.modifyMember(i, e, s))
    end pre canReceive(i, s),
```

```
  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is ~E.isEmptyOutbox(S.getMember(i, s))

value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      ids = C.subscribers(S.getChannel(M.recepients(m), s))
    in transferAll(m, ids, s') end
    pre canTransfer(i, s),

  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let
          e = E.enqueueOutbox(m, S.getMember(i, s)),
          s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is
    S.membersExist(ids, s) /\ ids <<= C.subscribers(S.getChannel(M.recepients(m), s))

end
```

A.4. MODEL 4 – Distributing the State among Members

Scheme 15: MODEL 4 - MEMBER

```
ID, MESSAGE, QUEUE, CHANNEL

scheme MEMBER(I:ID, M:MESSAGE(I), C:CHANNEL(I)) = extend QUEUE(I, M) with class

type
  Member’::
    id: I.MemberId
    inbox: Queue <-> re_inbox
    outbox: Queue <-> re_outbox
    owned: I.ChannelId -m-> C.Channel <-> re_owned
    subscribed: I.ChannelId -m-> I.MemberId <-> re_subscribed,
  Member = {| m: Member' :- iswfMember(m) |}

value
  isEmptyInbox: Member’ -> Bool
  isEmptyInbox(e) is isEmpty(inbox(e)),

  isEmptyOutbox: Member’ -> Bool
  isEmptyOutbox(e) is isEmpty(outbox(e)),

  isInInbox: M.Message >< Member’ -> Bool
  isInInbox(m, e) is m isin elems inbox(e),

  isInOutbox: M.Message >< Member’ -> Bool
  isInOutbox(m, e) is m isin elems outbox(e)

value
  isChannelOwned: I.ChannelId >< Member’ -> Bool
  isChannelOwned(ic, e) is ic isin dom owned(e),

  isChannelSubscribed: I.ChannelId >< Member’ -> Bool
  isChannelSubscribed(ic, e) is ic isin dom subscribed(e),

  channelsOwned: Member’ -> I.ChannelId-set
  channelsOwned(e) is dom owned(e),

  channelsSubscribed: Member’ -> I.ChannelId-set
  channelsSubscribed(e) is dom subscribed(e),

  subscribers: I.ChannelId >< Member’ -~-> I.MemberId-set
  subscribers(ic, e) is C.subscribers(owned(e)(ic)) pre isChannelOwned(ic, e)

value
  iswfMessage: M.Message >< Member’ -> Bool
  iswfMessage(m, e) is
    let mr = M.recepients(m) in
      (isInInbox(m, e) \/ isInOutbox(m, e)) /\
      (isChannelOwned(mr, e) \/ isChannelSubscribed(mr, e))
    end,

  iswfInbox: Queue >< Member’ -> Bool
  iswfInbox(q, e) is
    isEmpty(q) \/
    M.sender(hd q) = id(e) /\ iswfMessage(hd q, e) /\ iswfInbox(tl q, e),

  iswfOutbox: Queue >< Member’ -> Bool
  iswfOutbox(q, e) is
    isEmpty(q) \/
    iswfMessage(hd q, e) /\ iswfOutbox(tl q, e),

  iswfChannels: Member’ -> Bool
  iswfChannels(e) is
```

```
    dom owned(e) inter dom subscribed(e) = {} /\
    isChannelSubscribed(I.adminChannel(id(e)), e) /\
    (all ic: I.ChannelId :- isChannelOwned(ic, e) => C.owner(owned(e)(ic)) = id(e)),

  iswfMember: Member' -> Bool
  iswfMember(e) is iswfInbox(inbox(e), e) /\ iswfOutbox(outbox(e), e) /\ iswfChannels(e)

value
  init: I.MemberId -> Member
  init(im) as e post
    id(e) = im /\ inbox(e) = init /\ outbox(e) = init /\
    owned(e) = [] /\ subscribed(e) = [ I.adminChannel(im) +> I.admin ]

value
  enqueueInbox: M.Message >< Member -~-> Member
  enqueueInbox(m, e) is
    re_inbox(enqueue(m, inbox(e)), e)
    pre canEnqueueInbox(m, e),

  canEnqueueInbox: M.Message >< Member -> Bool
  canEnqueueInbox(m, e) is iswfMessage(m, e)

value
  dequeueInbox: Member -~-> (M.Message >< Member)
  dequeueInbox(e) is
    let (m', q) = dequeue(inbox(e)) in (m', re_inbox(q, e)) end
    pre canDequeueInbox(e),

  canDequeueInbox: Member -> Bool
  canDequeueInbox(e) is ~isEmptyInbox(e)

value
  enqueueOutbox: M.Message >< Member -> Member
  enqueueOutbox(m, e) is
    re_outbox(enqueue(m, outbox(e)), e)
    pre canEnqueueOutbox(m, e),

  canEnqueueOutbox: M.Message >< Member -> Bool
  canEnqueueOutbox(m, e) is iswfMessage(m, e)

value
  dequeueOutbox: Member -~-> (M.Message >< Member)
  dequeueOutbox(e) is
    let (m', q) = dequeue(outbox(e)) in (m', re_outbox(q, e)) end
    pre ~canDequeueOutbox(e),

  canDequeueOutbox: Member -> Bool
  canDequeueOutbox(e) is ~isEmptyOutbox(e)

value
  addChannelOwned: I.ChannelId >< Member -~-> Member
  addChannelOwned(ic, e) is
    re_owned(owned(e) !! [ ic +> C.init(id(e)) ], e)
    pre canAddChannelOwned(ic, e),

  canAddChannelOwned: I.ChannelId >< Member -> Bool
  canAddChannelOwned(ic, e) is ~isChannelOwned(ic, e) /\ ~isChannelSubscribed(ic, e)

value
  deleteChannelOwned: I.ChannelId >< Member -~-> Member
  deleteChannelOwned(ic, e) is
    re_owned(owned(e) \ {ic}, e)
    pre canDeleteChannelOwned(ic, e) /\ ~I.isAdminChannel(ic),

  canDeleteChannelOwned: I.ChannelId >< Member -> Bool
  canDeleteChannelOwned(ic, e) is isChannelOwned(ic, e)
```

```
value
  addSubscriber: I.ChannelId >< I.MemberId >< Member -~-> Member
  addSubscriber(ic, im, e) is
    re_owned(owned(e) !! [ic +> C.subscribe(im, owned(e)(ic))], e)
    pre canAddSubscriber(ic, im, e),

  canAddSubscriber: I.ChannelId >< I.MemberId >< Member -> Bool
  canAddSubscriber(ic, im, e) is
    isChannelOwned(ic, e) /\ ~C.isSubscriber(im, owned(e)(ic))

value
  deleteSubscriber: I.ChannelId >< I.MemberId >< Member -~-> Member
  deleteSubscriber(ic, im, e) is
    re_owned(owned(e) !! [ic +> C.unsubscribe(im, owned(e)(ic))], e)
    pre canDeleteSubscriber(ic, im, e),

  canDeleteSubscriber: I.ChannelId >< I.MemberId >< Member -> Bool
  canDeleteSubscriber(ic, im, e) is
    isChannelOwned(ic, e) /\ C.isSubscriber(im, owned(e)(ic))

value
  addChannelSubscribed: I.ChannelId >< I.MemberId >< Member -~-> Member
  addChannelSubscribed(ic, io, e) is
    re_subscribed(subscribed(e) !! [ ic +> io ], e)
    pre canAddChannelSubscribed(ic, e),

  canAddChannelSubscribed: I.ChannelId >< Member -> Bool
  canAddChannelSubscribed(ic, e) is
    ~isChannelOwned(ic, e) /\ ~isChannelSubscribed(ic, e)

value
  deleteChannelSubscribed: I.ChannelId >< Member -~-> Member
  deleteChannelSubscribed(ic, e) is
    re_subscribed(subscribed(e) \ {ic}, e)
    pre canDeleteChannelSubscribed(ic, e) /\ ic ~= I.adminChannel(id(e)),

  canDeleteChannelSubscribed: I.ChannelId >< Member -> Bool
  canDeleteChannelSubscribed(ic, e) is isChannelSubscribed(ic, e)

end
```

Scheme 16: MODEL 4 - STATE

```
ID, MESSAGE, MEMBER, CHANNEL

scheme STATE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C)) = class

type
  State' = I.MemberId -m-> E.Member,
  State = {| s:State' :- iswf(s) |}

value
  init: State' = [I.admin +> E.init(I.admin), I.visitor +> E.init(I.visitor)]

value
  memberExists: I.MemberId >< State' -> Bool
  memberExists(i, s) is i isin dom s,

  membersExist: I.MemberId-set >< State' -> Bool
  membersExist(ids, s) is ids <<= dom s,

  getMember: I.MemberId >< State' -~-> E.Member
  getMember(im, s) is s(im) pre memberExists(im, s)

value
```

```
  channelExists: I.ChannelId >< State' -> Bool
  channelExists(ic, s) is
    (exists im: I.MemberId :-
      memberExists(im, s) /\ E.isChannelOwned(ic, getMember(im, s))
    ),

  getOwner: I.ChannelId >< State' -~-> I.MemberId
  getOwner(ic, s) as im post E.isChannelOwned(ic, getMember(im, s))
    pre channelExists(ic, s),

  getChannel: I.ChannelId >< State' -~-> C.Channel
  getChannel(ic, s) is E.owned(getMember(getOwner(ic, s), s))(ic)
    pre channelExists(ic, s)

value
  iswfChannel: C.Channel >< State' -> Bool
  iswfChannel(c, s) is memberExists(C.owner(c), s) /\ membersExist(C.subscribers(c), s),

  iswfMember: I.MemberId >< State' -> Bool
  iswfMember(im, s) is
    let e = getMember(im, s) in
      E.id(e) = im /\ E.iswfMember(e) /\
      (all ic: I.ChannelId :-
        (E.isChannelOwned(ic, e) => iswfChannel(getChannel(ic, s), s)) /\
        (E.isChannelSubscribed(ic, e) =>
          channelExists(ic, s) /\ getOwner(ic, s) = E.subscribed(e)(ic))
      )
    end,

  iswfMembers: State' -> Bool
  iswfMembers(s) is
    (all im: I.MemberId :- memberExists(im, s) => iswfMember(im, s)),

  iswfChannels: State' -> Bool
  iswfChannels(s) is
    (all im1, im2: I.MemberId :- im1 ~= im2 =>
      E.channelsOwned(getMember(im1, s)) inter E.channelsOwned(getMember(im2, s)) = {}
    ),

  iswfAdmin: State' -> Bool
  iswfAdmin(s) is
    memberExists(I.admin, s) /\
    (all mi: I.MemberId :- memberExists(mi, s) =>
      channelExists(I.adminChannel(mi), s) /\
      getOwner(I.adminChannel(mi), s) = I.admin /\
      C.subscribers(getChannel(I.adminChannel(mi), s)) = {mi}
    ),

  iswfVisitor: State' -> Bool
  iswfVisitor(s) is
    memberExists(I.visitor, s) /\
    (all ci: I.ChannelId :- channelExists(ci, s) =>
      I.visitor ~= getOwner(ci, s) /\
      (all mi: I.MemberId :-
        I.visitor isin E.subscribers(ci, getMember(mi, s))
        => ci = I.adminChannel(I.visitor)
      )
    ),

  iswf: State' -> Bool
  iswf(s) is iswfMembers(s) /\ iswfChannels(s) /\ iswfAdmin(s) /\ iswfVisitor(s)

value
  addMember: I.MemberId >< E.Member >< State -~-> State
  addMember(i, e, s) is s union [ i +> e ] pre ~memberExists(i, s),

  deleteMember: I.MemberId >< State -~-> State
```

```
  deleteMember(i, s) is s \ {i} pre memberExists(i, s),

  modifyMember: I.MemberId >< E.Member >< State -~-> State
  modifyMember(i, e, s) is s !! [ i +> e ] pre memberExists(i, s)

value
  addAdminChannel: I.MemberId >< State -~-> State
  addAdminChannel(im, s) is
    let
      e1 = getMember(I.admin, s),
      e2 = E.addChannelOwned(I.adminChannel(im), e1),
      e3 = E.addSubscriber(I.adminChannel(im), im, e2)
    in s !! [ I.admin +> e3 ] end
    pre ~channelExists(I.adminChannel(im), s),

  deleteAdminChannel: I.MemberId >< State -~-> State
  deleteAdminChannel(im, s) is
    let
      e1 = getMember(I.admin, s),
      e2 = E.deleteChannelOwned(I.adminChannel(im), e1),
      e3 = E.deleteSubscriber(I.adminChannel(im), im, e2)
    in s !! [ I.admin +> e3 ] end
    pre channelExists(I.adminChannel(im), s)

end
```

Scheme 17: MODEL 4 - GATEWAY

```
ID, MESSAGE, MEMBER, CHANNEL, STATE

scheme GATEWAY(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) =
class

value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is
    let
      s1 = S.addMember(i, E.init(i), s),
      s2 = S.addAdminChannel(i, s1)
    in s2 end
    pre canRegister(i, s),

  canRegister: I.MemberId >< S.State -> Bool
  canRegister(i, s) is
    ~S.memberExists(i, s)

value
  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is
    let
      s1 = S.deleteMember(i, s),
      s2 = S.deleteAdminChannel(i, s1)
    in s2 end
    pre canUnregister(i, s),

  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is
    S.memberExists(i, s) /\
    i ~= I.admin /\ i ~= I.visitor /\
    E.channelsOwned(S.getMember(i, s)) = {} /\
    E.channelsSubscribed(S.getMember(i, s)) = {I.adminChannel(i)}

value
  create: I.ChannelId >< I.MemberId >< S.State -~-> S.State
  create(ic, im, s) is
```

```
    S.modifyMember(im, E.addChannelOwned(ic, S.getMember(im, s)), s)
    pre canCreate(ic, im, s),

  canCreate: I.ChannelId >< I.MemberId >< S.State -> Bool
  canCreate(ic, im, s) is
    ~S.channelExists(ic, s) /\
    S.memberExists(im, s) /\
    im ~= I.visitor

value
  destroy: I.ChannelId >< S.State -~-> S.State
  destroy(ic, s) is
    let io = S.getOwner(ic, s)
    in S.modifyMember(io, E.deleteChannelOwned(ic, S.getMember(io, s)), s) end
    pre canDestroy(ic, s),

  canDestroy: I.ChannelId >< S.State -> Bool
  canDestroy(ic, s) is
    S.channelExists(ic, s) /\
    ~I.isAdminChannel(ic) /\
    C.noSubscribers(S.getChannel(ic, s))

value
  subscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribe(im, ic, s) is
    let
      io = S.getOwner(ic, s),
      eo = E.addSubscriber(ic, im, S.getMember(io, s)),
      es = E.addChannelSubscribed(ic, io, S.getMember(im, s))
    in S.modifyMember(io, eo, S.modifyMember(im, es, s)) end
    pre canSubscribe(im, ic, s),

  canSubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canSubscribe(im, ic, s) is
    S.memberExists(im, s) /\
    S.channelExists(ic, s) /\
    ~C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~C.isOwner(im, S.getChannel(ic, s)) /\
    im ~= I.visitor

value
  unsubscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  unsubscribe(im, ic, s) is
    let
      io = S.getOwner(ic, s),
      eo = E.deleteSubscriber(ic, im, S.getMember(io, s)),
      es = E.deleteChannelSubscribed(ic, S.getMember(im, s))
    in S.modifyMember(io, eo, S.modifyMember(im, es, s)) end
    pre canUnsubscribe(im, ic, s),

  canUnsubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canUnsubscribe(im, ic, s) b
    S.memberExists(im, s) /\
    S.channelExists(ic, s) /\
    C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~I.isAdminChannel(ic)

value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let
      i = M.sender(m),
      e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end
    pre canSend(m, s),

  canSend: M.Message >< S.State -> Bool
```

```
  canSend(m, s) is E.iswfMessage(m, S.getMember(M.sender(m), s))

value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let (m, e) = E.dequeueOutbox(S.getMember(i, s)) in
      (m, S.modifyMember(i, e, s))
    end
    pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is
    ~E.isEmptyOutbox(S.getMember(i, s))

value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      ids = C.subscribers(S.getChannel(M.recepients(m), s))
    in transferAll(m, ids, s') end
    pre canTransfer(i, s),

  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is
    ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let
          e = E.enqueueOutbox(m, S.getMember(i, s)),
          s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end
    pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is
    S.membersExist(ids, s) /\
    ids <<= C.subscribers(S.getChannel(M.recepients(m), s))

end
```

A.5. MODEL 5 – Message Types and All-Inclusive Messaging

Scheme 18: MODEL 5 - MESSAGE

```
ID

scheme MESSAGE(I:ID) = class

type
  MessageType ==
    register(I.MemberId) |
    unregister(I.MemberId) |
    create(I.ChannelId, I.MemberId) |
    destroy(I.ChannelId, I.MemberId) |
    subscribe(I.MemberId, I.ChannelId) |
    unsubscribe(I.MemberId, I.ChannelId) | _

type
  Message::
    mtype: MessageType
    sender: I.MemberId
    recepients: I.ChannelId
    error: Bool <-> re_error

end
```

Scheme 19: MODEL 5 - OPERATIONS

```
ID, MESSAGE, CHANNEL, MEMBER, STATE

scheme OPERATIONS(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E))=
class

value
  register: M.Message >< S.State -~-> S.State,
  unregister: M.Message >< S.State -~-> S.State,
  create: M.Message >< S.State -~-> S.State,
  destroy: M.Message >< S.State -~-> S.State,
  subscribe: M.Message >< S.State -~-> S.State,
  unsubscribe: M.Message >< S.State -~-> S.State,

  canRegister: M.Message >< S.State -> Bool,
  canUnregister: M.Message >< S.State -> Bool,
  canCreate: M.Message >< S.State -> Bool,
  canDestroy: M.Message >< S.State -> Bool,
  canSubscribe: M.Message >< S.State -> Bool,
  canUnsubscribe: M.Message >< S.State -> Bool

end
```

Scheme 20: MODEL 5 - GATEWAY

```
ID, MESSAGE, MEMBER, CHANNEL, STATE, OPERATIONS

scheme GATEWAY(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E),
O:OPERATIONS(I,M,C,E,S)) = class

value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let
      i = M.sender(m), e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end
    pre canSend(m, s),
```

```
  canSend: M.Message >< S.State -> Bool
  canSend(m, s) is E.iswfMessage(m, S.getMember(M.sender(m), s))

value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let
      (m, e) = E.dequeueOutbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s)
    in
      case M.mtype(m) of
        M.register(i) ->
          if O.canRegister(m, s')
          then (m, O.register(m, s')) else (M.re_error(true, m), s') end,
        M.unregister(i) ->
          if O.canUnregister(m, s')
          then (m, O.unregister(m, s')) else (M.re_error(true, m), s') end,
        M.create(ic, im) ->
          if O.canCreate(m, s')
          then (m, O.create(m, s')) else (M.re_error(true, m), s') end,
        M.destroy(ic, im) ->
          if O.canDestroy(m, s')
          then (m, O.destroy(m, s')) else (M.re_error(true, m), s') end,
        M.subscribe(im, ic) ->
          if O.canSubscribe(m, s')
          then (m, O.subscribe(m, s')) else (M.re_error(true, m), s') end,
        M.unsubscribe(im, ic) ->
          if O.canUnsubscribe(m, s')
          then (m, O.unsubscribe(m, s')) else (M.re_error(true, m), s') end
      end
    end pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is ~E.isEmptyOutbox(S.getMember(i, s))

value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      ids = C.subscribers(S.getChannel(M.recepients(m), s))
    in transferAll(m, ids, s') end
    pre canTransfer(i, s),

  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let
          e = E.enqueueOutbox(m, S.getMember(i, s)),
          s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is
    S.membersExist(ids, s) /\ ids <<= C.subscribers(S.getChannel(M.recepients(m), s))

end
```

Scheme 21: MODEL 5 - INTERFACE

```
ID, MESSAGE, MEMBER, CHANNEL, STATE, GATEWAY, OPERATIONS

scheme INTERFACE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) =
class

object
   O: class
        value
          register: M.Message >< S.State -~-> S.State = registerReceive,
          unregister: M.Message >< S.State -~-> S.State = unregisterReceive,
          create: M.Message >< S.State -~-> S.State = createReceive,
          destroy: M.Message >< S.State -~-> S.State = destroyReceive,
          subscribe: M.Message >< S.State -~-> S.State = subscribeReceive,
          unsubscribe: M.Message >< S.State -~-> S.State = unsubscribeReceive
        value
          canRegister: M.Message >< S.State -> Bool = canRegisterReceive,
          canUnregister: M.Message >< S.State -> Bool = canUnregisterReceive,
          canCreate: M.Message >< S.State -> Bool = canCreateReceive,
          canDestroy: M.Message >< S.State -> Bool = canDestroyReceive,
          canSubscribe: M.Message >< S.State -> Bool = canSubscribeReceive,
          canUnsubscribe: M.Message >< S.State -> Bool = canUnsubscribeReceive
      end,
    G: GATEWAY(I,M,C,E,S,O)

value
  registerSend: I.MemberId >< S.State -~-> S.State
  registerSend(i, s) is
    let
      mtype = M.register(i),
      sender = I.visitor,
      recepients = I.adminChannel(I.visitor),
      error = false
    in G.send(M.mk_Message(mtype, sender, recepients, error), s) end
    pre canRegisterSend(i, s),

  canRegisterSend: I.MemberId >< S.State -> Bool
  canRegisterSend(i, s) is
    ~S.memberExists(i, s),

  registerReceive: M.Message >< S.State -~-> S.State
  registerReceive(m, s) is
    case M.mtype(m) of
      M.register(i) ->
        let
          s1 = S.addMember(i, E.init(i), s),
          s2 = S.addAdminChannel(i, s1)
        in s2 end
    end
    pre canRegisterReceive(m, s),

  canRegisterReceive: M.Message >< S.State -> Bool
  canRegisterReceive(m, s) is
    case M.mtype(m) of
      M.register(i) -> ~S.memberExists(i, s),
      _ -> false
    end

value
  unregisterSend: I.MemberId >< S.State -~-> S.State
  unregisterSend(i, s) is
    let
      mtype = M.unregister(i),
      sender = I.visitor,
      recepients = I.adminChannel(I.visitor),
      error = false
```

