Universidad Nacional del Sur

Tesis de Doctor en
Ciencias de la Computación

Tolerancia a Fallas y Gestión de Carga en Entornos Federados

Javier Echaiz

Bahía Blanca – Argentina
2011

Universidad Nacional del Sur

Tesis de Doctor en
Ciencias de la Computación

Tolerancia a Fallas y Gestión de Carga en Entornos Federados

Javier Echaiz

Bahía Blanca – Argentina
2011

A mi amada esposa, pues sin su soporte éste y otros logros

no hubiesen sido posibles, y a mis hijos que iluminan mis metas.

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, y no ha sido presentada previamente para la obtención de otro título en esta universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Ciencias e Ingeniería de la Computación, durante el período comprendido entre noviembre de 2005 y diciembre de 2010, bajo la dirección del Dr. Guillermo R. Simari, Profesor Titular del Departamento de Ciencias e Ingeniería de la Computación.

Javier Echaiz
jechaiz@cs.uns.edu.ar
Departamento de Ciencias e
Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, 1^o de marzo de 2011

UNIVERSIDAD NACIONAL DEL SUR
Secretaría General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el .../.../..., mereciendo la calificación de ..... (.....).

Agradecimientos

Heaven and earth are scant repayment for help rendered where none was received. A kindness done in the hour of need may itself be small, but in worth it exceeds the whole world.

Thiruvalluvar

Quisiera agradecer a mi director Guillermo R. Simari por su guía, apoyo y voluntad de encontrar siempre tiempo para mí en su ajustada agenda. Él es el responsable de estimular en mi la necesidad de buscar nuevos horizontes a través de la investigación. Mi gratitud se extiende así mismo hacia mi jurado, Armando De Giusti, Marcelo Naiouf y Javier Orozco por sus observaciones y comentarios.

Quiero agradecer también a mis compañeros del *Departamento de Ciencias e Ingeniería de la Computación* por proveer un cálido ambiente de trabajo; en particular a Jorge R. Ardenghi por brindarme no sólo su apoyo académico sino también sus valiosos consejos que me ayudan, aún hoy, a cumplir mis metas.

Mi gratitud también está dirigida a mi amigo, (ex)compañero de oficina, y actual Decano Rafael "Benja" García por darme un lugar en sus cátedras en mis inicios en la docencia y convertir nuestra relación de trabajo en una fuerte amistad. En este punto quiero también agradecer a Pablo Davicino, Alejandro Stankevicius, Marcela Capobianco, Sergio Davicino, Bárbara Camelli y Walter Ornella por estar siempre presentes como amigos invaluables.

Gracias al CONICET, sin cuyo apoyo económico a través de Becas de Investigación el desarrollo de esta Tesis no hubiese sido posible.

Mi agradeciomiento también se extiende a Donald Knuth y a Leslie Lamport por TEX y LATEX respectivamente; y a la Universidad de California, Berkeley por su maravilloso FreeBSD.

Mis últimas palabras de agradecimiento están reservadas a mi familia por su amor, estímulo, apoyo incondicional y fe en mi; ellos son el origen de mi determinación. Quiero agradecer especialmente a mi esposa por todo el amor que me brinda diariamente, su comprensión, paciencia y compañía durante estos años.

Resumen

Existe una creciente demanda de sistemas *online*, especialmente de aquellos que requieren procesamiento de información. Esta demanda sumada a las nuevas tecnologías de monitoreo (como por ejemplo las redes de sensores) impulsaron un nuevo tipo de aplicación que requiere bajas latencias y procesamiento continuo de grandes volúmenes de datos (los cuales arriban en forma de streams). El *procesamiento de streams* constituye un paradigma de cómputo relacionado con SIMD que permite que algunos tipos de aplicaciones puedan explotar una forma de procesamiento paralelo y puede emplearse en diferentes dominios, como por ejemplo para la implementación de sistemas financieros, monitoreo basado en sensores, sistemas militares, monitoreo de red, *etc.* Si bien los sistemas de gestión de bases de datos (DBMS) pueden utilizarse para implementar este tipo de aplicaciones, las restricciones de bajas latencias de procesamiento y grandes volúmenes de datos a procesar los vuelve inadecuados. Una mejor alternativa son los sistemas de gestión de streams de datos, usualmente *sistemas distribuidos de gestión de streams de datos* (DSMS por su sigla en inglés) debido a que estas aplicaciones son inherentemente distribuidas y por lo tanto las soluciones distribuidas son naturales y proveen mejoras en cuanto a escalabilidad y performance.