```
    in G.send(M.mk_Message(mtype, sender, recepients, error), s) end
    pre canUnregisterSend(i, s),

  canUnregisterSend: I.MemberId >< S.State -> Bool
  canUnregisterSend(i, s) is
    S.memberExists(i, s) /\
    i ~= I.admin /\ i ~= I.visitor /\
    E.channelsOwned(S.getMember(i, s)) = {} /\
    E.channelsSubscribed(S.getMember(i, s)) = {I.adminChannel(i)},

  unregisterReceive: M.Message >< S.State -~-> S.State
  unregisterReceive(m, s) is
    case M.mtype(m) of
      M.unregister(i) ->
        let
          s1 = S.deleteMember(i, s),
          s2 = S.deleteAdminChannel(i, s1)
        in s2 end
    end
    pre canUnregisterReceive(m, s),

  canUnregisterReceive: M.Message >< S.State -> Bool
  canUnregisterReceive(m, s) is
    case M.mtype(m) of
      M.unregister(i) ->
        S.memberExists(i, s) /\
        i ~= I.admin /\ i ~= I.visitor /\
        E.channelsOwned(S.getMember(i, s)) = {} /\
        E.channelsSubscribed(S.getMember(i, s)) = {I.adminChannel(i)},
      _ -> false
    end

value
  createSend: I.ChannelId >< I.MemberId >< S.State -~-> S.State
  createSend(ic, im, s) is
    let
      mtype = M.create(ic, im),
      sender = im,
      recepients = I.adminChannel(im),
      error = false
    in G.send(M.mk_Message(mtype, sender, recepients, error), s) end
    pre canCreateSend(ic, im, s),

  canCreateSend: I.ChannelId >< I.MemberId >< S.State -> Bool
  canCreateSend(ic, im, s) is
    ~S.channelExists(ic, s) /\
    S.memberExists(im, s) /\
    im ~= I.visitor,

  createReceive: M.Message >< S.State -~-> S.State
  createReceive(m, s) is
    case M.mtype(m) of
      M.create(ic, im) ->
        let e = E.addChannelOwned(ic, S.getMember(im, s))
        in S.modifyMember(im, e, s) end
    end
    pre canCreateReceive(m, s),

  canCreateReceive: M.Message >< S.State -> Bool
  canCreateReceive(m, s) is
    case M.mtype(m) of
      M.create(ic, im) ->
        ~S.channelExists(ic, s) /\
        S.memberExists(im, s) /\
        im ~= I.visitor,
      _ -> false
    end
```

```
value
  destroySend: I.ChannelId >< I.MemberId >< S.State -~-> S.State
  destroySend(ic, im, s) is
    let
      mtype = M.destroy(ic, im),
      sender = im,
      recepients = I.adminChannel(im),
      error = false
    in G.send(M.mk_Message(mtype, sender, recepients, error), s) end
    pre canDestroySend(ic, im, s),

  canDestroySend: I.ChannelId >< I.MemberId >< S.State -> Bool
  canDestroySend(ic, im, s) is
    S.channelExists(ic, s) /\
    im = S.getOwner(ic, s) /\
    ~I.isAdminChannel(ic) /\
    C.noSubscribers(S.getChannel(ic, s)),

  destroyReceive: M.Message >< S.State -~-> S.State
  destroyReceive(m, s) is
    case M.mtype(m) of
      M.destroy(ic, im) ->
        let e = E.deleteChannelOwned(ic, S.getMember(im, s))
        in S.modifyMember(im, e, s) end
    end
    pre canDestroyReceive(m, s),

  canDestroyReceive: M.Message >< S.State -> Bool
  canDestroyReceive(m, s) is
    case M.mtype(m) of
      M.destroy(ic, im) ->
        S.channelExists(ic, s) /\
        im = S.getOwner(ic, s) /\
        ~I.isAdminChannel(ic) /\
        C.noSubscribers(S.getChannel(ic, s)),
      _ -> false
    end

value
  subscribeSend: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribeSend(im, ic, s) is
    let
      mtype = M.subscribe(im, ic),
      sender = im,
      recepients = I.adminChannel(im),
      error = false
    in G.send(M.mk_Message(mtype, sender, recepients, error), s) end
    pre canSubscribeSend(im, ic, s),

  canSubscribeSend: I.MemberId >< I.ChannelId >< S.State -> Bool
  canSubscribeSend(im, ic, s) is
    S.memberExists(im, s) /\
    S.channelExists(ic, s) /\
    ~C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~C.isOwner(im, S.getChannel(ic, s)) /\
    im ~= I.visitor,

  subscribeReceive: M.Message >< S.State -~-> S.State
  subscribeReceive(m, s) is
    case M.mtype(m) of
      M.subscribe(im, ic) ->
      let
        io = S.getOwner(ic, s),
        eo = E.addSubscriber(ic, im, S.getMember(io, s)),
        es = E.addChannelSubscribed(ic, io, S.getMember(im, s))
      in S.modifyMember(io, eo, S.modifyMember(im, es, s)) end
```

```
    end
    pre canSubscribeReceive(m, s),

  canSubscribeReceive: M.Message >< S.State -> Bool
  canSubscribeReceive(m, s) is
    case M.mtype(m) of
      M.subscribe(im, ic) ->
        S.memberExists(im, s) /\
        S.channelExists(ic, s) /\
        ~C.isSubscriber(im, S.getChannel(ic, s)) /\
        ~C.isOwner(im, S.getChannel(ic, s)) /\
        im ~= I.visitor,
      _ -> false
    end

value
  unsubscribeSend: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  unsubscribeSend(im, ic, s) is
    let
      mtype = M.unsubscribe(im, ic),
      sender = im,
      recepients = I.adminChannel(im),
      error = false
    in G.send(M.mk_Message(mtype, sender, recepients, error), s) end
    pre canUnsubscribeSend(im, ic, s),

  canUnsubscribeSend: I.MemberId >< I.ChannelId >< S.State -> Bool
  canUnsubscribeSend(im, ic, s) is
    S.memberExists(im, s) /\
    S.channelExists(ic, s) /\
    C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~I.isAdminChannel(ic),

  unsubscribeReceive: M.Message >< S.State -~-> S.State
  unsubscribeReceive(m, s) is
    case M.mtype(m) of
      M.unsubscribe(im, ic) ->
      let
        io = S.getOwner(ic, s),
        eo = E.deleteSubscriber(ic, im, S.getMember(io, s)),
        es = E.deleteChannelSubscribed(ic, S.getMember(im, s))
      in S.modifyMember(io, eo, S.modifyMember(im, es, s)) end
    end
    pre canUnsubscribeReceive(m, s),

  canUnsubscribeReceive: M.Message >< S.State -> Bool
  canUnsubscribeReceive(m, s) is
    case M.mtype(m) of
      M.unsubscribe(im, ic) ->
        S.memberExists(im, s) /\
        S.channelExists(ic, s) /\
        C.isSubscriber(im, S.getChannel(ic, s)) /\
        ~I.isAdminChannel(ic),
      _ -> false
    end

end
```

A.6. MODEL 6 – Messaging through Channel Owners

Scheme 22: MODEL 6 - MESSAGE

```
ID

scheme MESSAGE(I:ID) = class

type
  MessageBody
value
  empty: MessageBody

type
  MessageType ==
    register(I.MemberId) |
    unregister(I.MemberId) |
    create(I.ChannelId, I.MemberId) |
    destroy(I.ChannelId, I.MemberId) |
    subscribe(I.MemberId, I.ChannelId) |
    unsubscribe(I.MemberId, I.ChannelId) |
    forward(I.MemberId) |
    other |

type
  Message::
    mtype: MessageType
    sender: I.MemberId
    recepients: I.ChannelId
    error: Bool <-> re_error
    body: MessageBody

end
```

Scheme 23: MODEL 6 - OPERATIONS

```
ID, MESSAGE, CHANNEL, MEMBER, STATE

scheme OPERATIONS(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E))=
class

value
  register: I.MemberId >< M.Message >< S.State -~-> S.State,
  unregister: I.MemberId >< M.Message >< S.State -~-> S.State,
  create: I.MemberId >< M.Message >< S.State -~-> S.State,
  destroy: I.MemberId >< M.Message >< S.State -~-> S.State,
  subscribe: I.MemberId >< M.Message >< S.State -~-> S.State,
  unsubscribe: I.MemberId >< M.Message >< S.State -~-> S.State,
  forward: I.MemberId >< M.Message >< S.State -~-> S.State,
  other: I.MemberId >< M.Message >< S.State -~-> S.State

value
  canRegister: I.MemberId >< M.Message >< S.State -> Bool,
  canUnregister: I.MemberId >< M.Message >< S.State -> Bool,
  canCreate: I.MemberId >< M.Message >< S.State -> Bool,
  canDestroy: I.MemberId >< M.Message >< S.State -> Bool,
  canSubscribe: I.MemberId >< M.Message >< S.State -> Bool,
  canUnsubscribe: I.MemberId >< M.Message >< S.State -> Bool,
  canForward: I.MemberId >< M.Message >< S.State -> Bool,
  canOther: I.MemberId >< M.Message >< S.State -> Bool

end
```

Scheme 24: MODEL 6 - GATEWAY

```
ID, MESSAGE, MEMBER, CHANNEL, STATE, OPERATIONS

scheme GATEWAY(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E),
O:OPERATIONS(I,M,C,E,S)) = class

value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let
      i = M.sender(m),
      e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end
    pre canSend(m, s),

  canSend: M.Message >< S.State -> Bool
  canSend(m, s) is E.iswfMessage(m, S.getMember(M.sender(m), s))

value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let
      (m, e) = E.dequeueOutbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s)
    in
      case M.mtype(m) of
        M.register(i) ->
          if O.canRegister(i, m, s')
          then (m, O.register(i, m, s')) else (M.re_error(true, m), s') end,
        M.unregister(i) ->
          if O.canUnregister(i, m, s')
          then (m, O.unregister(i, m, s')) else (M.re_error(true, m), s') end,
        M.create(ic, im) ->
          if O.canCreate(i, m, s')
          then (m, O.create(i, m, s')) else (M.re_error(true, m), s') end,
        M.destroy(ic, im) ->
          if O.canDestroy(i, m, s')
          then (m, O.destroy(i, m, s')) else (M.re_error(true, m), s') end,
        M.subscribe(im, ic) ->
          if O.canSubscribe(i, m, s')
          then (m, O.subscribe(i, m, s')) else (M.re_error(true, m), s') end,
        M.unsubscribe(im, ic) ->
          if O.canUnsubscribe(i, m, s')
          then (m, O.unsubscribe(i, m, s')) else (M.re_error(true, m), s') end,
        M.forward(im) ->
          if O.canForward(i, m, s')
          then (m, O.forward(i, m, s')) else (M.re_error(true, m), s') end,
        M.other ->
          if O.canOther(i, m, s')
          then (m, O.other(i, m, s')) else (M.re_error(true, m), s') end
      end
    end
    pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is ~E.isEmptyOutbox(S.getMember(i, s))

value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      io = S.getOwner(M.recepients(m), s)
    in
      if i ~= io then transferAll(m, {io}, s') else
```

```
        let ids = C.subscribers(S.getChannel(M.recepients(m), s))
        in transferAll(m, ids, s') end
      end
    end
    pre canTransfer(i, s),

  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let
          e = E.enqueueOutbox(m, S.getMember(i, s)),
          s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end
    pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is
    S.membersExist(ids, s) /\ ids <<= C.subscribers(S.getChannel(M.recepients(m), s))

end
```

Scheme 25: MODEL 6 - INTERFACE

```
ID, MESSAGE, MEMBER, CHANNEL, STATE, GATEWAY, OPERATIONS

scheme INTERFACE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) =
class

object
   O: class ... end,
   G: GATEWAY(I,M,C,E,S,O)

...

value
  forwardSend: I.MemberId >< I.MemberId >< M.MessageBody >< S.State -~-> S.State
  forwardSend(i1, i2, b, s) is
    let
      mtype = M.forward(i2),
      sender = i1,
      recepients = I.adminChannel(i1)
    in G.send(M.mk_Message(mtype, sender, recepients, false, b), s) end
    pre canForwardSend(i1, i2, b, s),

  canForwardSend: I.MemberId >< I.MemberId >< M.MessageBody >< S.State -> Bool
  canForwardSend(i1, i2, b, s) is
    S.memberExists(i1, s) /\
    S.memberExists(i2, s),

  forwardReceive: I.MemberId >< M.Message >< S.State -~-> S.State
  forwardReceive(ir, m, s) is
    case M.mtype(m) of
      M.forward(i) ->
        let
          mtype = M.other,
          sender = I.admin,
          recepients = I.adminChannel(i)
        in G.send(M.mk_Message(mtype, sender, recepients, false, M.body(m)), s) end
```

```
    end
    pre canForwardReceive(ir, m, s),

  canForwardReceive: I.MemberId >< M.Message >< S.State -> Bool
  canForwardReceive(ir, m, s) is
    case M.mtype(m) of
      M.forward(i) -> S.memberExists(i, s) /\ ir = I.admin,
      _ -> false
    end

value
  otherSend: I.MemberId >< I.ChannelId >< M.MessageBody >< S.State -~-> S.State
  otherSend(im, ic, b, s) is
    let
      mtype = M.other,
      sender = im,
      recepients = ic
    in G.send(M.mk_Message(mtype, sender, recepients, false, b), s) end
    pre canOtherSend(im, ic, b, s),

  canOtherSend: I.MemberId >< I.ChannelId >< M.MessageBody >< S.State -> Bool
  canOtherSend(im, ic, b, s) is
    S.memberExists(im, s) /\
    S.channelExists(ic, s) /\
    (C.isOwner(im, S.getChannel(ic, s)) \/ C.isSubscriber(im, S.getChannel(ic, s))),

  otherReceive: I.MemberId >< M.Message >< S.State -~-> S.State
  otherReceive(ir, m, s) is
    let
      ic = M.recepients(m),
      io = S.getOwner(ic, s)
    in
      if ir ~= io then s else
        G.send(M.mk_Message(M.mtype(m), ir, ic, false, M.body(m)), s)
      end
    end
    pre canOtherReceive(ir, m, s),

  canOtherReceive: I.MemberId >< M.Message >< S.State -> Bool
  canOtherReceive(ir, m, s) is
    let ic = M.recepients(m) in
      S.memberExists(ir, s) /\
      S.channelExists(ic, s) /\
      (C.isOwner(ir, S.getChannel(ic, s)) \/ C.isSubscriber(ir, S.getChannel(ic, s)))
    end

end
```

A.7. MODEL 7 – Type-Driven Programmable Messaging

Scheme 26: MODEL 7 - ACTION

```
ID, MESSAGE, CHANNEL, MEMBER, STATE, OPERATIONS, GATEWAY

scheme ACTION(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E),
O:OPERATIONS(I,M,C,E,S), G:GATEWAY(I,M,C,E,S,O)) = class

type
  Action ==
    addNewMember(I.MemberId) |
    addAdminChannel(I.MemberId) |
    deleteMember(I.MemberId) |
    deleteAdminChannel(I.MemberId) |
    addChannelOwned(I.ChannelId, I.MemberId) |
    deleteChannelOwned(I.ChannelId, I.MemberId) |
    addSubscriber(I.ChannelId, I.MemberId) |
    addChannelSubscribed(I.ChannelId, I.MemberId) |
    deleteSubscriber(I.ChannelId, I.MemberId) |
    deleteChannelSubscribed(I.ChannelId, I.MemberId) |
    send(M.MessageType, I.MemberId, I.ChannelId, Bool) |
    cond(O.Predicate, Action, Action) |
    seq(Action, Action)

value
  exec: Action >< I.MemberId >< M.Message >< S.State -~-> S.State
  exec(a, ir, m, s) is
    case a of
      addNewMember(im) -> S.addMember(im, E.init(im), s),
      addAdminChannel(im) -> S.addAdminChannel(im, s),
      deleteMember(im) -> S.deleteMember(im, s),
      deleteAdminChannel(im) -> S.deleteAdminChannel(im, s),
      addChannelOwned(ic, im) -> S.addChannelOwned(ic, im, s),
      deleteChannelOwned(ic, im) -> S.deleteChannelOwned(ic, im, s),
      addSubscriber(ic, im) -> S.addSubscriber(ic, im, s),
      addChannelSubscribed(ic, im) -> S.addChannelSubscribed(ic, im, s),
      deleteSubscriber(ic, im) -> S.deleteSubscriber(ic, im, s),
      deleteChannelSubscribed(ic, im) -> S.deleteChannelSubscribed(ic, im, s),
      send(mt, im, ic, b) -> G.send(M.mk_Message(mt, im, ic, b, M.body(m)), s),
      cond(p, a1, a2) ->
        if O.exec(p, ir, m, s)
        then exec(a1, ir, m, s) else exec(a2, ir, m, s) end,
      seq(a1, a2) -> let s' = exec(a1, ir, m, s) in exec(a2, ir, m, s') end
    end
    pre O.exec(iswf(a, ir, m, s), ir, m, s),

  iswf: Action >< I.MemberId >< M.Message >< S.State -~-> O.Predicate
  iswf(a, ir, m, s) is
    case a of
      addNewMember(im) -> O.not(O.memberExists(im)),
      addAdminChannel(im) -> O.not(O.channelExists(I.adminChannel(im))),
      deleteMember(im) -> O.memberExists(im),
      deleteAdminChannel(im) -> O.channelExists(I.adminChannel(im)),
      addChannelOwned(ic, im) -> O.not(O.channelExists(ic)),
      deleteChannelOwned(ic, im) -> O.isChannelOwner(im, ic),
      addSubscriber(ic, im) ->
        O.and(O.channelExists(ic), O.not(O.isChannelSubscriber(im, ic))),
      addChannelSubscribed(ic, im) -> O.andAll(<.
        O.channelExists(ic),
        O.not(O.isChannelOwner(im, ic)),
        O.not(O.isChannelSubscriber(im, ic)).>),
      deleteSubscriber(ic, im) ->
        O.and(O.channelExists(ic), O.isChannelSubscriber(im, ic)),
      deleteChannelSubscribed(ic, im) -> O.isChannelSubscribed(ic, im),
      send(mt, im, ic, b) -> O.andAll(<.
```

```
        O.memberExists(im), O.channelExists(ic),
        O.or(O.isChannelOwner(im, ic), O.isChannelSubscriber(im, ic)).>),
      cond(p, a1, a2) -> O.and(iswf(a1, ir, m, s), iswf(a2, ir, m, s)),
      seq(a1, a2) -> O.and(iswf(a1, ir, m, s), iswf(a2, ir, m, exec(a1, ir, m, s)))
    end

value
  getAction: I.MemberId >< M.Message -> Action
  getAction(ir, m) is
    case M.mtype(m) of
      M.register(im) -> seq(addNewMember(im), addAdminChannel(im)),
      M.unregister(im) -> seq(deleteMember(im), deleteAdminChannel(im)),
      M.create(ic, im) -> addChannelOwned(ic, im),
      M.destroy(ic, im) -> deleteChannelOwned(ic, im),
      M.subscribe(im, ic) -> seq(addSubscriber(ic, im), addChannelSubscribed(ic, im)),
      M.unsubscribe(im, ic) ->
        seq(deleteSubscriber(ic, im), deleteChannelSubscribed(ic, im)),
      M.forward(im) -> send(M.other, I.admin, I.adminChannel(im), false),
      M.other -> send(M.other, ir, M.recepients(m), false)
    end

end
```

Scheme 27: MODEL 7 - PREDICATE

```
ID, MESSAGE, CHANNEL, MEMBER, STATE, OPERATIONS, GATEWAY

scheme PREDICATE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E),
O:OPERATIONS(I,M,C,E,S), G:GATEWAY(I,M,C,E,S,O)) = class

type
  Predicate ==
    cons(Bool) |
    isReceiverAdmin |
    memberExists(I.MemberId) |
    channelExists(I.ChannelId) |
    isMemberAdmin(I.MemberId) |
    isMemberVisitor(I.MemberId) |
    isChannelOwner(I.MemberId, I.ChannelId) |
    isChannelSubscriber(I.MemberId, I.ChannelId) |
    isChannelSubscribed(I.ChannelId, I.MemberId) |
    isAdminChannel(I.ChannelId) |
    not(Predicate) |
    and(Predicate, Predicate) |
    or(Predicate, Predicate) |
    andAll(Predicate-list) |
    equal(O.Number, O.Number)

value
  exec: Predicate >< I.MemberId >< M.Message >< S.State -> Bool
  exec(p, ir, m, s) is
    case p of
      cons(b) -> b,
      isReceiverAdmin -> ir = I.admin,
      memberExists(im) -> S.memberExists(im, s),
      isMemberAdmin(im) -> im = I.admin,
      isMemberVisitor(im) -> im = I.visitor,
      isChannelOwner(im, ic) -> S.isChannelOwned(ic, im, s),
      isChannelSubscriber(im, ic) -> S.isSubscriber(im, ic, s),
      isChannelSubscribed(ic, im) -> S.isChannelSubscribed(ic, im, s),
      isAdminChannel(ic) -> I.isAdminChannel(ic),
      not(p') -> ~exec(p', ir, m, s),
      and(p1, p2) -> exec(p1, ir, m, s) /\ exec(p2, ir, m, s),
      or(p1, p2) -> exec(p1, ir, m, s) \/ exec(p2, ir, m, s),

      andAll(pl) ->
```

```
        pl = <..> \/ exec(hd pl, ir, m, s) /\ exec(andAll(tl pl), ir, m, s),
      equal(n1, n2) -> O.exec(n1, ir, m, s) = O.exec(n2, ir, m, s)
    end

value
  getPrecondition: I.MemberId >< M.Message -> Predicate
  getPrecondition(ir, m) is
    case M.mtype(m) of
      M.register(im) -> and(not(memberExists(im)), isReceiverAdmin),
      M.unregister(im) ->
        andAll(<. isReceiverAdmin, memberExists(im), not(isMemberAdmin(im)),
          not(isMemberVisitor(im)), equal(O.channelsOwned(im), O.constant(0)),
          equal(O.channelsSubscribed(im), O.constant(1)).>),
      M.create(ic, im) ->
        andAll(<.isReceiverAdmin, not(channelExists(ic)), memberExists(im),
          not(isMemberVisitor(im)).>),
      M.destroy(ic, im) ->
        andAll(<.isReceiverAdmin, channelExists(ic), isChannelOwner(im, ic),
          not(isAdminChannel(ic)), equal(O.channelSubscribers(ic), O.constant(0)).>),
      M.subscribe(im, ic) ->
        andAll(<.isReceiverAdmin, memberExists(im), channelExists(ic),
          not(isChannelSubscriber(im, ic)), not(isChannelOwner(im,ic)),
          not(isMemberVisitor(im)).>),
      M.unsubscribe(im, ic) ->
        andAll(<.isReceiverAdmin, memberExists(im), channelExists(ic),
          isChannelSubscriber(im, ic), not(isAdminChannel(ic)).>),
      M.forward(im) ->
        and(memberExists(im), isReceiverAdmin),
      M.other ->
        andAll(<.memberExists(ir), channelExists(M.recepients(m)),
          or(isChannelOwner(ir, M.recepients(m)),
          isChannelSubscriber(ir, M.recepients(m))).>)
    end

end
```

Scheme 28: MODEL 7 - NUMBER

```
ID, MESSAGE, CHANNEL, MEMBER, STATE, OPERATIONS, GATEWAY

scheme NUMBER(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E),
O:OPERATIONS(I,M,C,E,S), G:GATEWAY(I,M,C,E,S,O)) = class

type
  Number ==
    constant(Nat) |
    channelsOwned(I.MemberId) |
    channelsSubscribed(I.MemberId) |
    channelSubscribers(I.ChannelId) |
    add(Number, Number)

value
  exec: Number >< I.MemberId >< M.Message >< S.State -> Nat
  exec(n, ir, m, s) is
    case n of
      constant(m) -> m,
      channelsOwned(im) -> card S.channelsOwned(im, s),
      channelsSubscribed(im) -> card S.channelsSubscribed(im, s),
      channelSubscribers(ic) -> card S.subscribers(ic, s),
      add(n1, n2) -> exec(n1, ir, m, s) + exec(n2, ir, m, s)
    end

end
```

Scheme 29: MODEL 7 - OPERATIONS

```
ID, MESSAGE, CHANNEL, MEMBER, STATE

scheme OPERATIONS(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) =
class

type
  Action ==
    addNewMember(I.MemberId) |
    addAdminChannel(I.MemberId) |
    deleteMember(I.MemberId) |
    deleteAdminChannel(I.MemberId) |
    addChannelOwned(I.ChannelId, I.MemberId) |
    deleteChannelOwned(I.ChannelId, I.MemberId) |
    addSubscriber(I.ChannelId, I.MemberId) |
    addChannelSubscribed(I.ChannelId, I.MemberId) |
    deleteSubscriber(I.ChannelId, I.MemberId) |
    deleteChannelSubscribed(I.ChannelId, I.MemberId) |
    send(M.MessageType, I.MemberId, I.ChannelId, Bool) |
    seq(Action, Action)

value
  exec: Action >< I.MemberId >< M.Message >< S.State -~-> S.State,
  iswf: Action >< I.MemberId >< M.Message >< S.State -> Bool,
  getAction: I.MemberId >< M.Message -> Action

type
  Predicate ==
    isReceiverAdmin |
    memberExists(I.MemberId) |
    channelExists(I.ChannelId) |
    isMemberAdmin(I.MemberId) |
    isMemberVisitor(I.MemberId) |
    isChannelOwner(I.MemberId, I.ChannelId) |
    isChannelSubscriber(I.MemberId, I.ChannelId) |
    isChannelSubscribed(I.ChannelId, I.MemberId) |
    isAdminChannel(I.ChannelId) |
    not(Predicate) |
    and(Predicate, Predicate) |
    or(Predicate, Predicate) |
    andAll(Predicate-list) |
    equal(Number, Number)
value
  exec: Predicate >< I.MemberId >< M.Message >< S.State -~-> Bool,
  getPrecondition: I.MemberId >< M.Message -> Predicate

type
  Number ==
    constant(Nat) |
    channelsOwned(I.MemberId) |
    channelsSubscribed(I.MemberId) |
    channelSubscribers(I.ChannelId) |
    add(Number, Number)
value
  exec: Number >< I.MemberId >< M.Message >< S.State -~-> Nat

end
```

Scheme 30: MODEL 7 - GATEWAY

```
ID, MESSAGE, MEMBER, CHANNEL, STATE, OPERATIONS

scheme GATEWAY(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E),
O:OPERATIONS(I,M,C,E,S)) = class
```

```
value
  send: M.Message >< S.State -~-> S.State
  send(m, s) is
    let i = M.sender(m), e = E.enqueueInbox(m, S.getMember(i, s))
    in S.modifyMember(i, e, s) end
    pre canSend(m, s),

  canSend: M.Message >< S.State -> Bool
  canSend(m, s) is E.iswfMessage(m, S.getMember(M.sender(m), s))

value
  receive: I.MemberId >< S.State -~-> M.Message >< S.State
  receive(i, s) is
    let
      (m, e) = E.dequeueOutbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s)
    in
      let
        a = O.getAction(i, m), p = O.getPrecondition(i, m)
      in
        if O.exec(p, i, m, s)
        then (m, O.exec(a, i, m, s'))
        else (M.re_error(true, m), s') end
      end
    end pre canReceive(i, s),

  canReceive: I.MemberId >< S.State -> Bool
  canReceive(i, s) is ~E.isEmptyOutbox(S.getMember(i, s))

value
  transfer: I.MemberId >< S.State -~-> S.State
  transfer(i, s) is
    let
      (m, e) = E.dequeueInbox(S.getMember(i, s)),
      s' = S.modifyMember(i, e, s),
      io = S.getOwner(M.recepients(m), s)
    in
      if i ~= io then transferAll(m, {io}, s') else
        let ids = C.subscribers(S.getChannel(M.recepients(m), s))
        in transferAll(m, ids, s') end
      end
    end pre canTransfer(i, s),

  canTransfer: I.MemberId >< S.State -> Bool
  canTransfer(i, s) is
    ~E.isEmptyInbox(S.getMember(i, s))

value
  transferAll: M.Message >< I.MemberId-set >< S.State -~-> S.State
  transferAll(m, ids, s) is
    if ids = {} then s else
      let i: I.MemberId :- i isin ids in
        let
          e = E.enqueueOutbox(m, S.getMember(i, s)),
          s' = S.modifyMember(i, e, s)
        in transferAll(m, ids \ {i}, s') end
      end
    end
    pre canTransferAll(m, ids, s),

  canTransferAll: M.Message >< I.MemberId-set >< S.State -> Bool
  canTransferAll(m, ids, s) is
    S.membersExist(ids, s) /\
    ids <<= C.subscribers(S.getChannel(M.recepients(m), s))

end
```

Scheme 31: MODEL 7 - INTERFACE

```
ID, MESSAGE, MEMBER, CHANNEL, STATE, GATEWAY, OPERATIONS, ACTION, PREDICATE, NUMBER

scheme INTERFACE(I:ID, M:MESSAGE(I), C:CHANNEL(I), E:MEMBER(I,M,C), S:STATE(I,M,C,E)) =
class

object
   O: OPERATIONS(I,M,C,E,S),
   G: GATEWAY(I,M,C,E,S,O),
   A: ACTION(I,M,C,E,S,O,G),
   P: PREDICATE(I,M,C,E,S,O,G),
   N: NUMBER(I,M,C,E,S,O,G)

value
  register: I.MemberId >< S.State -~-> S.State
  register(i, s) is
    let
      mtype = M.register(i),
      sender = I.visitor, recepients = I.adminChannel(I.visitor)
    in G.send(M.mk_Message(mtype, sender, recepients, false, M.empty), s) end
    pre canRegister(i, s),

  canRegister: I.MemberId >< S.State -> Bool
  canRegister(i, s) is ~S.memberExists(i, s)

value
  unregister: I.MemberId >< S.State -~-> S.State
  unregister(i, s) is
    let
      mtype = M.unregister(i),
      sender = I.visitor, recepients = I.adminChannel(I.visitor)
    in G.send(M.mk_Message(mtype, sender, recepients, false, M.empty), s) end
    pre canUnregister(i, s),

  canUnregister: I.MemberId >< S.State -> Bool
  canUnregister(i, s) is
    S.memberExists(i, s) /\
    i ~= I.admin /\ i ~= I.visitor /\
    E.channelsOwned(S.getMember(i, s)) = {} /\
    E.channelsSubscribed(S.getMember(i, s)) = {I.adminChannel(i)}

value
  create: I.ChannelId >< I.MemberId >< S.State -~-> S.State
  create(ic, im, s) is
    let
      mtype = M.create(ic, im),
      sender = im, recepients = I.adminChannel(im)
    in G.send(M.mk_Message(mtype, sender, recepients, false, M.empty), s) end
    pre canCreate(ic, im, s),

  canCreate: I.ChannelId >< I.MemberId >< S.State -> Bool
  canCreate(ic, im, s) is
    ~S.channelExists(ic, s) /\ S.memberExists(im, s) /\ im ~= I.visitor

value
  destroy: I.ChannelId >< I.MemberId >< S.State -~-> S.State
  destroy(ic, im, s) is
    let
      mtype = M.destroy(ic, im),
      sender = im, recepients = I.adminChannel(im)
    in G.send(M.mk_Message(mtype, sender, recepients, false, M.empty), s) end
    pre canDestroy(ic, im, s),

  canDestroy: I.ChannelId >< I.MemberId >< S.State -> Bool
  canDestroy(ic, im, s) is
    S.channelExists(ic, s) /\ im = S.getOwner(ic, s) /\
```

```
    ~I.isAdminChannel(ic) /\ C.noSubscribers(S.getChannel(ic, s))

value
  subscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  subscribe(im, ic, s) is
    let
      mtype = M.subscribe(im, ic),
      sender = im, recepients = I.adminChannel(im)
    in G.send(M.mk_Message(mtype, sender, recepients, false, M.empty), s) end
    pre canSubscribe(im, ic, s),

  canSubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canSubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    ~C.isSubscriber(im, S.getChannel(ic, s)) /\
    ~C.isOwner(im, S.getChannel(ic, s)) /\ im ~= I.visitor

value
  unsubscribe: I.MemberId >< I.ChannelId >< S.State -~-> S.State
  unsubscribe(im, ic, s) is
    let
      mtype = M.unsubscribe(im, ic),
      sender = im, recepients = I.adminChannel(im)
    in G.send(M.mk_Message(mtype, sender, recepients, false, M.empty), s) end
    pre canUnsubscribe(im, ic, s),

  canUnsubscribe: I.MemberId >< I.ChannelId >< S.State -> Bool
  canUnsubscribe(im, ic, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    C.isSubscriber(im, S.getChannel(ic, s)) /\ ~I.isAdminChannel(ic)

value
  forward: I.MemberId >< I.MemberId >< M.MessageBody >< S.State -~-> S.State
  forward(i1, i2, b, s) is
    let
      mtype = M.forward(i2),
      sender = i1, recepients = I.adminChannel(i1)
    in G.send(M.mk_Message(mtype, sender, recepients, false, b), s) end
    pre canForward(i1, i2, b, s),

  canForward: I.MemberId >< I.MemberId >< M.MessageBody >< S.State -> Bool
  canForward(i1, i2, b, s) is S.memberExists(i1, s) /\ S.memberExists(i2, s)

value
  other: I.MemberId >< I.ChannelId >< M.MessageBody >< S.State -~-> S.State
  other(im, ic, b, s) is
    let
      mtype = M.other,
      sender = im, recepients = ic
    in G.send(M.mk_Message(mtype, sender, recepients, false, b), s) end
    pre canOther(im, ic, b, s),

  canOther: I.MemberId >< I.ChannelId >< M.MessageBody >< S.State -> Bool
  canOther(im, ic, b, s) is
    S.memberExists(im, s) /\ S.channelExists(ic, s) /\
    (C.isOwner(im, S.getChannel(ic, s)) \/ C.isSubscriber(im, S.getChannel(ic, s)))

end
```

A.8. MODEL 10 – XML for Representing State, Messages and Messaging

Scheme 32: MODEL 10 - XML_TEXT

```
scheme XML_TEXT = class

type
  XMLText = Text

value
  newXMLText: Text -> XMLText
  newXMLText(t) is t

value
  concatenate: XMLText >< XMLText -> XMLText
  concatenate(t1, t2) is t1 ^ t2,

  infix: Nat >< Nat >< XMLText -> XMLText
  infix(n1, n2, t) is
    if n1 > n2 then <..> else
      let
        n1' = if n1 = 0 then 1 else n1 end,
        n2' = if n2 > len t then len t else n2 end
       in <. t(i) | i in <. n1' .. n2' .> .> end
    end,

  prefix: Nat >< XMLText -> XMLText
  prefix(n, t) is infix(1, n, t),

  suffix: Nat >< XMLText -> XMLText
  suffix(n, t) is if n > len t then t else infix(len t - n + 1, len t, t) end

end
```

Scheme 33: MODEL 10 - XML_ELEMENT

```
scheme XML_ELEMENT = class

type
  Name = Text,
  Value = Text,
  XMLElement ::
    name : Name <-> re_name
    attributes : Name -m-> Value <-> re_attributes

value
  isAttribute: Name >< XMLElement -> Bool
  isAttribute(a, e) is a isin dom attributes(e),

  newXMLElement: Name -> XMLElement
  newXMLElement(n) is mk_XMLElement(n, []),

  renameElement: Name >< XMLElement -> XMLElement
  renameElement(a, e) is re_name(a, e),

  addAttribute: Name >< Value >< XMLElement -~-> XMLElement
  addAttribute(a, v, e) is
    re_attributes(attributes(e) union [ a +> v ], e) pre ~isAttribute(a, e),

  deleteAttribute: Name >< XMLElement -~-> XMLElement
  deleteAttribute(a, e) is
    re_attributes(attributes(e) \ {a}, e) pre isAttribute(a, e),

  renameAttribute: Name >< Name >< XMLElement -~-> XMLElement
  renameAttribute(n1, n2, e) is
```

```
    re_attributes(attributes(e) !! [ n2 +> attributes(e)(n1) ], e)
    pre isAttribute(n1, e),

  revalueAttribute: Name >< Value >< XMLElement -~-> XMLElement
  revalueAttribute(n, v, e) is
    re_attributes(attributes(e) !! [ n +> v ], e) pre isAttribute(n, e)

end
```

Scheme 34: MODEL 10 - XML_NODE

```
XML_TEXT, XML_ELEMENT

scheme XML_NODE = extend XML_TEXT with extend XML_ELEMENT with class

type
  XMLContent ==
    text(getText: XMLText) |
    elem(getElement: XMLElement),
  XMLNode'::
    content: XMLContent <-> re_content
    parent: XMLParentNode <-> re_parent
    children: XMLNode'-list <-> re_children,
  XMLParentNode == none | put(getXMLNode: XMLNode')

type
  XMLNode = {| n: XMLNode' :- iswf(n) |}

value
  iswf: XMLNode' -> Bool
  iswf(n) is
    (isRoot(n) => isXMLElementNode(n)) /\
    (~isRoot(n) =>
      let p = getXMLNode(parent(n)) in isXMLElementNode(p) /\ isXMLChild(n, p) end
    ) /\
    (isXMLTextNode(n) => ~isRoot(n) /\ noChildren(n)) /\ ~isXMLDescendant(n, n),

  isRoot: XMLNode' -> Bool
  isRoot(n) is parent(n) = none,

  noChildren: XMLNode' -> Bool
  noChildren(n) is children(n) = <..>,

  isXMLElementNode: XMLNode' -> Bool
  isXMLElementNode(n) is case content(n) of text(_) -> false, elem(_) -> true end,

  isXMLTextNode: XMLNode' -> Bool
  isXMLTextNode(n) is case content(n) of text(_) -> true, elem(_) -> false end,

  isXMLChild: XMLNode' >< XMLNode' -> Bool
  isXMLChild(n, p) is n isin elems(children(p)),

  isXMLDescendant: XMLNode' >< XMLNode' -> Bool
  isXMLDescendant(n, p) is n = p \/ isXMLChild(n, p) \/
    (exists np: XMLNode' :- isXMLDescendant(n, np) /\ isXMLDescendant(np, p))

value
  descendants: XMLNode -~-> Nat
  descendants(n) is descendants'(n, children(n)) pre isXMLElementNode(n),

  descendants': XMLNode >< XMLNode-list -~-> Nat
  descendants'(n, c) is
    if c = <..> then 1 else descendants(hd c) + descendants'(n, tl c) end
    pre isXMLElementNode(n),
```

```
  ancestors: XMLNode -> Nat
  ancestors(n) is if isRoot(n) then 1 else 1 + ancestors(getXMLNode(parent(n))) end

value
  newXMLNode: XMLContent -> XMLNode
  newXMLNode(c) is mk_XMLNode'(c, none, <..>)

value
  changeXMLContent: XMLContent >< XMLNode -~-> XMLNode
  changeXMLContent(c, n) is re_content(c, n) pre canChangeXMLContent(c, n),

  canChangeXMLContent: XMLContent >< XMLNode -> Bool
  canChangeXMLContent(c, n) is
    case c of elem(_) -> true, text(_) -> ~isRoot(n) /\ noChildren(n) end

value
  changeXMLParent: XMLNode >< XMLNode -~-> XMLNode
  changeXMLParent(p, n) is re_parent(put(p), n) pre canChangeXMLParent(p, n),

  canChangeXMLParent: XMLNode >< XMLNode -> Bool
  canChangeXMLParent(p, n) is
    isXMLElementNode(p) /\ isXMLChild(n, p) /\
    (all np: XMLNode :- isXMLDescendant(np, n) => ~isXMLDescendant(p, np))

value
  addXMLChild: Nat >< XMLNode >< XMLNode -~-> XMLNode
  addXMLChild(m, n, p) is
    let
      c1 = <. children(p)(i) | i in <. 1 .. m - 1 .> .>,
      c2 = <. changeXMLParent(p, n) .>,
      c3 = <. children(p)(i) | i in <. m .. len children(p) .> .>
    in re_children(c1 ^ c2 ^ c3, p) end
    pre canAddXMLChild(m, n, p),

  canAddXMLChild: Nat >< XMLNode >< XMLNode -> Bool
  canAddXMLChild(m, n, p) is
    hasChild(m, p) /\ isXMLElementNode(p) /\
    (all np: XMLNode :- isXMLDescendant(np, n) => ~isXMLDescendant(p, np)),

  hasChild: Nat >< XMLNode -> Bool
  hasChild(m, p) is m >= 1 /\ m <= numberOfChildren(p),

  numberOfChildren: XMLNode -> Nat
  numberOfChildren(p) is len children(p)

value
  appendXMLChild: XMLNode >< XMLNode -~-> XMLNode
  appendXMLChild(n, p) is
    re_children(children(p) ^ <.n.>, p) pre canAppendXMLChild(n, p),

  canAppendXMLChild: XMLNode >< XMLNode -> Bool
  canAppendXMLChild(n, p) is
    isXMLElementNode(p) /\
    (all np: XMLNode :- isXMLDescendant(np, n) => ~isXMLDescendant(p, np))

value
  deleteXMLChild: Nat >< XMLNode -~-> XMLNode
  deleteXMLChild(m, p) is
    let
      c1 = <. children(p)(i) | i in <. 1 .. m - 1 .> .>,
      c2 = <. children(p)(i) | i in <. m + 1 .. len children(p) .> .>
    in re_children(c1 ^ c2, p) end
    pre hasChild(m, p)
value
  changeXMLChild: Nat >< XMLNode >< XMLNode -> XMLNode
  changeXMLChild(m, n, p) is addXMLChild(m, n, deleteXMLChild(m, p))
    pre canChangeXMLChild(m, n, p),
```

```
  canChangeXMLChild: Nat >< XMLNode >< XMLNode -> Bool
  canChangeXMLChild(m, n, p) is
    hasChild(m, p) /\
    (all np: XMLNode :- isXMLDescendant(np, n) => ~isXMLDescendant(p, np))

end
```

Scheme 35: MODEL 10 - XML_TEXT_NODE

```
XML_NODE

scheme XML_TEXT_NODE(N:XML_NODE) = class

type
  XMLNode = N.XMLNode,
  XMLText = N.XMLText,
  XMLContent = N.XMLContent
value
  content: XMLNode -> XMLContent = N.content,
  getText: XMLContent -> XMLText = N.getText,
  isXMLTextNode: XMLNode -> Bool = N.isXMLTextNode,
  re_content: XMLContent >< XMLNode -> XMLNode = N.re_content,
  text: XMLText -> XMLContent = N.text,
  newXMLNode: XMLContent -> XMLNode = N.newXMLNode,
  newXMLText: Text -> Text = N.newXMLText,
  addXMLChild: Nat >< XMLNode >< XMLNode -~-> XMLNode = N.addXMLChild,
  hasChild: Nat >< XMLNode -> Bool = N.hasChild,
  numberOfChildren: XMLNode -> Nat = N.numberOfChildren,
  isXMLElementNode: XMLNode -> Bool = N.isXMLElementNode

value
  getText: XMLNode -~-> XMLText
  getText(n) is getText(content(n)) pre isXMLTextNode(n),

  putText: XMLText >< XMLNode -~-> XMLNode
  putText(t, n) is re_content(text(t), n) pre isXMLTextNode(n)

value
  newXMLTextNode: Text -> XMLNode
  newXMLTextNode(t) is newXMLNode(text(newXMLText(t))),

  addNewXMLTextChild: Nat >< Text >< XMLNode -~-> XMLNode
  addNewXMLTextChild(m, t, p) is
    addXMLChild(m, newXMLTextNode(t), p) pre canAddNewXMLChild(m, p),

  canAddNewXMLChild: Nat >< XMLNode -> Bool
  canAddNewXMLChild(m, p) is
    (hasChild(m, p) \/ m = numberOfChildren(p) + 1) /\ isXMLElementNode(p)

end
```

Scheme 36: MODEL 10 - XML_ELEMENT_NODE

```
XML_NODE

scheme XML_ELEMENT_NODE(N:XML_NODE) = class

type
  XMLNode = N.XMLNode,
  XMLElement = N.XMLElement,
  XMLContent = N.XMLContent,
  Name = N.Name
value
```