Esta tesis se enfoca en dos aspectos desafiantes pertenecientes al campo de los sistemas distribuidos en general y al de los DSMS en particular: (1) tolerancia a fallas capaz de resistir fallas a nivel de nodos y de red y (2) gestión de carga en sistemas federados.

Nuestro enfoque al problema de la tolerancia a fallas se basa en replicación capaz de enmascarar tanto las fallas a nivel de los nodos como a nivel de las redes. Nuestro

protocolo, denominado *Disponibilidad y Consistencia Ajustable a las Aplicaciones* (DCAA) puede manejar adecuadamente la relación entre disponibilidad y consistencia, manteniendo (si es posible) la disponibilidad especificada por el usuario o la aplicación, pero produciendo (eventualmente) los resultados correctos. Al mismo tiempo, DCAA también trata de producir el menor número de resultados incorrectos (imprecisos) que luego deberán requerir corrección. La principal diferencia entre DCAA y enfoques previos sobre tolerancia a fallas en el campo de los DSMS es que DCAA soporta *al mismo tiempo* diferentes restricciones en las aplicaciones, esto quiere decir que cada aplicación puede potencialmente tener distintas preferencias de disponibilidad y consistencia.

Por otro lado presentaremos un nuevo protocolo de gestión de carga denominado *Mecanismo de Precio Acotado* (MPA), el cual permite que nodos autónomos (participantes colaborativos) compartan su carga sin la necesidad de contar con recursos suficientes para la operación durante picos de carga. MPA es un protocolo basado en contratos donde cada nodo practica una *negociación offline* y los participantes migran carga en tiempo de ejecución únicamente a nodos (pares) con los cuales mantienen un contrato (y pagan mutuamente de acuerdo al precio contratado). Este protocolo de gestión de carga ofrece incentivos que promueven la participación de los nodos y produce una buena distribución de carga (a nivel global del sistema). Los aportes más importantes de nuestro enfoque por sobre trabajos previos basados en economías de cómputo son su estabilidad, predecibilidad, baja carga de procesamiento, privacidad y promoción de relaciones entre participantes, posibilitando que los mismos pueden crear y explotar estas relaciones privilegiadas. El protocolo MPA es general y por lo tanto puede utilizarse para la gestión de carga de cualquier entorno federado y no sólo bajo DSMS. Más aún, este nuevo protocolo de gestión de carga debe no sólo trabajar en los típicos entornos colaborativos sino que también debe ser capaz de solucionar escenarios más reales, donde cada nodo (probablemente parte de diferentes organizaciones autónomas) juega bajo distintas

reglas, tratando de maximizar su propia ganancia sin cooperar necesariamente con sus pares. Además de los modelos económicos existen varios trabajos basados en SLA (*Service Level Agreements*) para solucionar el problema de la gestión de carga cuando el entorno no es colaborativo. Mostraremos que los modelos SLA no proveen una solución completa y que los *acuerdos entre pares* usualmente proveen mejores resultados.

Si bien esta tesis parece tener dos focos en lugar de uno, es importante notar que atacaremos especialmente el problema de la gestión de carga en sistemas distribuidos federados. La relación entre este enfoque y la tolerancia a fallas radica en los contratos negociados: además de precio y tareas (carga), los contratos pueden incluir disponibilidad, característica que vuelve especialmente importante la tolerancia a fallas.

Términos clave: sistemas distribuidos, sistemas federados, gestión de carga, tolerancia a fallas, DSMS.

Director de tesis: Dr. Guillermo R. Simari.
Cargo: Profesor Titular del Dpto. de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur.

Abstract

There is an increased demand for online systems, especially those requiring information processing. This demand added to new monitoring technologies (like sensors networks) have motivated a new type of application that requires low latency and continuous processing of large volumes of data (arriving as streams). *Stream processing* is a computer programming paradigm, related to SIMD, that allows some applications to more easily exploit a form of parallel processing and it can be employed in many different domains, such as financial systems, sensor based monitoring, military systems, network monitoring, *etc.* Even when traditional database management systems (DBMS) can be used to handle these applications, the low latency and high volume processing constrains make them not suitable. A much better alternative are the data stream management systems, usually *distributed data stream management systems* (DSMS) because these are inherently distributed applications so distributed solutions are natural and providers of scalability and performance improvements.