```
  content: XMLNode -> XMLContent = N.content,
  getElement: XMLContent -> XMLElement = N.getElement,
  isXMLElementNode: XMLNode -> Bool = N.isXMLElementNode,
  re_content: XMLContent >< XMLNode -> XMLNode = N.re_content,
  elem: XMLElement -> XMLContent = N.elem,
  renameElement: Name >< XMLElement -> XMLElement = N.renameElement,
  addAttribute: Name >< Text >< XMLElement -~-> XMLElement = N.addAttribute,
  deleteAttribute: Name >< XMLElement -~-> XMLElement = N.deleteAttribute,
  renameAttribute: Name >< Name >< XMLElement -~-> XMLElement = N.renameAttribute,
  revalueAttribute: Name >< Text >< XMLElement -~-> XMLElement = N.revalueAttribute,
  isAttribute: Name >< XMLElement -> Bool = N.isAttribute,
  newXMLNode: XMLContent -> XMLNode = N.newXMLNode,
  newXMLElement: Name -> XMLElement = N.newXMLElement,
  addXMLChild: Nat >< XMLNode >< XMLNode -~-> XMLNode = N.addXMLChild,
  hasChild: Nat >< XMLNode -> Bool = N.hasChild,
  numberOfChildren: XMLNode -> Nat = N.numberOfChildren

value
  isAttribute: Name >< XMLNode -~-> Bool
  isAttribute(a, n) is isAttribute(a, getElement(n)) pre isXMLElementNode(n)

value
  getElement: XMLNode -~-> XMLElement
  getElement(n) is getElement(content(n)) pre isXMLElementNode(n),

  putElement: XMLElement >< XMLNode -~-> XMLNode
  putElement(e, n) is re_content(elem(e), n) pre isXMLElementNode(n),

  newXMLElementNode: Name -> XMLNode
  newXMLElementNode(a) is newXMLNode(elem(newXMLElement(a))),

  addNewXMLElementChild: Nat >< Name >< XMLNode -~-> XMLNode
  addNewXMLElementChild(m, a, p) is
    addXMLChild(m, newXMLElementNode(a), p) pre canAddNewXMLChild(m, p),

  canAddNewXMLChild: Nat >< XMLNode -> Bool
  canAddNewXMLChild(m, p) is
    (hasChild(m, p) \/ m = numberOfChildren(p) + 1) /\ isXMLElementNode(p)

value
  renameElement: Name >< XMLNode -~-> XMLNode
  renameElement(a, n) is putElement(renameElement(a, getElement(content(n))), n)
    pre isXMLElementNode(n),

  addAttribute: Name >< Text >< XMLNode -~-> XMLNode
  addAttribute(a, t, n) is putElement(addAttribute(a, t, getElement(n)), n)
    pre isXMLElementNode(n) /\ ~isAttribute(a, n),

  deleteAttribute: Name >< XMLNode -~-> XMLNode
  deleteAttribute(a, n) is putElement(deleteAttribute(a, getElement(n)), n)
    pre isXMLElementNode(n) /\ isAttribute(a, n),

  renameAttribute: Name >< Name >< XMLNode -~-> XMLNode
  renameAttribute(a1, a2, n) is putElement(renameAttribute(a1, a2, getElement(n)), n)
    pre isXMLElementNode(n) /\ isAttribute(a1, n),

  revalueAttribute: Name >< Text >< XMLNode -~-> XMLNode
  revalueAttribute(a, t, n) is putElement(revalueAttribute(a, t, getElement(n)), n)
    pre isXMLElementNode(n) /\ isAttribute(a, n)

end
```

Scheme 37: MODEL 10 - XML

```
XML_NODE, XML_TEXT_NODE, XML_ELEMENT_NODE
```

```
scheme XML = class

object
  N: XML_NODE,
  T: XML_TEXT_NODE(N),
  E: XML_ELEMENT_NODE(N)

type
  XMLContent = N.XMLContent,
  XMLParentNode = N.XMLParentNode,
  XMLNode = N.XMLNode

value
  text: XMLText -> XMLContent = N.text,
  getText: XMLContent-> XMLText = N.getText,
  elem: XMLElement -> XMLContent = N.elem,
  getElement: XMLContent -> XMLElement = N.getElement,
  content: XMLNode -> XMLContent = N.content,
  re_content: XMLContent >< XMLNode -> XMLNode = N.re_content,
  parent: XMLNode -> XMLParentNode = N.parent,
  re_parent: XMLParentNode >< XMLNode -> XMLNode = N.re_parent,
  children: XMLNode -> XMLNode-list = N.children,
  re_children: XMLNode-list >< XMLNode -> XMLNode = N.re_children,
  none: XMLParentNode = N.none,
  put: XMLNode -> XMLParentNode = N.put,
  getXMLNode: XMLParentNode -> XMLNode = N.getXMLNode,
  isRoot: XMLNode -> Bool = N.isRoot,
  noChildren: XMLNode -> Bool = N.noChildren,
  hasChild: Nat >< XMLNode -> Bool = N.hasChild,
  numberOfChildren: XMLNode -> Nat = N.numberOfChildren,
  isXMLElementNode: XMLNode -> Bool = N.isXMLElementNode,
  isXMLTextNode: XMLNode -> Bool = N.isXMLTextNode,
  isXMLChild: XMLNode >< XMLNode -> Bool = N.isXMLChild,
  isXMLDescendant: XMLNode >< XMLNode -> Bool = N.isXMLDescendant,
  newXMLNode: XMLContent -> XMLNode = N.newXMLNode,
  changeXMLContent: XMLContent >< XMLNode -> XMLNode = N.changeXMLContent,
  changeXMLParent: XMLNode >< XMLNode -~-> XMLNode = N.changeXMLParent,
  canChangeXMLParent: XMLNode >< XMLNode -> Bool = N.canChangeXMLParent,
  addXMLChild: Nat >< XMLNode >< XMLNode -~-> XMLNode = N.addXMLChild,
  canAddXMLChild: Nat >< XMLNode >< XMLNode -> Bool = N.canAddXMLChild,
  appendXMLChild: XMLNode >< XMLNode -~-> XMLNode = N.appendXMLChild,
  canAppendXMLChild: XMLNode >< XMLNode -> Bool = N.canAppendXMLChild,
  deleteXMLChild: Nat >< XMLNode -~-> XMLNode = N.deleteXMLChild,
  changeXMLChild: Nat >< XMLNode >< XMLNode -> XMLNode = N.changeXMLChild,
  canChangeXMLChild: Nat >< XMLNode >< XMLNode -> Bool = N.canChangeXMLChild

type
  XMLText = T.XMLText

value
  infix: Nat >< Nat >< XMLText -> XMLText = N.infix,
  prefix: Nat >< XMLText -> XMLText = N.prefix,
  suffix: Nat >< XMLText -> XMLText = N.suffix,
  getText: XMLNode -~-> XMLText = T.getText,
  newXMLText: Text -> XMLText = N.newXMLText,
  putText: XMLText >< XMLNode -~-> XMLNode = T.putText,
  newXMLTextNode: Text -> XMLNode = T.newXMLTextNode,
  addNewXMLTextChild: Nat >< Text >< XMLNode -~-> XMLNode = T.addNewXMLTextChild

type
  Name = N.Name,
  XMLElement = N.XMLElement

value
  name: XMLElement -> Name = N.name,
  attributes: XMLElement -> (Name -m-> Text) = N.attributes,
```

```
  isAttribute: Name >< XMLElement -> Bool = N.isAttribute,
  newXMLElement: Name -> XMLElement = N.newXMLElement,
  renameElement: Name >< XMLElement -> XMLElement = N.renameElement,
  addAttribute: Name >< Text >< XMLElement -~-> XMLElement = N.addAttribute,
  deleteAttribute: Name >< XMLElement -~-> XMLElement = N.deleteAttribute,
  renameAttribute: Name >< Name >< XMLElement -~-> XMLElement = N.renameAttribute,
  revalueAttribute: Name >< Text >< XMLElement -~-> XMLElement = N.revalueAttribute,
  isAttribute: Name >< XMLNode -~-> Bool = E.isAttribute,
  getElement: XMLNode -~-> XMLElement = E.getElement,
  putElement: XMLElement >< XMLNode -~-> XMLNode = E.putElement,
  newXMLElementNode: Name -> XMLNode = E.newXMLElementNode,
  addNewXMLElementChild: Nat >< Name >< XMLNode -~-> XMLNode = E.addNewXMLElementChild,
  canAddNewXMLChild: Nat >< XMLNode -> Bool = E.canAddNewXMLChild,
  renameElement: Name >< XMLNode -~-> XMLNode = E.renameElement,
  addAttribute: Name >< Text >< XMLNode -~-> XMLNode = E.addAttribute,
  deleteAttribute: Name >< XMLNode -~-> XMLNode = E.deleteAttribute,
  renameAttribute: Name >< Name >< XMLNode -~-> XMLNode = E.renameAttribute,
  revalueAttribute: Name >< Text >< XMLNode -~-> XMLNode = E.revalueAttribute

type
  XML = {| n : XMLNode :- iswf(n) |}

value
  iswf: XMLNode -> Bool
  iswf(n) is isXMLElementNode(n) /\ parent(n) = none

value
  descendants: XMLNode -~-> Nat
  descendants(n) is descendants'(n, children(n)) pre isXMLElementNode(n),

  descendants': XMLNode >< XMLNode-list -~-> Nat
  descendants'(n, c) is
    if c = <..> then 1 else descendants(hd c) + descendants'(n, tl c) end
    pre isXMLElementNode(n),

  ancestors: XMLNode -> Nat
  ancestors(n) is if isRoot(n) then 1 else 1 + ancestors(getXMLNode(parent(n))) end

type
  XMLChild = Nat,
  XMLPath = XMLChild-list

value
  isXMLPath: XMLPath >< XMLNode -> Bool
  isXMLPath(p, x) is
    p = <..> \/ hasChild(hd p, x) /\ isXMLPath(tl p, children(x)(hd p)),

  isXMLRootPath: XMLPath -> Bool
  isXMLRootPath(p) is p = <..>,

  getXMLPath: XMLNode -> XMLPath
  getXMLPath(n) is
    if parent(n) = none then <..> else
      let p = getXMLNode(parent(n)) in
        let m: Nat :- n = children(p)(m) in getXMLPath(p) ^ <.m.> end
      end
    end,

  getXMLNode: XMLPath >< XMLNode -~-> XMLNode
  getXMLNode(p, x) is
    if p = <..> then x else getXMLNode(tl p, children(x)(hd p)) end
    pre isXMLPath(p, x)


value
  root: Name -> XMLNode
  root(a) is newXMLElementNode(a)
```

```
value
  appendNewXMLTextChild: XMLPath >< XMLText >< XMLNode -~-> XMLNode
  appendNewXMLTextChild(p, t, x) is
    if len p = 0
    then appendXMLChild(newXMLTextNode(t), x)
    else changeXMLChild(hd p, appendNewXMLTextChild(tl p, t, children(x)(hd p)), x) end
    pre canAppendNewXMLTextChild(p, x),

  canAppendNewXMLTextChild: XMLPath >< XMLNode -> Bool
  canAppendNewXMLTextChild(p, x) is isXMLPath(p, x) /\ isXMLElementNode(getXMLNode(p, x))

value
  appendNewXMLElementChild: XMLPath >< Name >< XMLNode -~-> XMLNode
  appendNewXMLElementChild(p, a, x) is
    if len p = 0 then appendXMLChild(newXMLElementNode(a), x) else
      changeXMLChild(hd p, appendNewXMLElementChild(tl p, a, children(x)(hd p)), x) end
    pre canAppendNewXMLElementChild(p, x),

  canAppendNewXMLElementChild: XMLPath >< XMLNode -> Bool
  canAppendNewXMLElementChild(p, x) is
    isXMLPath(p, x) /\ isXMLElementNode(getXMLNode(p, x))

value
  addNewXMLTextChild: XMLPath >< Nat >< XMLText >< XMLNode -~-> XMLNode
  addNewXMLTextChild(p, m, t, x) is
    if len p = 0 then addNewXMLTextChild(m, t, x)
    else changeXMLChild(hd p, addNewXMLTextChild(tl p, m, t, children(x)(hd p)), x) end
    pre canAddNewXMLTextChild(p, m, x),

  canAddNewXMLTextChild: XMLPath >< Nat >< XMLNode -> Bool
  canAddNewXMLTextChild(p, m, x) is
    isXMLPath(p ^ <.m.>, x) /\ isXMLElementNode(getXMLNode(p, x))

value
  addNewXMLElementChild: XMLPath >< Nat >< Name >< XMLNode -~-> XMLNode
  addNewXMLElementChild(p, m, a, x) is
    if len p = 0 then addNewXMLElementChild(m, a, x) else
      changeXMLChild(hd p, addNewXMLElementChild(tl p, m, a, children(x)(hd p)), x) end
    pre canAddNewXMLElementChild(p, m, x),

  canAddNewXMLElementChild: XMLPath >< Nat >< XMLNode -> Bool
  canAddNewXMLElementChild(p, m, x) is
    isXMLPath(p ^ <.m.>, x) /\ isXMLElementNode(getXMLNode(p, x))

value
  appendXMLChild: XMLPath >< XMLNode >< XMLNode -~-> XMLNode
  appendXMLChild(p, n, x) is
    if len p = 0 then appendXMLChild(n, x)
    else changeXMLChild(hd p, appendXMLChild(tl p, n, children(x)(hd p)), x) end
    pre canAppendXMLChild(p, n, x),

  canAppendXMLChild: XMLPath >< XMLNode >< XMLNode -> Bool
  canAppendXMLChild(p, n, x) is
    isXMLPath(p, x) /\ isXMLElementNode(getXMLNode(p, x)) /\
    (all np: XMLNode :- isXMLDescendant(np, n) => ~isXMLDescendant(getXMLNode(p, x), np))

value
  changeXMLNode: XMLPath >< XMLNode >< XMLNode -~-> XMLNode
  changeXMLNode(p, n, x) is
    if p = <..> then n else
      changeXMLChild(hd p, changeXMLNode(tl p, n, children(x)(hd p)), x)
    end pre canChangeXMLNode(p, n, x),

  canChangeXMLNode: XMLPath >< XMLNode >< XMLNode -> Bool
  canChangeXMLNode(p, n, x) is
    isXMLPath(p, x) /\
```

```
    (all np: XMLNode :- isXMLDescendant(np, n) => ~isXMLDescendant(x, np))

value
  deleteXMLNode: XMLPath >< XMLNode -~-> XMLNode
  deleteXMLNode(p, x) is
    if len p = 1 then deleteXMLChild(hd p, x)
    else changeXMLChild(hd p, deleteXMLNode(tl p, children(x)(hd p)), x) end
    pre canDeleteXMLNode(p, x),

  canDeleteXMLNode: XMLPath >< XMLNode -> Bool
  canDeleteXMLNode(p, x) is isXMLPath(p, x) /\ ~isXMLRootPath(p)

end
```

Scheme 38: MODEL 10 - XMLALL

```
../XML/XML,
XBOOL, XNAT, XPATH, XTEXT, XNODE, XNODES, XTREE

scheme XMLALL(X:XML) = class

object
  XB: XBOOL(X, XR),
  XN: XNAT(X, XR),
  XP: XPATH(X, XR),
  XT: XTEXT(X, XR),
  XO: XNODE(X, XR),
  XS: XNODES(X, XR),
  XE: XTREE(X, XR),

  XR: class
    type
      XMLNode = X.XMLNode,
      XBool = XB.XBool,
      XNat = XN.XNat,
      XPath = XP.XPath,
      XText = XT.XText,
      XNode = XO.XNode,
      XNodes = XS.XNodes,
      XTree = XE.XTree

    value
      iswf: XBool >< XMLNode -> Bool = XB.iswf,
      iswf: XNat >< XMLNode -> Bool = XN.iswf,
      iswf: XPath >< XMLNode -> Bool = XP.iswf,
      iswf: XText >< XMLNode -> Bool = XT.iswf,
      iswf: XNode >< XMLNode -> Bool = XO.iswf,
      iswf: XNodes >< XMLNode -> Bool = XS.iswf,
      iswf: XTree >< XMLNode -> Bool = XE.iswf

    value
      exec: XBool >< XMLNode -~-> Bool = XB.exec,
      exec: XNat >< XMLNode -~-> Nat = XN.exec,
      exec: XPath >< XMLNode -~-> XMLNode = XP.exec,
      exec: XText >< XMLNode -~-> Text = XT.exec,
      exec: XNode >< XMLNode -~-> XMLNode = XO.exec,
      exec: XNodes >< XMLNode -~-> XMLNode-list = XS.exec,
      exec: XTree >< XMLNode -~-> XMLNode = XE.exec
end

end
```

Scheme 39: MODEL 10 - XMLREST

```
../XML/XML
```

```
scheme XMLREST(X:XML) = class

type
  XMLNode = X.XMLNode

type
  XBool, XNat, XPath, XText, XNode, XNodes, XTree

value
  iswf: XBool >< XMLNode -> Bool,
  iswf: XNat >< XMLNode -> Bool,
  iswf: XPath >< XMLNode -> Bool,
  iswf: XText >< XMLNode -> Bool,
  iswf: XNode >< XMLNode -> Bool,
  iswf: XNodes >< XMLNode -> Bool,
  iswf: XTree >< XMLNode -> Bool,

  exec: XBool >< XMLNode -~-> Bool,
  exec: XNat >< XMLNode -~-> Nat,
  exec: XPath >< XMLNode -~-> XMLNode,
  exec: XText >< XMLNode -~-> Text,
  exec: XNode >< XMLNode -~-> XMLNode,
  exec: XNodes >< XMLNode -~-> XMLNode-list,
  exec: XTree >< XMLNode -~-> XMLNode

end
```

Scheme 40: MODEL 10 - XBOOL

```
../XML/XML, XMLREST

scheme XBOOL(X:XML, R:XMLREST(X)) = class

type
  XNat = R.XNat,
  XText = R.XText,
  XNode = R.XNode,
  XNodes = R.XNodes,
  XMLNode = X.XMLNode,
  XMLParentNode = X.XMLParentNode,
  Name = X.Name,
  XMLElement = X.XMLElement

value
  exec: XNat >< XMLNode -~-> Nat = R.exec,
  iswf: XText >< XMLNode -> Bool = R.iswf,
  exec: XText >< XMLNode -~-> Text = R.exec,
  iswf: XNat >< XMLNode -> Bool = R.iswf,
  exec: XNode >< XMLNode -~-> XMLNode = R.exec,
  iswf: XNode >< XMLNode -> Bool = R.iswf,
  iswf: XNodes >< XMLNode -> Bool = R.iswf,
  exec: XNodes >< XMLNode -~-> XMLNode-list = R.exec,
  parent: XMLNode -> XMLParentNode = X.parent,
  none: XMLParentNode = X.none,
  children: XMLNode -> XMLNode-list = X.children,
  isXMLTextNode: XMLNode -> Bool = X.isXMLTextNode,
  isXMLElementNode: XMLNode -> Bool = X.isXMLElementNode,
  getElement: XMLNode -~-> XMLElement = X.getElement,
  attributes: XMLElement -> (Name -m-> Text) = X.attributes,
  name: XMLElement -> Name = X.name

type
  XBool ==
    tt |
```

```
    not(XBool) |
    and(XBool, XBool) |
    isTrue(XBool, XNode) |
    areTrue(XBool, XNodes) |
    isText(XNode) |
    isElem(XNode) |
    isRoot(XNode) |
    hasChild(XNat, XNode) |
    hasAttribute(Name, XNode) |
    hasName(Name, XNode) |
    isLess(XNat, XNat) |
    isEqualBool(XBool, XBool) |
    isEqualNat(XNat, XNat) |
    isEqualText(XText, XText)

value
  iswf: XBool >< XMLNode -> Bool
  iswf(b, x) is
    case b of
      tt -> true,
      not(b') -> iswf(b', x),
      and(b1, b2) -> iswf(b1, x) /\ iswf(b2, x),
      isTrue(b', o) -> iswf(o, x) /\ iswf(b', exec(o, x)),
      areTrue(b', ns) -> iswf(ns, x) /\
        (all n: XMLNode :- n isin elems exec(ns, x) => iswf(b', n)),
      isText(o) -> iswf(o, x),
      isElem(o) -> iswf(o, x),
      isRoot(o) -> iswf(o, x),
      hasChild(n, o) -> iswf(o, x) /\ iswf(n, exec(o, x)),
      hasAttribute(a, o) -> iswf(o, x) /\ isXMLElementNode(exec(o, x)),
      hasName(a, o) -> iswf(o, x) /\ isXMLElementNode(exec(o, x)),
      isLess(n1, n2) -> iswf(n1, x) /\ iswf(n2, x),
      isEqualBool(b1, b2) -> iswf(b1, x) /\ iswf(b2, x),
      isEqualNat(n1, n2) -> iswf(n1, x) /\ iswf(n2, x),
      isEqualText(t1, t2) -> iswf(t1, x) /\ iswf(t2, x)
    end,

  exec: XBool >< XMLNode -~-> Bool
  exec(b, x) is
    case b of
      tt -> true,
      not(b') -> ~exec(b', x),
      and(b1, b2) -> exec(b1, x) /\ exec(b2, x),
      isTrue(b', o) -> exec(b', exec(o, x)),
      areTrue(b', ns) ->
        (all n: XMLNode :-
          n isin elems exec(ns, x) => exec(b', n)
        ),
      isText(o) -> isXMLTextNode(exec(o, x)),
      isRoot(o) -> parent(exec(o, x)) = none,
      hasChild(n, o) ->
        let x' = exec(o, x), n' = exec(n, x')
        in n' >= 1 /\ n' <= len children(x') end,
      hasAttribute(a, o) ->
        a isin dom attributes(getElement(exec(o, x))),
      hasName(a, o) -> name(getElement(exec(o, x))) = a,
      isLess(n1, n2) -> exec(n1, x) < exec(n2, x),
      isEqualBool(b1, b2) -> exec(b1, x) = exec(b2, x),
      isEqualNat(n1, n2) -> exec(n1, x) = exec(n2, x),
      isEqualText(t1, t2) -> exec(t1, x) = exec(t2, x)
    end

end
```

Scheme 41: MODEL 10 - XNAT

```
../XML/XML, XMLREST
```

```
scheme XNAT(X:XML, R:XMLREST(X)) = class

type
  XBool = R.XBool,
  XNode = R.XNode,
  XNodes = R.XNodes,
  XMLNode = X.XMLNode,
  XMLText = X.XMLText,
  XMLElement = X.XMLElement,
  Name = X.Name
value
  iswf: XBool >< XMLNode -> Bool = R.iswf,
  exec: XBool >< XMLNode -~-> Bool = R.exec,
  iswf: XNode >< XMLNode -> Bool = R.iswf,
  exec: XNode >< XMLNode -~-> XMLNode = R.exec,
  iswf: XNodes >< XMLNode -> Bool = R.iswf,
  exec: XNodes >< XMLNode -~-> XMLNode-list = R.exec,
  children: XMLNode -> XMLNode-list = X.children,
  isXMLElementNode: XMLNode -> Bool = X.isXMLElementNode,
  isXMLTextNode: XMLNode -> Bool = X.isXMLTextNode,
  attributes: XMLElement -> (Name -m-> Text) = X.attributes,
  getElement: XMLNode -~-> XMLElement = X.getElement,
  getText: XMLNode -~-> XMLText = X.getText,
  descendants: XMLNode -~-> Nat = X.descendants,
  ancestors: XMLNode -~-> Nat = X.ancestors

type
  XNat ==
    num(Nat) |
    add(XNat, XNat) |
    subtract(XNat, XNat) |
    children(XNode) |
    attributes(XNode) |
    textLength(XNode) |
    descendants(XNode) |
    ancestors(XNode) |
    length(XNodes) |
    condition(XBool, XNat, XNat)

value
  iswf: XNat >< XMLNode -> Bool
  iswf(n, x) is
    case n of
      num(_) -> true,
      add(n1, n2) -> iswf(n1, x) /\ iswf(n2, x),
      subtract(n1, n2) -> iswf(n1, x) /\ iswf(n2, x) /\ exec(n1, x) >= exec(n2, x),
      children(o) -> iswf(o, x),
      attributes(o) -> iswf(o, x) /\ isXMLElementNode(exec(o, x)),
      textLength(o) -> iswf(o, x) /\ isXMLTextNode(exec(o, x)),
      descendants(o) -> iswf(o, x) /\ isXMLElementNode(exec(o, x)),
      ancestors(o) -> iswf(o, x),
      length(ns) -> iswf(ns, x),
      condition(b, n1, n2) -> iswf(b, x) /\ iswf(n1, x) /\ iswf(n2, x)
    end

value
  exec: XNat >< XMLNode -~-> Nat
  exec(n, x) is
    case n of
      num(m) -> m,
      add(n1, n2) -> exec(n1, x) + exec(n2, x),
      subtract(n1, n2) -> exec(n1, x) - exec(n2, x),
      children(o) -> len children(exec(o, x)),
      attributes(o) -> card dom attributes(getElement(exec(o, x))),
      textLength(o) -> len getText(exec(o, x)),
      descendants(o) -> descendants(exec(o, x)),
```