This thesis focuses on two challenges faced by distributed systems in general and DSMS in particular: (1) fault tolerance able to resist node and network failures and (2) load management in federated systems.

The fault tolerance approach is based on replication and our protocol can resist most node and network failures. It is called *Disponibilidad y Consistencia Ajustable a las Aplicaciones* (DCAA) and addresses the availability/consistency relation by maintaining (if possible), the availability specified by the user or the application but (eventually) producing correct results. At the same time, DCAA also attempts to produce the minimum number of incorrect (inaccurate) results that will need

correction. The main difference of DCAA over previous approaches for fault tolerance in DSMS is that DCAA supports *at the same time* different application constrains, this means that each application can potentially choose a different preference of availability and consistency.

Our load management protocol, called *Mecanismo de Precio Acotado* (MPA) enable autonomous nodes (collaborative participants) to share their load without having the required resources for peak load work. MPA is a contract based protocol where nodes practice an *offline negotiation* and participants migrate load at execution time only to peers with whom they hold a contract (and pay each other according to the contracted price). This load management protocol offers incentives that promote participation and produces a good (system wide level) load distribution. The key differences of our approach over previous works based on computational economies are its stability, predictability, lightweight, privacy, and promotion of the relationships among participants, enabling them to create and exploit these privileged relationships. The MPA protocol is general, so it can be used to manage load in any federated environment, and not only DSMS. Moreover, this new load management protocol should not only work under the typical collaborative environment, but also should be able to address the more realistic scenery where each node (probably part of different and autonomous organizations) plays under different rules trying to maximize their own gain, without necessarily cooperating with their partners. Besides economic models there are various works based on SLA (service level agreements) to solve the load management problem when the environment is not a collaborative one. We will show that SLA models do not provide a complete solution and that *peer agreements* usually provide better results.

Although this thesis seems to have two focuses instead of one, it is important to notice that we especially address the load management problem under federated distributed systems. The relation among this focus and fault tolerance is in the negotiated contracts: besides price and tasks (load), contracts can include availability,

which raises the importance of fault tolerance.

KEYWORDS: distributed systems, federated systems, load sharing, fault tolerance, DSMS.

THESIS SUPERVISOR: Dr. Guillermo R. Simari.

POSITION: Full Professor Dept. of Computer Science and Engineering, Universidad Nacional del Sur.

Índice General

Prefacio v
Agradecimientos vii
Resumen ix
Abstract xi
Acrónimos xxiii
Terminología xxv

CAPÍTULOS

I. INTRODUCCIÓN 2
1.1. Aplicaciones de procesamiento de streams 4
1.2. Sistemas de gestión de streams de datos 6
1.2.1. Streams de datos 6
1.2.2. Operación Distribuida 8
1.3. Desafíos y Contribuciones 9
1.3.1. Desafíos en tolerancia a fallas 9
1.3.2. Contribuciones en el área de tolerancia a fallas 13
1.3.3. Desafíos en gestión de carga 15
1.3.4. Contribuciones en gestión de carga 17
1.4. Organización de la tesis 19
1.5. Resumen 19

II. ANTECEDENTES Y TRABAJOS RELACIONADOS 20
2.1. Los DBMSs tradicionales y sus extensiones 21
2.1.1. Extensiones a los DBMSs 22
2.1.2. Tolerancia a fallas en los DBMSs tradicionales 25
2.2. Data stream management systems (DSMS) 27

2.2.1. Investigación en el área de los DSMS . . . 28
2.2.2. Tolerancia a fallas en los DSMS . . . 31
2.3. Sistemas workflow . . . 32
2.4. Sistemas de Publicación/Subscripción . . . 33
2.5. Redes de sensores . . . 34
2.6. Replicación de máquinas de estados . . . 35
2.7. Recuperación mediante rollback . . . 36
2.8. Gestión de carga . . . 36
2.8.1. Algoritmos de gestión de carga cooperativos . . . 36
2.8.2. Diseño de mecanismo algorítmico distribuido . . . 38
2.8.3. Gestión de carga basada en modelos económicos . . . 39
2.8.4. Recursos compartidos en sistemas peertopeer . . . 40
2.8.5. Service Level Agreements . . . 40
2.9. Resumen . . . 40
III. ARQUITECTURA . . . 42
3.1. Modelo de streams de datos . . . 43
3.2. Operadores . . . 45
3.2.1. Operadores sin estado . . . 45
3.2.2. Operadores con estado . . . 47
3.2.3. Operadores persistentes . . . 50
3.2.4. Diagramas de consulta . . . 50
3.3. Arquitectura del sistema . . . 51
3.4. Arquitectura del nodo . . . 52
3.5. Data flow . . . 54
3.6. Resumen . . . 55
IV. TOLERANCIA A FALLAS . . . 56
4.1. Definición de Falta, Error, Falla . . . 57
4.2. Clasificación de Fallas . . . 58
4.3. DCAA . . . 60

4.4. Definición del problema 61
4.4.1. Objetivos de diseño 64
4.4.2. Modelo de fallas y asunciones 68
4.4.3. Clasificación de operadores 71
4.5. Generalidades sobre DCAA 74
4.6. Arquitectura de software extendida 76
4.7. Modelo de datos mejorado 78
4.8. Estado ESTABLE 79
4.8.1. Serialización de las tuplas de entrada 80
4.8.2. Impacto de la selección del valor *stime* 83
4.8.3. Detección de fallas 84
4.9. Estado FALLA_UPSTREAM 87
4.9.1. Cambio de vecinos upstream 87
4.9.2. Gestión de la disponibilidad y la consistencia en presencia de fallas 89
4.10. Estado ESTABILIZACIÓN 91
4.10.1. Reconciliación del estado de un nodo 91
4.10.2. Estabilización de los streams de salida 92
4.10.3. Procesamiento de nuevas tuplas durante la reconciliación . 94
4.10.4. Correcciones en background de tuplas de entrada 95
4.11. Recuperación del nodo fallado 97
4.12. Gestión del buffer 98
4.12.1. Requerimientos de buffering 99
4.12.2. Evitando el buffering de tuplas tentativas 99
4.12.3. Algoritmo básico de gestión de buffer 101
4.12.4. Gestión de las fallas prolongadas 102
4.13. Aplicaciones cliente y los orígenes de datos 104
4.13.1. Proxies 104
4.13.2. Proxies en el origen de los datos 105
4.14. Propiedades de DCAA 106

4.14.1. Propiedades de disponibilidad y consistencia 107
4.14.2. Propiedades de las fallas múltiples 112
4.15. Resumen 117
V. GESTIÓN DE CARGA 122
5.1. Definición del problema 125
5.1.1. Tareas y carga 125
5.1.2. Utilidad 126
5.1.3. Correspondencia de elección social 129
5.2. Descripción de MPA 130
5.3. Migraciones en tiempo de ejecución 133
5.4. Condiciones de carga dinámica 137
5.5. Contratos offline de precio fijo 140
5.6. Contratos de precio acotado 147
5.6.1. Rango de precio mínimo 148
5.6.2. Negociación del precio final 152
5.7. Aplicación de MPA a sistemas DSMS federados 158
5.8. Propiedades de MPA 158
5.8.1. Asunciones adicionales 159
5.8.2. Propiedades 160
5.9. Resumen 172
VI. EVALUACIÓN 174
6.1. Implementación del simulador 175
6.1.1. Topologías simuladas 176
6.2. Convergencia hacia asignaciones aceptables 178
6.3. Velocidad de convergencia 181
6.4. Estabilidad bajo variaciones de carga 183
6.5. Implementación 185
6.5.1. Experimentos con el prototipo 185
6.5.2. Limitaciones y extensiones 188

6.6. Resumen 189

VII. CONCLUSIONES Y RESULTADOS OBTENIDOS 192

7.1. Procesamiento de streams tolerante a fallas 192

7.2. Gestión de carga en sistemas federados 195

7.3. Trabajos Futuros 196

APÉNDICES

BIBLIOGRAFÍA 199

ÍNDICE TEMÁTICO 225

Índice de Tablas

2.1. Extensiones a los DBMS y sus similitudes y diferencias con los DSMS 25

2.2. Principales diferencias entre los DBMSs y los DSMS 28

2.3. Consultas contínuas y DSMS 28

4.1. Nuevos tipos de tuplas 79

4.2. Algoritmo para cambio de réplica de un vecino upstream para mantener disponibilidad 89

4.3. Algoritmo para cambio de réplica de un vecino upstream para mantener disponibilidad mientras se corrige entrada en bg 119