```
      ancestors(o) -> ancestors(exec(o, x)),
      length(ns) -> len exec(ns, x),
      condition(b, n1, n2) -> if exec(b, x) then exec(n1, x) else exec(n2, x) end
    end
    pre iswf(n, x)

end
```

Scheme 42: MODEL 10 - XTEXT

```
../XML/XML, XMLREST

scheme XTEXT(X:XML, R:XMLREST(X)) = class

type
  XNat = R.XNat,
  XMLNode = X.XMLNode,
  XNode = R.XNode,
  XMLElement = X.XMLElement,
  Name = X.Name
value
  iswf: XNat >< XMLNode -> Bool = R.iswf,
  exec: XNat >< XMLNode -> Nat = R.exec,
  iswf: XNode >< XMLNode -> Bool = R.iswf,
  exec: XNode >< XMLNode -> XMLNode = R.exec,
  infix: Nat >< Nat >< Text -> Text = X.infix,
  prefix: Nat >< Text -> Text = X.prefix,
  suffix: Nat >< Text -> Text = X.suffix,
  isXMLTextNode: XMLNode -> Bool = X.isXMLTextNode,
  getText: XMLNode -~-> Text = X.getText,
  isXMLElementNode: XMLNode -> Bool = X.isXMLElementNode,
  getElement: XMLNode -~-> XMLElement = X.getElement,
  name: XMLElement -> Name = X.name,
  attributes: XMLElement -> (Name -m-> Text) = X.attributes

type
  XText ==
    text(Text) |
    concatenate(XText, XText) |
    infix(XNat, XNat, XText) |
    prefix(XNat, XText) |
    suffix(XNat, XText) |
    getText(XNode) |
    getElementName(XNode) |
    getAttributeValue(XNode, XText)

value
  iswf: XText >< XMLNode -> Bool
  iswf(t, x) is
    case t of
      text(_) -> true,
      concatenate(t1, t2) -> iswf(t1, x) /\ iswf(t2, x),
      infix(n1, n2, t') -> iswf(n1, x) /\ iswf(n2, x) /\ iswf(t', x),
      prefix(n, t') -> iswf(n, x) /\ iswf(t', x),
      suffix(n, t') -> iswf(n, x) /\ iswf(t', x),
      getText(o) -> iswf(o, x) /\ isXMLTextNode(exec(o, x)),
      getElementName(o) -> iswf(o, x) /\ isXMLElementNode(exec(o, x)),
      getAttributeValue(o, t) ->
        iswf(o, x) /\ iswf(t, x) /\
        let x' = exec(o, x), t' = exec(t, x')
        in isXMLElementNode(x') /\ t' isin dom attributes(getElement(x')) end
    end

value
  exec: XText >< XMLNode -~-> Text
```

```
  exec(t, x) is
    case t of
      text(t') -> t',
      concatenate(t1, t2) -> exec(t1, x) ^ exec(t2, x),
      infix(n1, n2, t') -> infix(exec(n1, x), exec(n2, x), exec(t', x)),
      prefix(n, t') -> prefix(exec(n, x), exec(t', x)),
      suffix(n, t') -> suffix(exec(n, x), exec(t', x)),
      getText(o) -> getText(exec(o, x)),
      getElementName(o) -> name(getElement(exec(o, x))),
      getAttributeValue(o, t) ->
        let x' = exec(o, x), t' = exec(t, x')
        in attributes(getElement(x'))(t') end
    end
    pre iswf(t, x)

end
```

Scheme 43: MODEL 10 - XNODE

```
../XML/XML, XMLREST

scheme XNODE(X:XML, R:XMLREST(X)) = class

type
  XMLNode = X.XMLNode,
  XText = R.XText,
  XBool = R.XBool,
  XNat = R.XNat,
  XNodes = R.XNodes,
  Name = X.Name,
  XPath = R.XPath
value
  isXMLTextNode: XMLNode -> Bool = X.isXMLTextNode,
  isXMLElementNode: XMLNode -> Bool = X.isXMLElementNode,
  iswf: XText >< XMLNode -> Bool = R.iswf,
  exec: XText >< XMLNode -~-> Text = R.exec,
  iswf: XBool >< XMLNode -> Bool = R.iswf,
  exec: XBool >< XMLNode -~-> Bool = R.exec,
  iswf: XNat >< XMLNode -> Bool = R.iswf,
  exec: XNat >< XMLNode -~-> Nat = R.exec,
  iswf: XPath >< XMLNode -> Bool = R.iswf,
  exec: XPath >< XMLNode -~-> XMLNode = R.exec,
  iswf: XNodes >< XMLNode -> Bool = R.iswf,
  exec: XNodes >< XMLNode -~-> XMLNode-list = R.exec,
  putText: Text >< XMLNode -~-> XMLNode = X.putText,
  renameElement: Name >< XMLNode -~-> XMLNode = X.renameElement,
  isAttribute: Name >< XMLNode -~-> Bool = X.isAttribute,
  addAttribute: Name >< Text >< XMLNode -~-> XMLNode = X.addAttribute,
  deleteAttribute: Name >< XMLNode -~-> XMLNode = X.deleteAttribute,
  renameAttribute: Name >< Name >< XMLNode -~-> XMLNode = X.renameAttribute,
  revalueAttribute: Name >< Text >< XMLNode -~-> XMLNode = X.revalueAttribute

type
  XNode ==
    one(XPath) |
    changeText(XPath, XText) |
    renameElement(XPath, XText) |
    addAttribute(XPath, XText, XText) |
    deleteAttribute(XPath, XText) |
    renameAttribute(XPath, XText, XText) |
    revalueAttribute(XPath, XText, XText) |
    getOne(XNat, XNodes) |
    sequential(XNode, XNode) |
    conditional(XBool, XNode, XNode)
```

```
value
  iswf: XNode >< XMLNode -> Bool
  iswf(n, x) is
    case n of
      one(p) -> iswf(p, x),
      changeText(p, t) ->
        iswf(p, x) /\ let x' = exec(p, x)  in iswf(t, x') /\ isXMLTextNode(x') end,
      renameElement(p, t) ->
        iswf(p, x) /\ let x' = exec(p, x) in iswf(t, x') /\ isXMLElementNode(x') end,
      addAttribute(p, t1, t2) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t1, x') /\ iswf(t2, x') /\
          isXMLElementNode(x') /\ ~isAttribute(exec(t1, x'), x')
        end,
      deleteAttribute(p, t) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t, x') /\ isXMLElementNode(x') /\ isAttribute(exec(t, x'), x')
        end,
      renameAttribute(p, t1, t2) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t1, x') /\ iswf(t2, x') /\
          isXMLElementNode(x') /\ isAttribute(exec(t1, x'), x')
        end,
      revalueAttribute(p, t1, t2) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t1, x') /\ iswf(t2, x') /\
          isXMLElementNode(x') /\ isAttribute(exec(t1, x'), x')
        end,
      getOne(m, ns) -> iswf(m, x) /\ iswf(ns, x) /\ exec(m, x) isin inds exec(ns, x),
      sequential(n1, n2) -> iswf(n1, x) /\ iswf(n2, exec(n1, x)),
      conditional(b, n1, n2) -> iswf(b, x) /\ iswf(n1, x) /\ iswf(n2, x)
    end

value
  exec: XNode >< XMLNode -~-> XMLNode
  exec(n, x) is
    case n of
      one(p) -> exec(p, x),
      changeText(p, t) -> let x' = exec(p, x) in putText(exec(t, x'), x') end,
      renameElement(p, t) -> let x' = exec(p, x) in renameElement(exec(t, x'), x') end,
      addAttribute(p, t1, t2) ->
        let x' = exec(p, x) in addAttribute(exec(t1, x'), exec(t2, x'), x') end,
      deleteAttribute(p, t) ->
        let x' = exec(p, x) in deleteAttribute(exec(t, x'), x') end,
      renameAttribute(p, t1, t2) ->
        let x' = exec(p, x) in renameAttribute(exec(t1, x'), exec(t2, x'), x') end,
      revalueAttribute(p, t1, t2) ->
        let x' = exec(p, x) in revalueAttribute(exec(t1, x'), exec(t2, x'), x') end,
      getOne(m, ns) -> exec(ns, x)(exec(m, x)),
      sequential(n1, n2) -> exec(n2, exec(n1, x)),
      conditional(b, n1, n2) -> if exec(b, x) then exec(n1, x) else exec(n2, x) end
    end
    pre iswf(n, x)

end
```

Scheme 44: MODEL 10 - XNODES

```
../XML/XML, XMLREST
```

```
scheme XNODES(X:XML, R:XMLREST(X)) = class

type
  XMLNode = X.XMLNode,
  XPath = R.XPath,
  XBool = R.XBool,
  XNode = R.XNode
value
  iswf: XPath >< XMLNode -> Bool = R.iswf,
  exec: XPath >< XMLNode -~-> XMLNode = R.exec,
  iswf: XBool >< XMLNode -> Bool = R.iswf,
  exec: XBool >< XMLNode -~-> Bool = R.exec,
  iswf: XNode >< XMLNode -> Bool = R.iswf,
  exec: XNode >< XMLNode -~-> XMLNode = R.exec,
  children: XMLNode -> XMLNode-list = X.children,
  isXMLElementNode: XMLNode -> Bool = X.isXMLElementNode

type
  XNodes ==
    empty |
    one(XNode) |
    children(XPath) |
    filter(XNodes, XBool) |
    concatenate(XNodes, XNodes)

value
  iswf: XNodes >< XMLNode -> Bool
  iswf(ns, x) is
    case ns of
      empty -> true,
      one(n) -> iswf(n, x),
      children(p) -> iswf(p, x) /\ isXMLElementNode(exec(p, x)),
      filter(ns, b) -> iswf(ns, x),
      concatenate(ns1, ns2) -> iswf(ns1, x) /\ iswf(ns2, x)
     end

value
  exec: XNodes >< XMLNode -~-> XMLNode-list
  exec(ns, x) is
    case ns of
      empty -> <..>,
      one(n) -> <.exec(n, x).>,
      children(p) -> children(exec(p, x)),
      filter(ns, b) ->
        let ns' = exec(ns, x) in
          <. ns'(i) | i in <. 1 .. len ns' .> :- iswf(b, ns'(i)) /\ exec(b, ns'(i)) .>
        end,
      concatenate(ns1, ns2) -> exec(ns1, x) ^ exec(ns2, x)
    end
    pre iswf(ns, x)

end
```

Scheme 45: MODEL 10 - XPATH

```
../XML/XML, XMLREST

scheme XPATH(X:XML, R:XMLREST(X)) = class

type
  XMLNode = X.XMLNode,
  XMLParentNode = X.XMLParentNode,
  XBool = R.XBool,
  XNat = R.XNat
```

```
value
  parent: XMLNode -> XMLParentNode = X.parent,
  none: XMLParentNode = X.none,
  getXMLNode: XMLParentNode -~-> XMLNode = X.getXMLNode,
  children: XMLNode -> XMLNode-list = X.children,
  iswf: XBool >< XMLNode -> Bool = R.iswf,
  exec: XBool >< XMLNode -~-> Bool = R.exec,
  iswf: XNat >< XMLNode -> Bool = R.iswf,
  exec: XNat >< XMLNode -~-> Nat = R.exec,
  hasChild: Nat >< XMLNode -~-> Bool = X.hasChild

type
  XPath ==
    self |
    parent(XPath) |
    child(XNat, XPath) |
    compose(XPath, XPath) |
    conditional(XBool, XPath, XPath)

value
  iswf: XPath >< XMLNode -> Bool
  iswf(p, x) is
    case p of
      self -> true,
      parent(p') -> parent(x) ~= none /\ iswf(p', getXMLNode(parent(x))),
      child(n, p') ->
        iswf(p', x) /\ iswf(n, exec(p', x)) /\
        let x' = exec(p', x), n' = exec(n, x') in hasChild(n', x') end,
      compose(p1, p2) -> iswf(p1, x) /\ iswf(p2, exec(p1, x)),
      conditional(b, p1, p2) -> iswf(b, x) /\ iswf(p1, x) /\ iswf(p2, x)
    end

value
  exec: XPath >< XMLNode -~-> XMLNode
  exec(p, x) is
    case p of
      self -> x,
      parent(p') -> exec(p', getXMLNode(parent(x))),
      child(n, p') -> let x' = exec(p', x), n' = exec(n, x') in children(x')(n') end,
      compose(p1, p2) -> exec(p2, exec(p1, x)),
      conditional(b, p1, p2) -> if exec(b, x) then exec(p1, x) else exec(p2, x) end
    end
    pre iswf(p, x)

end
```

Scheme 46: MODEL 10 - XDATA

```
../XML/XML,
XMLFAMILY

scheme XDATA(X:XML, F:XMLFAMILY(X)) = class

type
  XMLNode = X.XMLNode,
  XNode = F.XNode,
  XPath = F.XPath,
  XMLPath = X.XMLPath,
  XText = F.XText,
  XNat = F.XNat,
  XMLText = X.XMLText,
  Name = X.Name
value
  iswf: XPath >< XMLNode -> Bool = F.iswf,
  exec: XPath >< XMLNode -~-> XMLNode = F.exec,
```

```
  iswf: XNode >< XMLNode -> Bool = F.iswf,
  exec: XNode >< XMLNode -~-> XMLNode = F.exec,
  iswf: XText >< XMLNode -> Bool = F.iswf,
  exec: XText >< XMLNode -~-> Text = F.exec,
  iswf: XNat >< XMLNode -> Bool = F.iswf,
  exec: XNat >< XMLNode -~-> Nat = F.exec,
  isRoot: XMLNode -> Bool = X.isRoot,
  getXMLPath: XMLNode -> XMLPath = X.getXMLPath,
  appendNewXMLTextChild: XMLPath >< XMLText >< XMLNode -~-> XMLNode
    = X.appendNewXMLTextChild,
  appendNewXMLElementChild: XMLPath >< Name >< XMLNode -~-> XMLNode
    = X.appendNewXMLElementChild,
  addNewXMLTextChild: XMLPath >< Nat >< XMLText >< XMLNode -~-> XMLNode
    = X.addNewXMLTextChild,
  addNewXMLElementChild: XMLPath >< Nat >< Name >< XMLNode -~-> XMLNode
    = X.addNewXMLElementChild,
  deleteXMLNode: XMLPath >< XMLNode -~-> XMLNode = X.deleteXMLNode,
  changeXMLNode: XMLPath >< XMLNode >< XMLNode -~-> XMLNode = X.changeXMLNode,
  isXMLElementNode: XMLNode -> Bool = X.isXMLElementNode,
  hasChild: Nat >< XMLNode -> Bool = X.hasChild,
  root: Text -> XMLNode = X.root

type
  XData ==
    none |
    root(XText) |
    appendNewTextChild(XPath, XText) |
    appendNewElementChild(XPath, XText) |
    addNewTextChild(XPath, XNat, XText) |
    addNewElementChild(XPath, XNat, XText) |
    deleteNode(XPath) |
    changeNode(XPath, XNode)

value
  iswf: XData >< XMLNode -> Bool
  iswf(d, x) is
    case d of
      none -> true,
      root(t) -> iswf(t, x),
      appendNewTextChild(p, t) ->
        iswf(p, x) /\ iswf(t, exec(p, x)) /\ isXMLElementNode(exec(p, x)),
      appendNewElementChild(p, t) ->
        iswf(p, x) /\ iswf(t, exec(p, x)) /\ isXMLElementNode(exec(p, x)),
      addNewTextChild(p, n, t) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t, x') /\ iswf(n, x') /\ isXMLElementNode(x') /\ hasChild(exec(n, x'), x')
        end,
      addNewElementChild(p, n, t) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t, x') /\ iswf(n, x') /\ isXMLElementNode(x') /\ hasChild(exec(n, x'), x')
        end,
      deleteNode(p) -> iswf(p, x) /\ ~isRoot(exec(p, x)),
      changeNode(p, n) -> iswf(p, x) /\ iswf(n, exec(p, x))
    end

value
  exec: XData >< XMLNode -~-> XMLNode
  exec(d, x) is
    case d of
      none -> x,
      root(t) -> root(exec(t, x)),
      appendNewTextChild(p, t) ->
        let x' = exec(p, x)
        in appendNewXMLTextChild(getXMLPath(x'), exec(t, x'), x) end,
      appendNewElementChild(p, t) ->
```

```
        let x' = exec(p, x)
        in appendNewXMLElementChild(getXMLPath(x'), exec(t, x'), x) end,
      addNewTextChild(p, n, t) ->
        let x' = exec(p, x)
        in addNewXMLTextChild(getXMLPath(x'), exec(n, x'), exec(t, x'), x) end,
      addNewElementChild(p, n, t) ->
        let x' = exec(p, x)
        in addNewXMLElementChild(getXMLPath(x'), exec(n, x'), exec(t, x'), x) end,
      deleteNode(p) -> deleteXMLNode(getXMLPath(exec(p, x)), x),
      changeNode(p, n) ->
        let x' = exec(p, x), x'' = exec(n, x')
        in changeXMLNode(getXMLPath(x'), x'', x) end
    end
    pre iswf(d, x)

end
```

Scheme 47: MODEL 10 - XTREE

```
../XML/XML, XMLREST

scheme XTREE(X:XML, R:XMLREST(X)) = class

type
  XMLNode = X.XMLNode,
  XNode = R.XNode,
  XPath = R.XPath,
  XMLPath = X.XMLPath,
  XText = R.XText,
  XNat = R.XNat,
  XMLText = X.XMLText,
  Name = X.Name
value
  iswf: XPath >< XMLNode -> Bool = R.iswf,
  exec: XPath >< XMLNode -~-> XMLNode = R.exec,
  iswf: XNode >< XMLNode -> Bool = R.iswf,
  exec: XNode >< XMLNode -~-> XMLNode = R.exec,
  iswf: XText >< XMLNode -> Bool = R.iswf,
  exec: XText >< XMLNode -~-> Text = R.exec,
  iswf: XNat >< XMLNode -> Bool = R.iswf,
  exec: XNat >< XMLNode -~-> Nat = R.exec,
  isRoot: XMLNode -> Bool = X.isRoot,
  getXMLPath: XMLNode -> XMLPath = X.getXMLPath,
  appendNewXMLTextChild: XMLPath >< XMLText >< XMLNode -~-> XMLNode
    = X.appendNewXMLTextChild,
  appendNewXMLElementChild: XMLPath >< Name >< XMLNode -~-> XMLNode
    = X.appendNewXMLElementChild,
  addNewXMLTextChild: XMLPath >< Nat >< XMLText >< XMLNode -~-> XMLNode
    = X.addNewXMLTextChild,
  addNewXMLElementChild: XMLPath >< Nat >< Name >< XMLNode -~-> XMLNode
    = X.addNewXMLElementChild,
  deleteXMLNode: XMLPath >< XMLNode -~-> XMLNode = X.deleteXMLNode,
  changeXMLNode: XMLPath >< XMLNode >< XMLNode -~-> XMLNode = X.changeXMLNode,
  isXMLElementNode: XMLNode -> Bool = X.isXMLElementNode,
  hasChild: Nat >< XMLNode -> Bool = X.hasChild,
  root: Text -> XMLNode = X.root

type
  XTree ==
    none |
    root(XText) |
    appendNewTextChild(XPath, XText) |
    appendNewElementChild(XPath, XText) |
    addNewTextChild(XPath, XNat, XText) |
    addNewElementChild(XPath, XNat, XText) |
```