4.4. Resumen de las propiedades del protocolo de disponibilidad y consistencia 120

5.1. Heurísticas para establecer contratos offline 147

5.2. Ejemplo de contraofertas (y los correspondientes sobreprecios expresados como porcentaje sobre el rango del precio total) de vendedores con diferentes valuaciones para recursos y diferente número de competidores 156

5.3. Resumen de las propiedades del mecanismo de precio acotado 159

Índice de Figuras

1.1. Vista de alto nivel del procesamiento de streams 5

1.2. Ejemplo de streams y esquemas para una aplicación de monitoreo de red 7

1.3. Ejemplo de un diagrama de consulta para la aplicación de monitoreo de red 8

1.4. Ejemplo de procesamiento de streams distribuido 9

1.5. Tipos de fallas y los objetivos de tolerancia a fallas 13

1.6. Diferentes tipos de asignaciones de carga en un sistema de dos nodos 15

3.1. Ejemplo de salidas de operadores sin estado 46

3.2. Ejemplo de salida de un operador Aggregate 48

3.3. Ejemplo de salida de un operador Join 50

3.4. Ejemplo de un diagrama de consulta para la aplicación de monitoreo de red 51

3.5. Arquitectura de software de un nodo Federación 53

3.6. Data flow en un sistema Federación 54

4.1. Diagrama de consulta distribuido y replicado 60

4.2. Fallas que originan tuplas tentativas 61

4.3. Diagrama de consulta compuesto por operadores bloqueantes (Join) y no bloqueantes (Union) 71

4.4. Taxonomía de operadores de procesamiento de streams 72

4.5. Autómata finito de DCAA 75

4.6. Extensiones de software a la arquitecura DSMS para soportar DCAA 77

4.7. Ejemplo del uso de tuplas TENTATIVAs y de DESHACER 79

4.8. Los límites permiten ordenar las tuplas determinísticamente 81

4.9. Ejemplo de tuplas organizadas mediante buckets para intervalo límite $d = 5$ 82

4.10. Ubicación de `SUnion` en un diagrama de consulta 83

4.11. Ejemplo de un DSMS distribuido y replicado 88
4.12. Protocolo de comunicación inter-réplica y el estado de ESTABILIZACIÓN 95
4.13. Ejemplo de uso de buffers de salida, sin falla de nodo 100
4.14. Ejemplo de uso de buffers de salida, con falla de nodo 101
4.15. Ubicaciones donde las tuplas se bufferean con checkpoint/redo 101
4.16. Ubicaciones donde las tuplas se bufferean con checkpoint/redo cuando todas las tuplas tentativas son corregidas 102
4.17. Ejemplo de asignación de tamaño de buffer con operadores capaces de converger 104
4.18. Proxies cliente y origen de datos 105
4.19. Paths en un diagrama de consulta 106
4.20. Tipos de árboles posibles en un diagrama de consulta distribuido y replicado 107
4.21. Cambio entre réplicas de vecinos upstream durante diferentes estados de consistencia 114
5.1. Necesidad de reasignación de carga en un DSMS de cuatro participantes 126
5.2. Reasignación de carga en un DSMS de cuatro participantes 127
5.3. Precios y costos de procesamiento 128
5.4. Decisiones de movimiento de carga basadas en los costos marginales . 134
5.5. Tres escenarios de movimiento de carga para dos partners 137
5.6. Distribuciones de carga limitadas con $k = 0{,}01$, $T = 1{,}0$ y diferentes valores de *alpha* 142
5.7. Probabilidad de sobrecarga frente a un pico de carga 143
5.8. Magnitud de sobrecarga esperada cuando ocurre un pico de carga . . 144
5.9. Ejemplo de costo y beneficio producto de un aumento en el número de contratos de precio fijo 145
5.10. Beneficios para el comprador por contratos por debajo del umbral T . 146
5.11. Contratos de precio fijo no siempre producen asignaciones aceptables 148
5.12. Ejemplo del cómputo de $\delta_k(conj\_tareas)$ con $k = 3$ 150
5.13. Movimientos de carga entre tres nodos empleando un rango pequeño de precios 151
5.14. Ejemplos de mejor y peor caso de distribuciones de carga 170

6.1. Convergencia hacia asignaciones aceptables para diferentes cargas y número de contratos 179
6.2. Digrama de consulta de monitoreo de red 186
6.3. Ambiente experimental para monitoreo de red 186
6.4. Carga en tres nodos Federación corriendo consulta de monitoreo de red 187
6.5. Movimientos de carga para la aplicación de monitoreo de red 187
6.6. Contrato nuevo Nodo 3 188