```
    deleteNode(XPath) |
    changeNode(XPath, XNode)

value
  iswf: XTree >< XMLNode -> Bool
  iswf(r, x) is
    case r of
      none -> true,
      root(t) -> iswf(t, x),
      appendNewTextChild(p, t) ->
        iswf(p, x) /\ iswf(t, exec(p, x)) /\ isXMLElementNode(exec(p, x)),
      appendNewElementChild(p, t) ->
        iswf(p, x) /\ iswf(t, exec(p, x)) /\ isXMLElementNode(exec(p, x)),
      addNewTextChild(p, n, t) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t, x') /\ iswf(n, x') /\ isXMLElementNode(x') /\ hasChild(exec(n, x'), x')
        end,
      addNewElementChild(p, n, t) ->
        iswf(p, x) /\
        let x' = exec(p, x) in
          iswf(t, x') /\ iswf(n, x') /\ isXMLElementNode(x') /\ hasChild(exec(n, x'), x')
        end,
      deleteNode(p) -> iswf(p, x) /\ ~isRoot(exec(p, x)),
      changeNode(p, n) -> iswf(p, x) /\ iswf(n, exec(p, x))
    end

value
  exec: XTree >< XMLNode -~-> XMLNode
  exec(r, x) is
    case r of
      none -> x,
      root(t) -> root(exec(t, x)),
      appendNewTextChild(p, t) ->
        let x' = exec(p, x)
        in appendNewXMLTextChild(getXMLPath(x'), exec(t, x'), x) end,
      appendNewElementChild(p, t) ->
        let x' = exec(p, x)
        in appendNewXMLElementChild(getXMLPath(x'), exec(t, x'), x) end,
      addNewTextChild(p, n, t) ->
        let x' = exec(p, x)
        in addNewXMLTextChild(getXMLPath(x'), exec(n, x'), exec(t, x'), x) end,
      addNewElementChild(p, n, t) ->
        let x' = exec(p, x)
        in addNewXMLElementChild(getXMLPath(x'), exec(n, x'), exec(t, x'), x) end,
      deleteNode(p) -> deleteXMLNode(getXMLPath(exec(p, x)), x),
      changeNode(p, n) ->
        let x' = exec(p, x), x'' = exec(n, x')
        in changeXMLNode(getXMLPath(x'), x'', x) end
    end
    pre iswf(r, x)

end
```

Scheme 48: MODEL 10 - XMLFAMILY

```
../XML/XML, XMLALL

scheme XMLFAMILY(X:XML) = extend XMLALL(X) with class

type
  Name = X.Name,
  XBool = XB.XBool,
  XNode = XO.XNode,
  XPath = XP.XPath,
```

```
  XText = XT.XText,
  XNodes = XS.XNodes,
  XNat = XN.XNat,
  XMLNode = X.XMLNode
value
  and: XBool >< XBool -> XBool = XB.and,
  hasName: Name >< XNode -> XBool = XB.hasName,
  isEqualNat: XNat >< XNat -> XBool = XB.isEqualNat,
  self: XPath = XP.self,
  num: Nat -> XNat = XN.num,
  children: XNode -> XNat = XN.children,
  tt: XBool = XB.tt,
  isTrue: XBool >< XNode -> XBool = XB.isTrue,
  child: XNat >< XPath -> XPath = XP.child,
  isText: XNode -> XBool = XB.isText,
  filter: XNodes >< XBool -> XNodes = XS.filter,
  isEqualText: XText >< XText -> XBool = XB.isEqualText,
  text: Text -> XText = XT.text,
  getElementName: XNode -> XText = XT.getElementName,
  children: XPath -> XNodes = XS.children,
  one: XPath -> XNode = XO.one,
  getOne: XNat >< XNodes -> XNode = XO.getOne

value
  hasNameChildren: Name >< XBool-list -> XBool
  hasNameChildren(m, bl) is
    let b1 = hasName(m, one(self)), b2 = isEqualNat(num(len bl), children(one(self)))
    in if bl = <..> then and(b1, b2) else and(and(b1, b2), hasChildren(1, bl)) end end,

  hasChildren: Nat >< XBool-list -~-> XBool
  hasChildren(n, bl) is
    let b = isTrue(hd bl, one(child(num(n), self)))
    in if len bl = 1 then b else and(b, hasChildren(n+1, tl bl)) end end
    pre bl ~= <..>,

  hasNameText: Name -> XBool
  hasNameText(m) is and(hasName(m, one(self)), isText(one(self))),

  pathSequence: Nat-list -> XPath
  pathSequence(nl) is
    if nl = <..> then self else child(num(hd nl), pathSequence(tl nl)) end

value
  childrenWithName: Name -> XNodes
  childrenWithName(m) is
    filter(children(self), isEqualText(text(m), getElementName(one(self)))),

  firstChildWithName: Name -> XNode
  firstChildWithName(m) is getOne(num(1), childrenWithName(m))