Acrónimos

ACM	Association for Computing Machinery.
AFS	Andrew File System.
API	Application Program Interface.
CLEX	Cluster Level EXecution.
COMA	Cache Only Memory Access.
CPU	Central Processing Unit.
DSM	Distributed Shared Memory.
EGID	Effective Group Id.
EUID	Effective User Id.
GID	Group Id.
GNU	acrónimo recursivo, "GNU's Not Unix".
GUI	Graphical User Interface.
HA	High Availability.
HP	High Performance.
HW	Hardware.
ID	Identifier.
IDL	Interface Definition Language.
IEEE	Institute of Electrical & Electronics Engineers.
I/O	Input/Output.
IP	Internet Protocol.
IPC	Inter Process Communication.
LAN	Local Area Network.
MIPS	Million Instructions Per Second.
MP	Migración de Procesos.
NFS	Network File System.

NIC Network Iterface Card, placa de red.

NID Node Id.

NIS Network Information Service.

NOW Network of Workstations.

NUMA Non Uniform Memory Access.

PC Personal Computer.

PCB Process Control Block.

PDA Personal Digital Assistant.

PID Process Id.

POSIX Portable Operating System Interface.

PPID Parent Process Id.

RAID Redundant Array of Independent (or Inexpensive) Disks.

RAM Random Access Memory.

RFC Request For Comments.

ROM Read-Only Memory.

RPC Remote Procedure Call.

SFS The Self-Certifying File System.

SID Session Id.

SMP Symmetric multiprocessing.

S.O. Sistema Operativo.

SPMD Single Program / Multiple Data.

SSI Single System Image.

SW Software.

TCP Transport Control Protocol.

UDP User Datagram Protocol.

UID User Id.

VGID Virtual Group Id.

VPID Virtual Process Id.

WAN Wide Area Network.

WWW World Wide Web.

YP Yellow Pages.

Terminología

Este glosario contiene términos en inglés (referentes al área de Sistemas) y la acepción de los mismos en castellano utilizada en esta Tesis. Para cada uno de ellos se indica la página en donde aparece por primera vez.

bottleneck cuello de botella.

broadcast mensaje que se envía a todos los nodos de la red.

callback devolución de llamada.

CPU-bound ocurre cuando el procesador no puede ejecutar con suficiente velocidad como para mantener el número de procesos en la cola *run* consistentemente bajo.

daemon demonio, proceso servidor que no posee una terminal controladora.

default omisión.

diskless máquina sin unidades de disco rígido.

drop acción de descartar un mensaje recibido.

end point una aplicación, servicio, protocolo u otro agente computacional que emplea la red para transferir datos hacias otro *end point*.

ethernet norma o estándar (IEEE 802.3) que determina la forma en que los puestos de la red envían y reciben datos sobre un medio físico compartido que se comporta como un bus lógico, independientemente de su configuración física. Originalmente fue diseñada para enviar datos a 10 Mbps, aunque posteriormente ha sido perfeccionado para trabajar a 100 Mbps, 1 Gbps o 10 Gbps y se habla de versiones futuras de 40 Gbps y 100 Gbps.

exit code valor numérico retornado por el proceso que finaliza su ejecución hacia su proceso padre que generalmente se emplea para indicar si la operación fue exitosa (valor 0) o no (valor mayor que 0). Este valor puede obtenerse desde un shell UNIX mediante la variable especial $?

flag bandera, marca.

garbage collector programa de recolección de basura, se devuelven al sistema recursos asignados que no están siendo utilizados actualmente.

grid computing infraestructura distribuida geográficamente para la ejecución de aplicaciones científicas y de ingeniería. El término ha ido ganando popularidad, a la vez que se extiende abarcando desde el networking hasta la inteligencia artificial.

handler rutina de gestión de una señal, una interrupción, o un dispositivo.

header encabezado.

home nodo origen, *i.e.,* desde donde fue ejecutado inicialmente un proceso.

job tarea.

library librería.

load balancing balance de carga.

load sharing carga compartida.

mainframe los ordenadores centrales o mainframes son ordenadores grandes, potentes y caros usados principalmente por grandes compañías para el procesamiento de grandes cantidades de datos, por ejemplo, el procesamiento de transacciones bancarias. El término apareció a principios de los setenta con la introducción de ordenadores más pequeños como la serie DEC PDP, que fueron conocidos como miniordenadores, por lo que los usuarios acuñaron el término ordenador central para describir a los tipos de ordenadores más grandes y antiguos.