end
```

Apéndice B – Requerimientos

Requirement 1: Managing the Visitor Member Type

identifier	F1
category	Functional
description	A Visitor member shall be implemented. Visitor members are restricted to exchange messages only with the Administrator member. All visitor members have a pre-assigned identifier. No other member can obtain a Visitor status. This member type is used for registering users.
terms	Member, Visitor, Administrator
justification	Allows new members to enter the system as Visitors and then register as regular members by submitting a request to the Administrator.
priority	High
feasibility	A Visitor is a particular type of member.
verification	Register a new member through the Visitor APIs.

Requirement 2: Managing the User Member Type

identifier	F2
category	Functional
description	A User member shall be implemented by the system as a regular identifiable member. A user can request the Administrator to create and destroy channels, to subscribe and unsubscribe to channels, to enable, configure and disable extensions and to unregister. A User can send and receive messages through channels as authorized by the channel owners. A User cannot create new channels or register new members by itself.
terms	Member, Administrator, User
justification	Allows regular members to access all the services provided by G-EEG-PROT.
priority	High
feasibility	A User is a particular type of member.
verification	Register a new member through the visitor APIs and unregister the recently registered member through the Member APIs.

Requirement 3: Managing the Administrator Member Type

identifier	F3
category	Functional
description	An Administrator member shall be implemented by the system as a distinguished member, who in addition to sending and receiving messages over channels, can authorize the registration of new members and the creation and configuration of channels. A single Administrator member is always present in the system. The Administrator member has a pre-assigned identifier. No other member can obtain the Administrator status. The Administrator can also manage communications between users.
terms	Member, Administrator, Membership Management
justification	Creates a privileged member who can use, develop and administer G-EEG-PROT using the same messaging paradigm for all.
priority	High
feasibility	An Administrator is a particular type of member.

verification	Register a new member through the visitor APIs. Check how the operation is fulfilled through messages exchanged between the visitor and the Administrator members.

Requirement 4: Registering a New Member

identifier	F4
category	Functional
description	The system shall implement an operation for registering new members. The request to carry out this operation is sent as a message from the Visitor to the Administrator providing the desired member name. The Administrator confirms if the operation was successful, otherwise the reason for failure, e.g. message incomplete. If successful, the result is a new user with the given name, who is identified by an identifier assigned by the Administrator.
terms	Member, User, Visitor, Administrator
justification	The operation enables to register a new user.
priority	High
feasibility	Feasible as long as a new identifier can be created.
verification	Register a new member through the visitor APIs. Check the messages exchanged and the new identifier assigned to the member.

Requirement 5: Unregistering an Existing Member

identifier	F5
category	Functional
description	The system shall implement an operation for unregistering an existing user. The operation is requested by the member who is willing to unregister by sending a message to the Administrator. The message carries the member's identifier. The Administrator confirms if the operation was successful or otherwise specifies the reason for failure. One common reason is the existence of resources (owned or subscribed channels) that the member has to release before unregistering. If successful, the requesting member is removed from the system. Visitors and Administrator members cannot be unregistered.
terms	Member, User, Visitor, Administrator
justification	The system shall provide an operation enabling an existing user to unregister from the system.
priority	High
feasibility	Always feasible.
verification	Register a new member through the visitor APIs and unregister the recently registered member through the member APIs.

Requirement 6: Recovering a Member

identifier	F6
category	Functional
description	The system shall implement an operation to recover the structure of a member. The request is sent to the Administrator by the Visitor member. After receiving the reply, the member structure shall be recovered by reading the local database and instantiating the objects required for managing the channels owned and subscribed by the member and all other required objects.
terms	Member, Visitor, Administrator, Channel Owned, Channel Subscribed
justification	Allows members to continue their operation after a system shutdown.
priority	High
feasibility	Always feasible.
verification	Register a member and create a channel. Stop the execution. Restart the member through

	the visitor APIs, and request the restarted member to send a message through the previously created channel.

Requirement 7: Managing Pre-Defined Administration Channels

identifier	F7
category	Functional
description	The system shall implement pre-defined Administration channels to connect each user with the Administrator. The channel owner is the Administrator and the user is the only member subscribed to the channel. Every user is free to use the Administration channel to carry out administrative operations, like requesting to create a channel, subscribe to a channel, enable an extension, etc. The Administrator channel must be created for each user upon member registration.
terms	Member, Administrator, Channel, Subscriber
justification	Administration channel is the medium for communicating users with the Administrator, for them to request administrative services.
priority	High
feasibility	Always feasible.
verification	Register a new member and send a request to the Administrator to create a channel. Check that the request was sent through the Administrator channel.

Requirement 8: Managing Pre-Defined Visitor Channels

identifier	F8
Category	Functional
description	The system shall implement pre-defined Visitor channels to connect visitor members with the Administrator. The Administrator is the owner of the channel and visitor is automatically subscribed to the channel. No other member can subscribe to the Visitor channel. The only messages allowed to flow through the Visitor channel are messages for member registration, for recovery the member's structure and their replies.
Terms	Member, Administrator, Channel, Subscriber
justification	Visitor channel is the medium for registering new users and recovering existing ones.
Priority	High
feasibility	Always feasible.
verification	Register a new member and check that the registration message and its reply were exchanged through the Visitor channel.

Requirement 9: Creating a Channel

identifier	F9
category	Functional
description	The system shall implement an operation for creating a channel. The operation can be requested by users or Administrator members. Users' requests are sent to the Administrator for approval. The channel name is specified by the user upon the channel creation. If the creation is successful, a unique identifier is assigned to the channel by the Administrator. The user who issued the request becomes the owner of the channel. The channel owner is authorized to register new subscribers and to destroy the channel. Initially, the channel has no subscribers.
terms	Channel, Channel Owner, Administrator
justification	Permits communication channels to be opened dynamically.
priority	High
feasibility	Feasible as long as a new identifier can be created.

verification	Register a new member and request to create a channel.

Requirement 10: Destroying a Channel

identifier	F10
category	Functional
description	The system shall implement an operation to destroy a channel. The operation is requested by the channel owner and sent to the Administrator. Before the actual destruction, a message is sent to all channel subscribers to inform them about the channel closure. Upon reception of this message, subscribers must require unsubscribing to the channel. A channel cannot be destroyed if it has a subscriber.
terms	Channel, Channel Owner, Subscriber, Administrator
justification	Permits communication channels to be destroyed dynamically.
priority	High
feasibility	Always feasible.
verification	Register a new member and create a channel. Register another member and subscribe it to the channel. As channel owner, request to destroy the channel. The subscriber must receive a message informing that the channel will be destroyed. As channel subscriber, request to unsubscribe from the channel. As channel owner, request again to destroy the channel. As subscriber, check that the channel is destroyed by trying to send a message through the channel.

Requirement 11: Subscribing a Member to a Channel

identifier	F11
category	Functional
description	The system shall implement an operation to authorize members to access a channel. The operation is requested by a member who is not yet authorized to use the channel by sending a message to the channel owner through their administrator channels. The owner is free to implement its own policy for channel subscription and responds to the member reporting success or failure of the request. After successfully subscribing a user to a channel, the user can send and receive messages through the channel.
terms	Channel, Channel Owner, Channel Subscriber, Right to Read
justification	Allows members to subscribe to channels dynamically.
priority	High
feasibility	Always feasible.
verification	Register a new member and create a channel. Register a new member and request to subscribe this member to the channel recently created by the other member.

Requirement 12: Unsubscribing a Member from a Channel

identifier	F12
category	Functional
description	The system shall implement an operation enabling a member to unsubscribe from a channel. The operation is requested by an existing channel subscriber and sent to the channel owner through their administrator channels. The channel subscriber sends the request to the Administrator through its administrator channel, and the Administrator forwards the request to the channel owner through the owner's administrator channel. The owner reports success or failure of the operation to the subscriber. For instance, a reason of failure is that the user is not subscribed to the channel. After successfully unsubscribing a member from a channel, the member cannot access the channel any more – send or receive messages through it, but can still access other channels it owns or subscribes, included the administrator channel.
terms	Channel, Channel Owner, Channel Subscriber, Subscription Channel

justification	Allows members to unsubscribe to channels dynamically.
priority	High
feasibility	Always feasible.
verification	Register a new member and create a channel. Register a new member and subscribe this member to the recently created channel. As subscriber, send a message through the channel. As channel owner, receive the message. As subscriber, unsubscribe from the channel and try to send another message through the channel. The last operation should fail.

Requirement 13: Composing a Message

identifier	F13
category	Functional
description	The system shall implement an operation to compose a message from individual parts: header, body and attachments. The header and body parts are mandatory, while attachments may repeat any number of times, including zero. The operation returns the complete, ready to send message. It is performed locally.
terms	Message, Header, Body, Attachment
justification	Allows members to create messages to be sent
priority	Medium
feasibility	Always feasible.
verification	Register a member and create a channel. Register another member, subscribe it to the channel, create a message and include an attachment, and send the message through the channel. As channel owner, receive the message previously sent and check the message parts.

Requirement 14: Obtaining Message Header

identifier	F14
category	Functional
description	The system shall implement an operation to extract the header from a given message. The operation is performed locally.
terms	Message, Header
justification	The information included in the message header enables the Gateway to distribute messages among members through channels.
priority	Medium
feasibility	Always feasible.
verification	Register a member and create a channel. Register a new member and create another channel. Check the header of the messages received by both members.

Requirement 15: Obtaining Message Body

identifier	F15
category	Functional
description	The system shall implement an operation to extract the body from a given message. The operation is performed locally.
terms	Message, Body
justification	Allows members to extract the body from any messages received.
priority	Medium
feasibility	Always feasible.
verification	Register a member and create a channel. Register a new member and subscribe it to the recently created channel. As channel owner, send a message. As channel subscriber, receive the message. Check the message body.

Requirement 16: Obtaining Message Attachments

identifier	F16
category	Functional
description	The system shall implement an operation to extract the list of attachments from a given message. The operation is performed locally.
terms	Message, Attachment
justification	Allows members to extract attachments from any messages received.
priority	Medium
feasibility	Always feasible.
verification	Register a member and create a channel. Register a new member and subscribe it to the recently created channel. As channel owner, send a message with two attachments. As channel subscriber, receive the message. Check the received attached files.

Requirement 17: Sending a Message by the Channel Owner

identifier	F17
category	Functional
description	The system shall implement an operation enabling a channel owner to send a message to all subscribers of a given channel. The operation processes the message and upon successful completion, the message is forwarded to all channel subscribers. A failure occurs if the channel is closed or the processing of the message is incorrect. In case of failure, the message is rejected and sent back to the sender. The rejected message includes the reason of failure. The process of the message is performed locally.
terms	Message, Owner, Subscriber, Channel
justification	Allows channel owners to send messages through channels.
priority	High
feasibility	Always feasible according to G-EEG specifications.
verification	Register a member and create a channel. Register another member and subscribe it to the created channel. As channel owner, send a message through the channel. As channel subscriber, receive the message.

Requirement 18: Sending a Message by the Channel Subscriber

identifier	F18
category	Functional
description	The system shall implement an operation enabling a channel subscriber to send a message through the channel. When a channel subscriber sends a message, the message is forwarded through the channel, to the channel owner. The channel owner processes the message according to behavior specified in Requirement F16.
terms	Message, Subscriber, Owner, Channel
justification	Allows channel subscribers to send messages through channels.
priority	High
feasibility	Always feasible according to G-EEG specifications.
verification	Register a member and create a channel. Register another member and subscribe it to the created channel. As channel subscriber, send a message through the channel. As channel owner, receive the message.

Requirement 19: Receiving a Message

identifier	F19
category	Functional

description	The system shall implement an operation enabling a member to receive a message from a given channel. The application using G-EEG shall implement a listener for receiving messages. A failure occurs if the channel is closed.
terms	Message, Subscriber, Owner, Channel
justification	Allows members to receive messages from channels.
priority	High
feasibility	Always feasible according to G-EEG specifications.
verification	Register a member and create a channel. Register another member, subscribe it to the created channel and send a message. As channel owner, receive the message.

Requirement 20: Forwarding a Message between Members

identifier	F20
category	Functional
description	The system shall implement an operation enabling a member to forward a message to another member. Message exchange between two members can be carried out via their administration channels through the Administrator.
terms	Member, Message
justification	Allows members to exchange messages between them without need to create additional channels.
priority	High
feasibility	Always feasible based on the administration channels
verification	Register two members. As the first member registered, forward a message to the other member through the administration channel. As the second member registered, receive the message through the administration channel.

Requirement 21: Managing a Repository of Extensions

identifier	F21
category	Functional
description	The system shall implement a repository (G-EEG-REPOS) for storing data about the extensions deployed in the system. After deploying a new extension, the repository of extensions shall be updated accordingly. Data about the extension include extension type – channel-oriented or member-oriented, and number of required parameters, among others.
terms	Repository, Extension
justification	Allows the extensibility of G-EEG.
priority	High
feasibility	Updating a database is a basic functionality provided by software applications.
verification	Check the data stored in the repository about available extensions. Simulate deploying a new extension and update the repository with data about the new extension.

Requirement 22: Enabling Extensions

identifier	F22
category	Functional
description	The system shall implement a function to dynamically enable an extension. Only extensions existing in G-EEG-REPOS can be enabled. Channel-oriented extensions can only be enabled by the channel owner. Member-oriented extensions can be enabled by any member who automatically becomes responsible for managing the extension. Vertical extensions coordinating the flow of messages through several channels can only be enabled by the channel owner of one of the coordinated channels. This function shall create all the required

	communication structures to ensure that the extension behavior can be fulfilled.
terms	Extension, Channel-Oriented Extension, Member-Oriented Extension, Vertical Extension
justification	Allows the extensibility of G-EEG.
priority	High
feasibility	Always feasible according to G-EEG specification.
verification	Register a member, create a channel and enable a channel-oriented extension. Register another member, subscribe it to the channel created and send a message through the channel. Check that the message received by the channel owner was processed according to the enabled extension.

Requirement 23: Configuring Extensions

identifier	F23
category	Functional
description	The system shall implement a function to dynamically configure an extension. Only the member who enabled the extension can configure it. The mechanism for configuring extensions should support any numbers of parameters.
terms	Extension, Parameters
justification	Extensions may require parameters for being able to carry out their functionality. For instance, Validation requires a document defining the structure of messages to be used for validating messages.
priority	High
feasibility	Requires managing the parameters by the extension and keeping the parameters in a database. The former is feasible according to G-EEG specification and the latter include basic functions of managing persistence by software applications.
verification	Register a member, create a channel, enable the Validation extension and configure it. Register another member, subscribe it to the channel created and send a valid message through the channel. As channel owner, receive the valid message, send an invalid message and receive an error message.

Requirement 24: Disabling Extensions

identifier	F24
category	Functional
description	The system shall implement a function to dynamically disable an extension. Only enabled extensions can be disabled. Only the member who enabled the extension can disable it. Once the extension is disabled, the functionality provided by it is not further available. The data about the disabled extension stored in G-EEG-REPOS should remain unchanged. The extension can be later enabled by the same or another member involving the same or different channel(s) and member(s).
terms	Extension
justification	Allows the extensibility of G-EEG.
priority	High
feasibility	Always feasible according to G-EEG specification.
verification	Register a member, create a channel, enable the Validation extension and disable the extension. Register another member, subscribe to the channel created and send an invalid message through the channel. Check the message received was not processed by the extension (the invalid message was received by the channel owner).

Requirement 25: Providing Channel Auditing Extension

identifier	F25

category	Functional
description	The system shall implement an extension enabling to store all message exchanged through a channel. The storage mechanism shall allow later retrieval of messages. The extension can be configured with the name of the database used for storing messages.
terms	Extension, Channel-Oriented Extension, Auditing
justification	Enables to log messages exchanged through a channel.
priority	High
feasibility	Always feasible according to G-EEG specification.
verification	Register a member, create a channel and enable the Auditing extension. Register another member, subscribe it to the channel and send a message through the channel. Check that the message was stored in a database.

Requirement 26: Providing Message Validation Extension

identifier	F26
category	Functional
description	The system shall implement an extension enabling to validate the syntax of messages exchanged through a channel. The extension must be configured with the accepted message format to be used for validating messages. As messages are written in XML, the message format can be defined using XML Schema. If the message is valid, the message shall be delivered to the channel owner and all channel subscribers. If the message is invalid, the message sender shall receive an acknowledgement and the message will not be delivered.
terms	Extension, Channel-Oriented Extension, Validation
justification	Enables to validate the syntax of messages exchanged through a channel.
priority	High
feasibility	Always feasible according to G-EEG specification.
verification	Register a member, create a channel, and enable the Validation extension. Configure the validation extension with the XML Schema specifying the valid structure of messages. Register another member, subscribe it to the channel, and send a valid message through the channel. Check the message sent is received by the channel owner. Send an invalid message and receive a message acknowledging that the message sent is not valid. Check that the channel owner does not receive the message.

Requirement 27: Providing Message Transformation Extension

identifier	F27
category	Functional
description	The system shall implement an extension enabling to transform messages exchanged through a channel. As messages are written in XML, the message transformation can be specified through an XSLT template. If the transformation results successful, the message will be delivered to the channel owner and all subscribers. If the transformation fails, the message sender will be acknowledged and the message will not be delivered.
terms	Extension, Channel-Oriented Extension, Transformation
justification	Enables to transform the syntax and language of messages exchanged through a channel.
priority	High
feasibility	Always feasible according to G-EEG specification.
verification	Register a member, create a channel, and enable the Transformation extension. Configure the transformation extension with an XSLT template. Register another member, subscribe it to the channel and send a message through the channel. As channel owner, check that the message received was transformed based on the XSLT used for configuring the extension.

Requirement 28: Availability of the Administrator Member

identifier	NF28
category	Non-functional
description	The administrator member must always be available.
terms	Availability, Administrator
justification	Users must always be able to communicate with the Administrator member.
priority	High
feasibility	Hardware components can be installed to accomplish this requirement.
verification	Test the availability of the Administrator member while the server in which the member is deployed is running out of power.

Requirement 29: Assuring At-Least-Once Delivery

identifier	NF29
category	Non-functional
description	The system shall provide a reliable procedure for delivering messages. All messages sent must be delivered at least once. Messages sent cannot be lost.
terms	Reliability
justification	Members must trust that the messages sent are delivered.
priority	Medium
feasibility	It can be assured through the use of storage mechanisms and technologies providing reliable communication.
verification	Register a new member and create a channel. Register another member, subscribe it to the created channel and stop the member. As channel owner, send a message. As channel subscriber, restart the member and receive the message.

Requirement 30: Assuring Non-Permutation for Message Delivery

identifier	NF30
category	Non-functional
description	The system shall deliver messages in sequence. If a member sends several messages through the same channel, messages must be delivered in the same order as they were sent.
terms	Reliability
justification	Members must trust G-EEG-PROT delivers messages in a predictable way.
priority	High
feasibility	It can be attained by control mechanisms and data structures used during implementation.
verification	Register a member and create a channel. Register another member, subscribe it to the created channel and send three messages. As channel owner, receive the messages in the same order as they were sent.

Apéndice C – Artefactos de Desarrollo

Figura 79: Diagrama de Casos de Uso – Suscribir Miembro a Canal

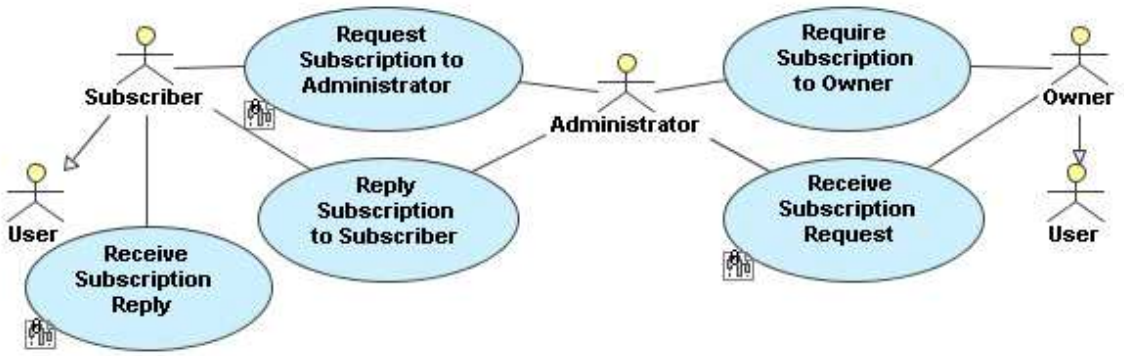


Figura 80: Diagrama de Casos de Uso – Dessuscribir Miembro de Canal

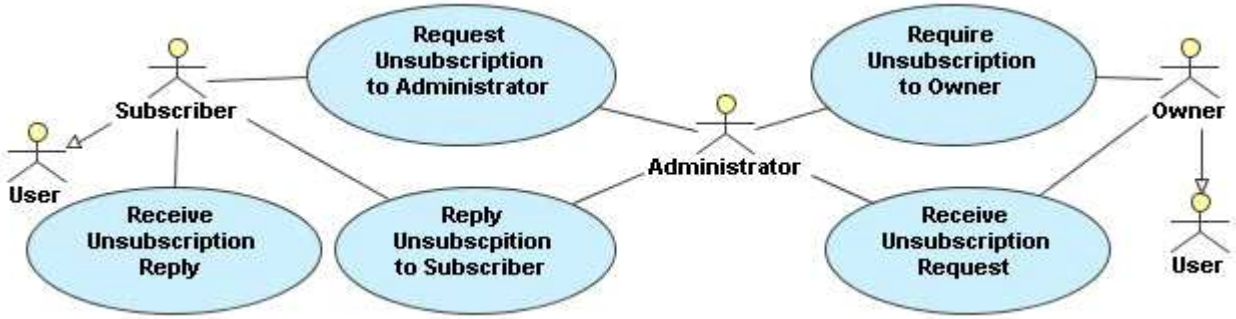


Figura 81: Arquitectura – Vista Estructural de Alto Nivel

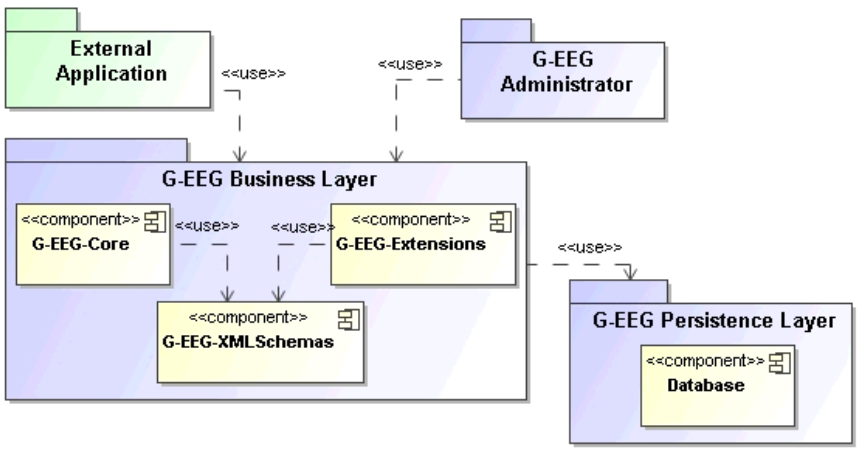


Figura 82: Diagram de Secuencia – Creación de un Canal – Interacciones del Miembro

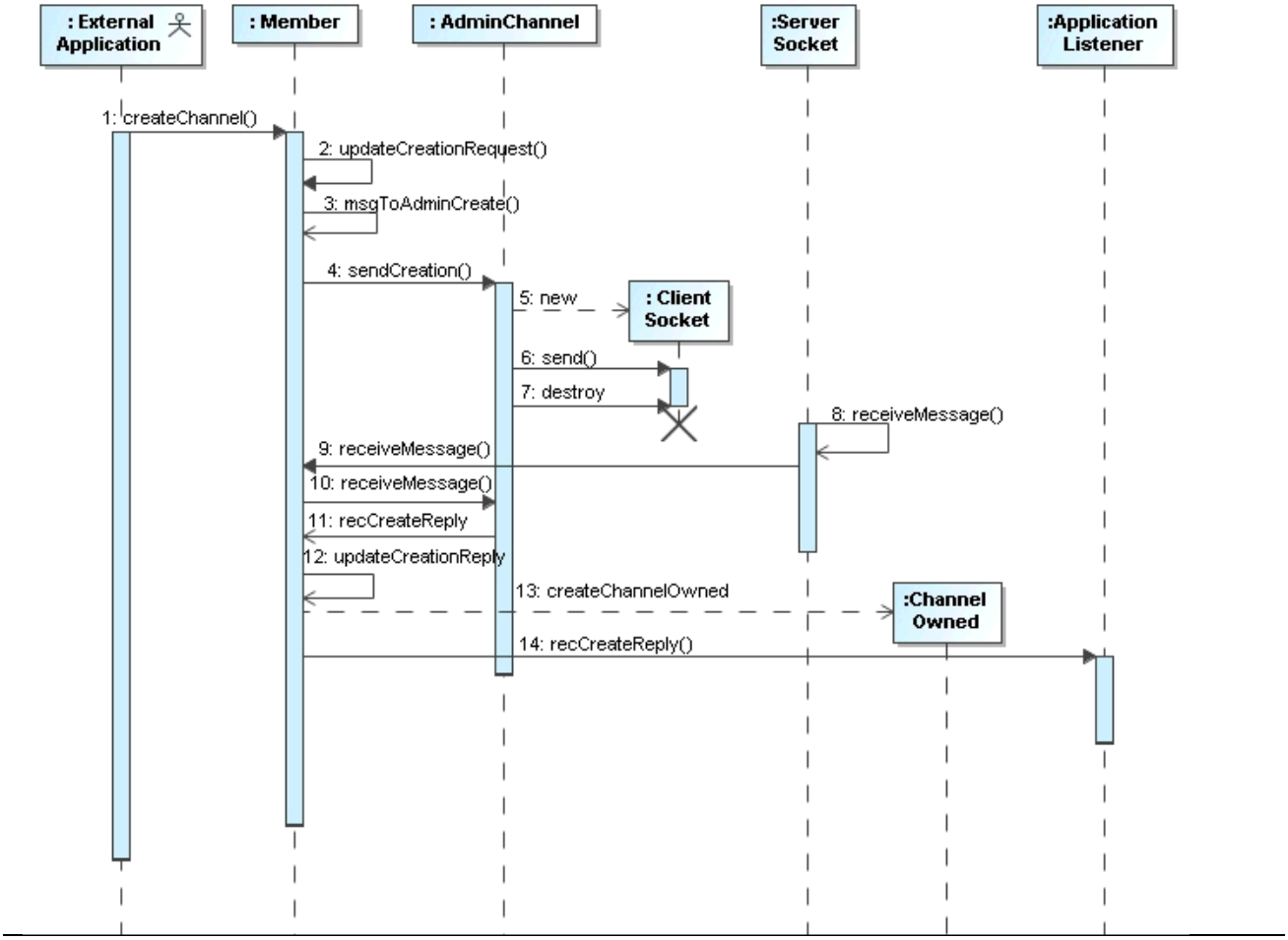


Figura 83: Diagram de Secuencia – Creación de un Canal – Interacciones del Administrador

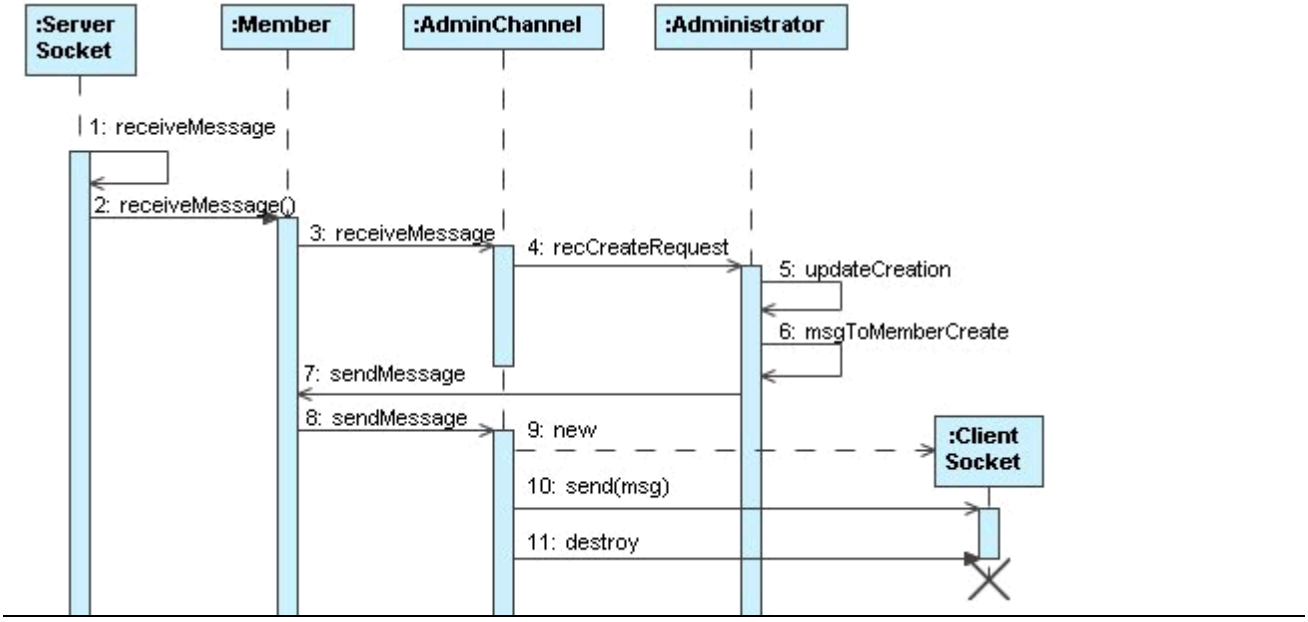


Figura 84: Diagrama de Secuencia – Envío de Mensaje como Suscriptor de Canal

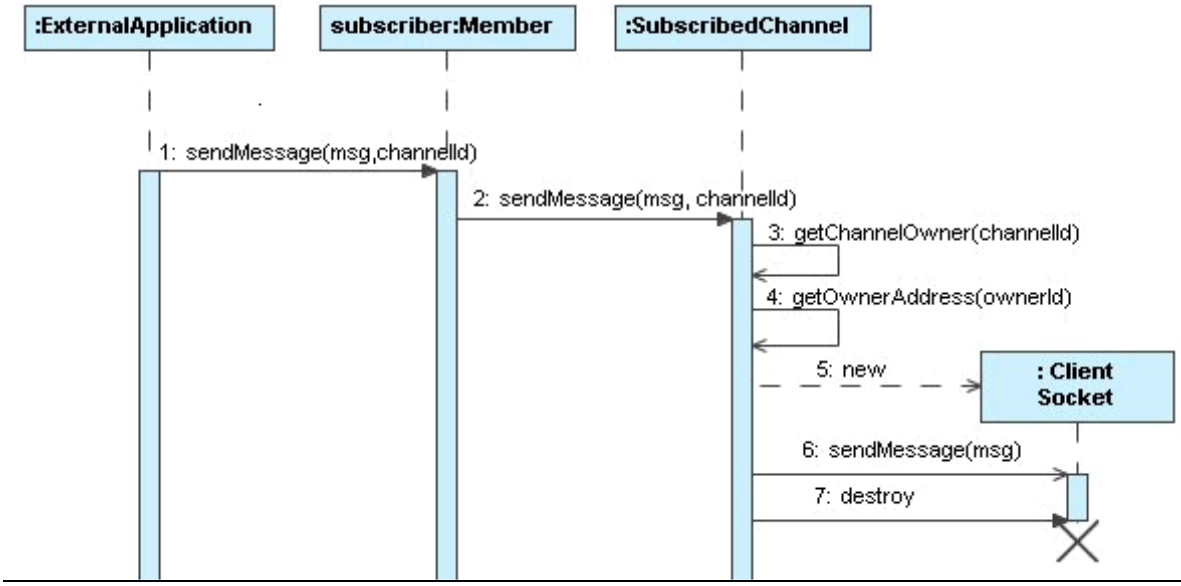


Figura 85: Diagrama de Secuencia – Envío de Mensaje como Dueño de Canal

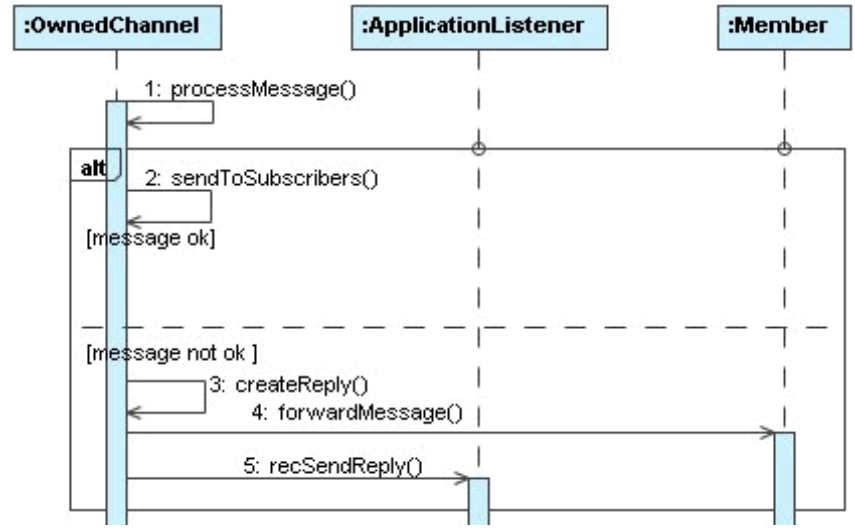


Figura 86: Diagrama de Secuencia – Recepción de Mensaje

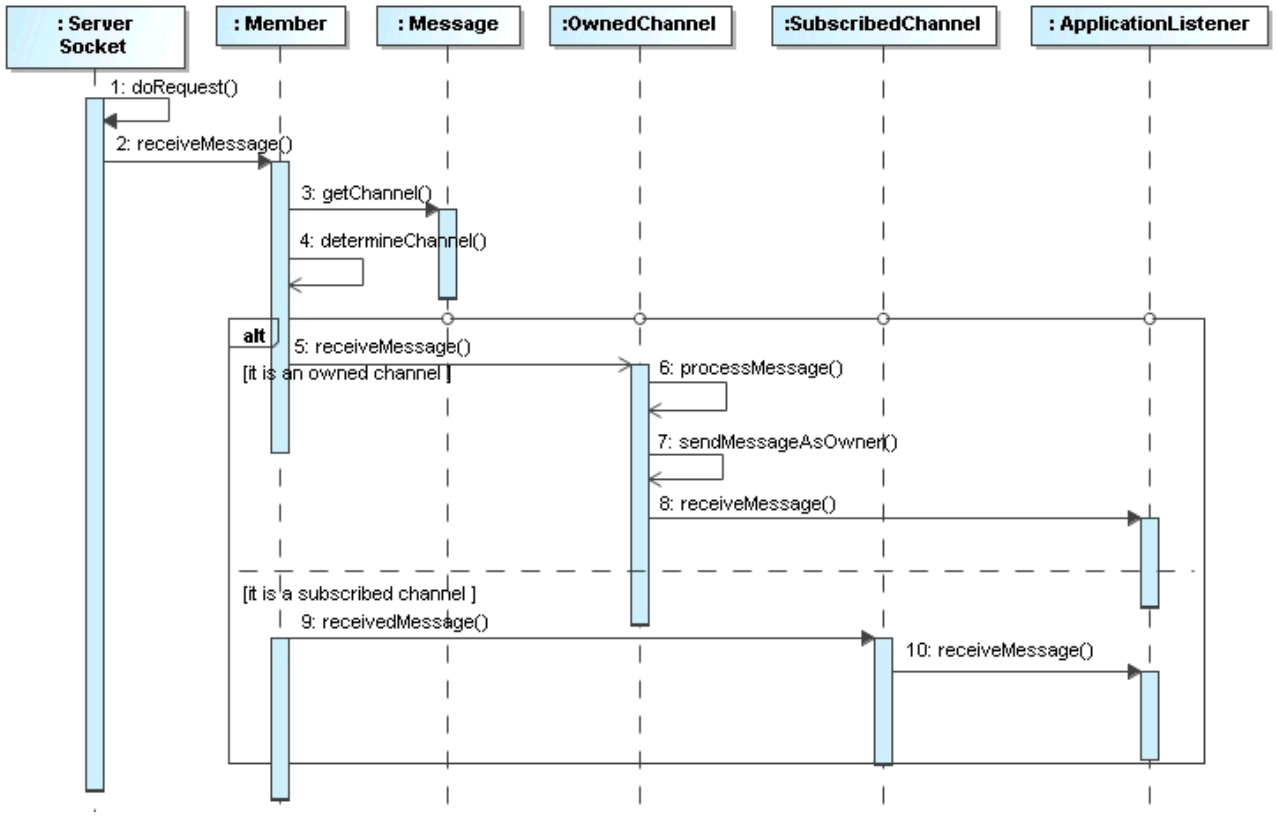


Figura 87: Diagrama de Secuencia – Destrucción de Canal – Requerimiento del Miembro

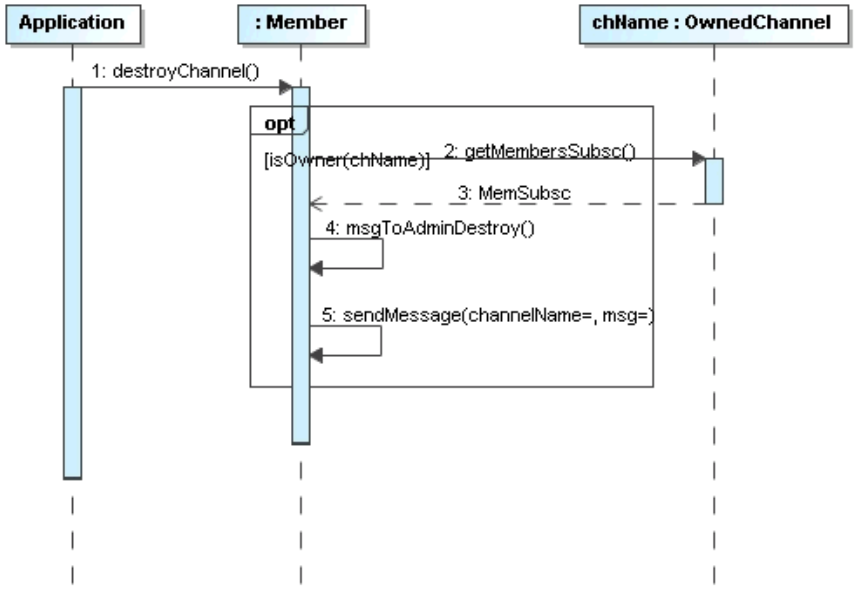


Figura 88: Diagrama de Secuencia – Destrucción de Canal – Recepción del Requerimiento por Dueño del Canal

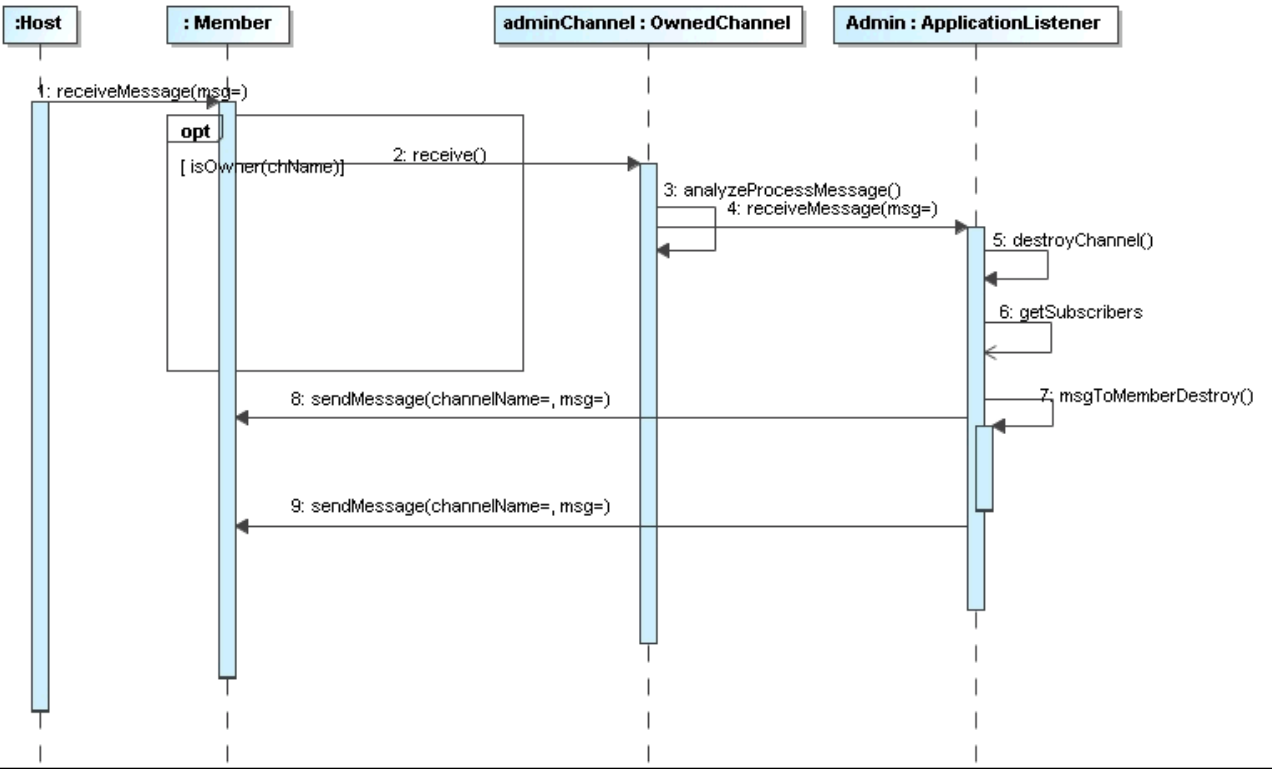


Figura 89: Diagrama de Secuencia – Destrucción de Canal – Recepción de Requerimiento por Suscriptor

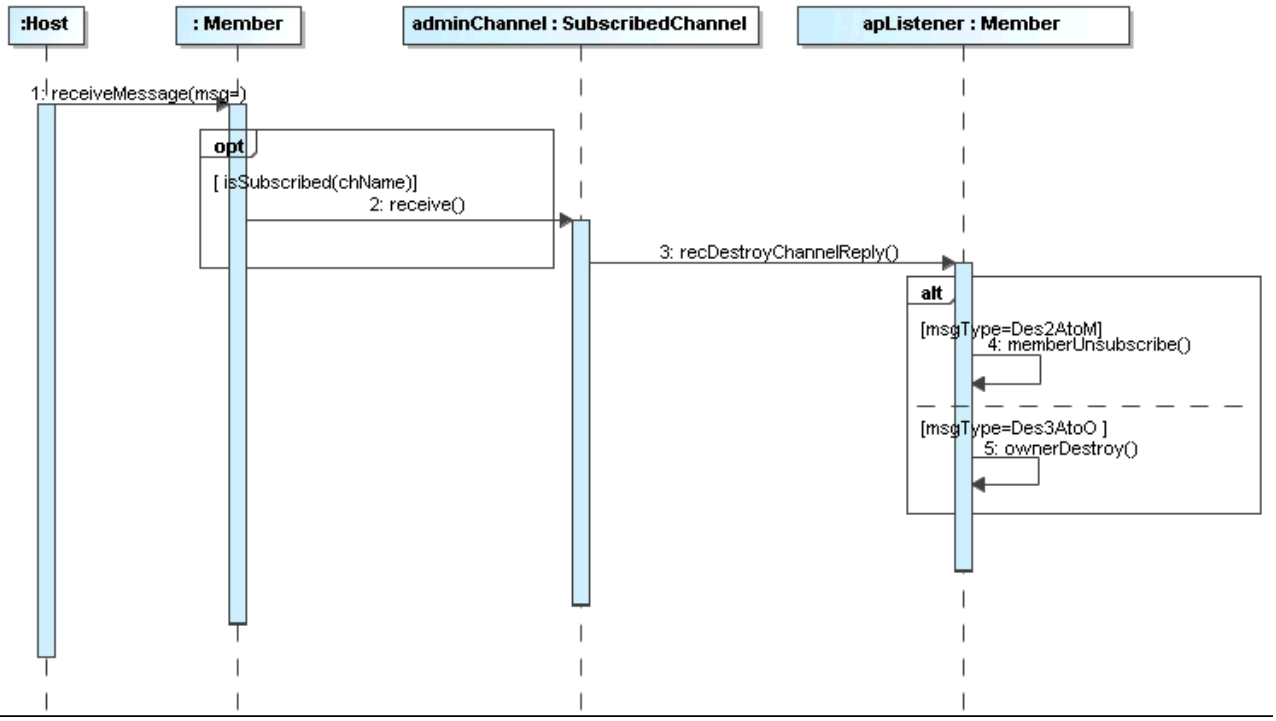


Figura 90: Diagrama de Secuencia – Habilitación de Extensión de Validación

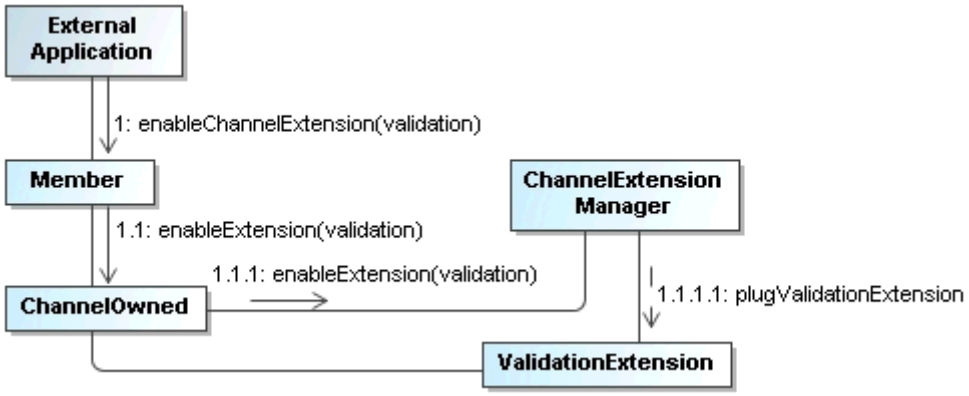
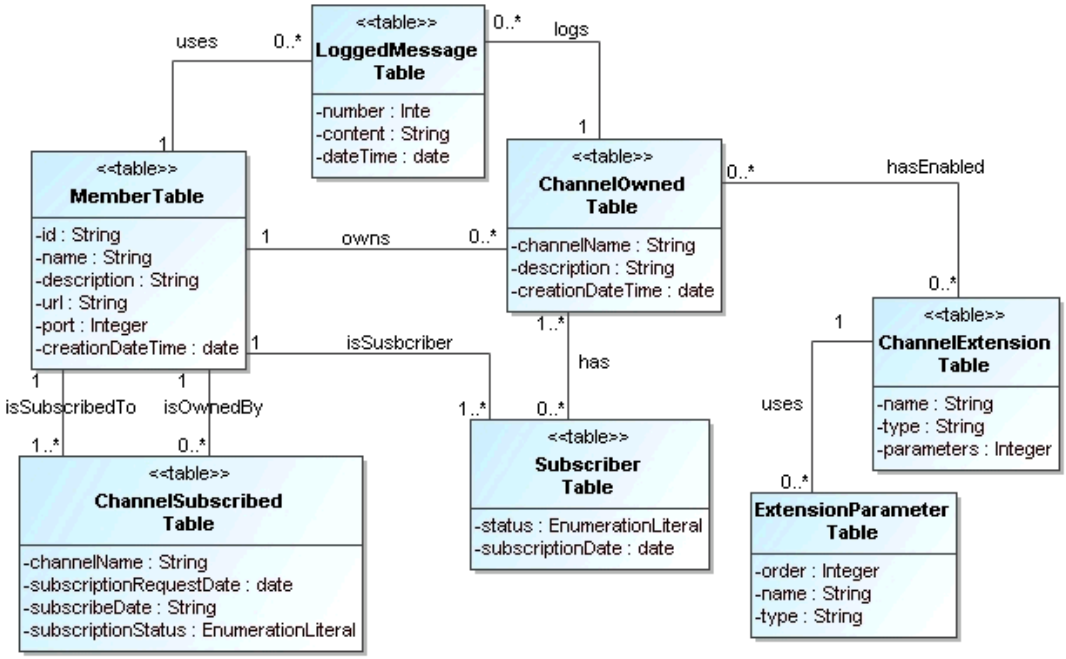


Figura 91: Diseño de Base de Datos



Apéndice D – Artefactos de Implementación

Figura 92: Archivo de Configuración de G-EEG Prototype

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This schema defines the format of messages handled by XG2G -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://iist.unu.edu/emacao/gateway/xmlUtil"
  elementFormDefault="qualified">

<xs:element name="Parameters">
<xs:complexType>
  <xs:sequence>
    <xs:element name="AdminIdFile"          type="xs:string" />
    <xs:element name="AdminNameFile"        type="xs:string" />
    <xs:element name="AdminUrlFile"         type="xs:string" />
    <xs:element name="AdminPortFile"        type="xs:int" />
    <xs:element name="AnonIdFile"           type="xs:string" />
    <xs:element name="AnonNameFile"         type="xs:string" />
    <xs:element name="AnonPortFile"         type="xs:int" />
    <xs:element name="AnonChannelIdFile"    type="xs:string" />
    <xs:element name="MaxChOwnedFile"       type="xs:int" />
    <xs:element name="MaxChSubscribedFile"  type="xs:int" />
    <xs:element name="MaxChExtensionsFile"  type="xs:int" />
    <xs:element name="MaxBufferSizeFile"    type="xs:int" />
    <xs:element name="MaxSubscribFile"      type="xs:int" />
    <xs:element name="InitialPortFile"      type="xs:int" />
    <xs:element name="FinalPortFile"        type="xs:int" />
    <xs:element name="InitMemberIdFile"     type="xs:int" />
    <xs:element name="InitChannelIdFile"    type="xs:int" />
  </xs:sequence>
</xs:complexType>
</xs:element>

</xs:schema>
```

Figura 93: G-EEG Prototype – Formato de Mensaje en XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This schema defines the format of messages handled by XG2G -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xg2g="http://iist.unu.edu/emacao/gateway/xmlUtil"
  targetNamespace="http://iist.unu.edu/emacao/gateway/xmlUtil"
  elementFormDefault="qualified">

<xs:element name="Message">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Header" type="xg2g:header"/>
      <xs:element name="Body"   type="xg2g:body"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="header">
  <xs:sequence>
```

```
    <xs:element name="ChannelId"   type="xg2g:channelId"/>
    <xs:element name="MessageType" type="xg2g:messageType"/>
    <xs:element name="MessageStep" type="xg2g:messageStep"/>
    <xs:element name="SenderId"    type="xg2g:memberId"/>
    <xs:element name="ReceiverId"  type="xg2g:memberId"/>
    <xs:element name="MessageId"   type="xg2g:messageId"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="body">
  <xs:choice>
    <xs:element name="UserMessage"  type="xs:anyType"/>
    <xs:element name="Registration" type="xg2g:registration" />
    <xs:element name="Creation"     type="xg2g:creation" />
    <xs:element name="Subscribe"    type="xg2g:subscribe" />
    <xs:element name="Unsubscribe"  type="xg2g:subscribe" />
    <xs:element name="Destroy"      type="xg2g:destroy" />
    <xs:element name="Unregister"   type="xg2g:unregister" />
    <xs:element name="Extension"    type="xg2g:extension" />
    <xs:element name="Manage"       type="xg2g:manage" />
    <xs:element name="Forward"      type="xg2g:forward" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="registration">
  <xs:sequence>
  <xs:element name="MemberName"  type="xs:string" />
  <xs:element name="AnonUrl"     type="xs:string" />
  <xs:element name="AnonPort"    type="xs:int" />
  <xs:element name="MemberId"    type="xg2g:memberId" minOccurs="0" maxOccurs="1"/>
  <xs:element name="PrivChannel" type="xg2g:channelId" minOccurs="0" maxOccurs="1"/>
  <xs:element name="MemberPort"  type="xs:int" minOccurs="0"  maxOccurs="1" />
  <xs:element name="Status"      type="xg2g:status" minOccurs="0"  maxOccurs="1"/>
  <xs:element name="RequestId"   type="xg2g:messageId" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="creation">
  <xs:sequence>
  <xs:element name="MemberId"    type="xg2g:memberId" />
  <xs:element name="ChannelName" type="xs:string" />
  <xs:element name="ChannelDbId" type="xs:long" />
  <xs:element name="ChannelId"   type="xg2g:channelId" minOccurs="0" maxOccurs="1"/>
  <xs:element name="Status"      type="xg2g:status"    minOccurs="0" maxOccurs="1"/>
  <xs:element name="RequestId"   type="xg2g:messageId" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="subscribe">
<xs:sequence>
  <xs:element name="MemberId"    type="xg2g:memberId" />
  <xs:element name="MemberUrl"   type="xs:string"  />
  <xs:element name="MemberPort"  type="xs:int" />
  <xs:element name="ChannelId"   type="xg2g:channelId" />
  <xs:element name="ChannelDbId" type="xs:long" />
  <xs:element name="OwnerId"     type="xg2g:memberId" />
  <xs:element name="OwnerUrl"    type="xs:string" minOccurs="0" maxOccurs="1"/>
  <xs:element name="OwnerPort"   type="xs:int" minOccurs="0" maxOccurs="1"/>
  <xs:element name="Status"      type="xg2g:status" minOccurs="0"  maxOccurs="1"/>
  <xs:element name="RequestId"   type="xg2g:messageId" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="destroy">
  <xs:sequence>

    <xs:element name="ChannelId"  type="xg2g:channelId" />
```

```
    <xs:element name="OwnerId"    type="xg2g:memberId" />
    <xs:element name="Status"     type="xg2g:status" minOccurs="0" maxOccurs="1"/>
    <xs:element name="RequestId"  type="xg2g:messageId" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="unregister">
  <xs:sequence>
    <xs:element name="MemberId"  type="xg2g:memberId" />
    <xs:element name="Status"    type="xg2g:status" minOccurs="0" maxOccurs="1"/>
    <xs:element name="RequestId" type="xg2g:messageId" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="extension">
  <xs:sequence>
  <xs:element name="ExtType"   type="xg2g:extType" />
  <xs:element name="ExtAction" type="xg2g:extAction" />
  <xs:element name="ExtName"   type="xs:string" />
  <xs:element name="MemberId"  type="xg2g:memberId" minOccurs="0"  maxOccurs="1"/>
  <xs:element name="ChannelId" type="xg2g:memberId" minOccurs="0"  maxOccurs="1"/>
  <xs:element name="Status"    type="xg2g:status"   minOccurs="0"  maxOccurs="1"/>
  <xs:element name="RequestId" type="xg2g:messageId" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="manage">
  <xs:sequence>
    <xs:element name="Action"    type="xg2g:actionType" />
    <xs:element name="RequestId" type="xg2g:messageId" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="forward">
  <xs:sequence>
  <xs:element name="MemberId"    type="xg2g:memberId" />
  <xs:element name="MemberUrl"   type="xs:string"  minOccurs="0"  maxOccurs="1" />
  <xs:element name="MemberPort"  type="xs:int"     minOccurs="0"  maxOccurs="1" />
  <xs:element name="MsgToMember" type="xs:anyType" minOccurs="0"  maxOccurs="1" />
  <xs:element name="RequestId"   type="xg2g:messageId" />
</xs:sequence>
</xs:complexType>

<xs:simpleType name="channelId">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="messageType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Register" />
    <xs:enumeration value="Create" />
    <xs:enumeration value="Subscribe" />
    <xs:enumeration value="Unsubscribe" />
    <xs:enumeration value="Destroy" />
    <xs:enumeration value="Unregister" />
    <xs:enumeration value="Extend" />
    <xs:enumeration value="Manage" />
    <xs:enumeration value="User-Defined" />
    <xs:enumeration value="Forward" />
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="messageStep" >
  <xs:restriction base="xs:int">
    <xs:minInclusive value="1" />
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="memberId">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="messageId">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="1" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="status">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="extType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Channel" />
    <xs:enumeration value="Member" />
    <xs:enumeration value="Vertical" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="extAction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Enable" />
    <xs:enumeration value="Disable" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="actionType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Enable" />
    <xs:enumeration value="Disable" />
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Figura 94: G-EEG Prototype – Formato de Mensaje de Respuesta en XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This schema defines the format of messages handled by XG2G -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xg2g="http://iist.unu.edu/emacao/gateway/xmlUtil"
  targetNamespace="http://iist.unu.edu/emacao/gateway/xmlUtil"
  elementFormDefault="qualified">

<xs:element name="ReplyMessage">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ReplyBody" type="xg2g:msgReplyBody"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="msgReplyBody">
  <xs:sequence>
    <xs:element name="MsgReplyType" type="xg2g:msgReplyType"/>
    <xs:element name="ReplyStatus"  type="xg2g:replyStatus" />
    <xs:element name="Comment"      type="xs:string" minOccurs="0" maxOccurs="1" />

    <xs:element name="MsgRepChId"
```

```
      type="xg2g:msgRepChId" minOccurs="0" maxOccurs="1"/>
    <xs:element name="MsgRepMsgId"
      type="xg2g:msgRepMsgId" minOccurs="0" maxOccurs="1" />
    <xs:element name="MsgToMember" type="xs:anyType" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="msgReplyType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="RegisterReply" />
    <xs:enumeration value="CreateReply" />
    <xs:enumeration value="SendReply" />
    <xs:enumeration value="ReceiveReply" />
    <xs:enumeration value="SubscribeReply" />
    <xs:enumeration value="UnsubscribeReply" />
    <xs:enumeration value="DestroyReply" />
    <xs:enumeration value="UnregisterReply" />
    <xs:enumeration value="ExtendReply" />
    <xs:enumeration value="ManageReply" />
    <xs:enumeration value="User-DefinedReply" />
    <xs:enumeration value="ForwardReply" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="replyStatus">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="msgRepChId">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="msgRepMsgId">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="1" />
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Figura 95: G-EEG Prototype – Mensajes Definidos por el Usuario en XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This schema defines the format of messages handled by XG2G -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xg2g="http://iist.unu.edu/emacao/gateway/xmlUtil"
  targetNamespace="http://iist.unu.edu/emacao/gateway/xmlUtil"
  elementFormDefault="qualified">

<xs:element name="UserMessage" type="xs:anyType" />

</xs:schema>
```

Figura 96: Interface de Visitor

```
package edu.unu.iist.geeg.core;
public interface IVisitor {

  /* service for registering a new member */
  public Member registerMember(String name, ApplicationListener el, int port);

  /* service for recovering the member structure */
  public Member getMember(String mId, ApplicationListener el, int port);

}
```

Figura 97: Interface de Servicios Administrativos Básicos

```
package edu.unu.iist.geeg.core;
public interface ICoreAdminServices {

  /* service for creating a channel */
  public void createChannel(String chName);

  /* service for detroying a channel */
  public void destroyChannel(String chId);

  /* service for subscribying to a channel */
  public void subscribeChannel(String chId, String oId);

  /* service for unsubscribing from a channel */
  public void unsubscribeChannel(String chId, String oId);

  /* service for unregistering the member */
  public void unregisterMember();

  /* service for forwarding the message to other member */
  public void forwardMessage(String mId, String msg);

}
```

Figura 98: Interface de Servicios Operacionales Básicos

```
package edu.unu.iist.geeg.core;
public interface ICoreOperServices {

  /* Service for sending a message */
  public void sendMessage(String chId, String msg);

  /* Service for receiving a message from Host */
  public void receiveMessage(String msg);

  /* Service for forwarding the message to other member */
  public void forwardMessage(String mId, String msg);

}
```

Figura 99: Listener de la Aplicación

```
package edu.unu.iist.geeg.core;
public interface ApplicationListener {
```

```
  /* Service for receiving a message */
  public void receiveMessage(String msg);

  /* Service for receiving a reply after sending a message */
  public void recSendMessageReply(String msg);

  /* Service for receiving a reply after registering a new member */
  public void recRegisterReply(String msg);

  /* Service for receiving a reply after unregstering the member */
  public void recUnRegisterReply(String msg);

  /* Service for receiving a reply after creating a channel */
  public void recCreateChannelReply(String msg);

  /* Service for receiving a reply after destroying a channel */
  public void recDestroyChannelReply(String msg);

  /* Service for receiving a reply after subscribing to a channel */
  public void recSubscribeChannelReply(String msg);

  /* Service for receiving a reply after unsubscribing to a channel */
  public void recUnsubscribeChannelReply(String msg);

  /* Service for receiving a reply after receiving a message */
  public void recReceiveMessageReply(String msg);

  /* Service for receiving a reply after forwarding a message */
  public void recForwardReply(String msg);

  /* Service for receiving a reply after recovering the member */
  public void recGetMemberReply(String msg);

}
```

Figura 100: Servicios de Extensión – Extensiones Orientadas a Canal

```
package edu.unu.iist.emacao.geeg.extensions;
import java.io.IOException;

import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public interface IChannelExtension {

  public abstract String processMessage(String message) throws IOException,
    ParserConfigurationException, SAXParseException, SAXException;
  public void enableExtension(
    Long extDbId, String extName, Long chDbId, String file);
  public void disableExtension(
    String extName, Long chDbId, boolean update);
  public void configureExtension(
    String chId, String extId, String extName, String data);
  public String requestExtension(
    String chId, String extId, String extName, String data);
}
```