master maestro.

multicast mensaje que se envía a todos los nodos de un determinado grupo.

nonpreemptive no apropiativo.

offset desplazamiento.

overhead sobrecarga.

pathname cadena (*string*) con el camino completo al recurso, usualmente un archivo.

pipe tipo especial de archivo que implementa semántica FIFO para procesos lectores y escritores.

port pórtico.

preemptive apropiativo.

process id identificador de proceso, PID.

remote execution ejecución remota.

scheduling planificación.

server servidor.

shell intérprete de comandos, *e.g.,* `zsh`, `bash`, `csh`.

signal señal, uno de los mecanismos de IPC de un sistema UNIX. Originalmente creado para matar procesos.

socket punto final de un enlace de comunicación de dos vías entre dos procesos a través de la red.

source origen o fuente.

speedup aumento en velocidad.

system call llamada a sistema.

target destino.

thread hilo.

timeout pausa, intervalo de suspensión momentánea de actividad, pág. 59.

timer crónometro.

timesharing tiempo compartido.

unicast mensaje que se envía a un nodo particular.

worker trabajador, también llamado esclavo.

workstation estación de trabajo, nodo, máquina, computadora.

zombie proceso que no fue enterredo por su padre.

Capítulo 1

Introducción

Índice

1.1. Aplicaciones de procesamiento de streams 4
1.2. Sistemas de gestión de streams de datos 6
1.2.1. Streams de datos 6
1.2.2. Operación Distribuida 8
1.3. Desafíos y Contribuciones 9
1.3.1. Desafíos en tolerancia a fallas 9
1.3.2. Contribuciones en el área de tolerancia a fallas . . . 13
1.3.3. Desafíos en gestión de carga 15
1.3.4. Contribuciones en gestión de carga 17
1.4. Organización de la tesis 19
1.5. Resumen 19

Since the early days of mankind the primary motivation for the establishment of communities has been the idea that being part of an organized group the capabilities of an individual are improved. The great progress in the area of intercomputer communication led to the development of means by which stand-alone processing subsystems can be integrated into multicomputer communities.

— *Miron Livny*

En los últimos años ha emergido una nueva clase de aplicaciones intensivas sobre los datos (*data-intensive*). Estas aplicaciones, usualmente conocidas como *stream processing applications*, suelen requerir procesamiento

continuo y de baja latencia de grandes volúmenes de información provenientes de diversas fuentes de datos y a alta velocidad. Las aplicaciones de streaming provienen de diferentes dominios, motivadas por diferentes necesidades.

Los avances en miniaturización (electrónica) y las redes wireless posibilitaron el desarrollo de dispositivos capaces de sensar el mundo físico y comunicar información acerca de este mundo. Ejemplos de estos dispositivos incluyen una gran variedad de sensores de entorno [Tec11], etiquetas miniatura para seguimiento de objetos [KLN09], dispositivos de sensado de ubicación [HHS+02, BGR+90, PMBT01], *etc.* El despliegue de estos dispositivos permite aplicaciones tales como las de monitoreo de entornos mediante sensores, (por ejemplo monitores de temperatura de edificios, monitoreo de calidad de aire), aplicaciones de ingeniería civil, (*e.g.,* monitoreo de autopistas, monitoreo de estado de cañerías), equipo de rastreo basado RFID, aplicaciones militares (*e.g., platoon tracking*, detección de objetivos), y aplicaciones médicas (*e.g.,* monitoreo de pacientes basado en sensores). Todas estas aplicaciones deben procesar constantemente streams de información provenientes de los dispositivos desplegados.

En algunas áreas, como en las redes de computadoras (*e.g.,* detección de intrusos, monitoreo de redes, rastreo de propagación de worms, logs de web o monitoreo de streams de clicks, servicios financieros), las aplicaciones tradicionalmente procesan un gran volumen de streams de datos. Sin embargo, estas aplicaciones almacenan datos persistentemente para luego procesarlos offline [ESV03], lo que introduce una demora significativa entre el tiempo en el cual ocurren los eventos y cuando son analizados. Alternativamente las aplicaciones pueden procesar los datos online empleando software especializado [Roe99], pero esta solución tiene generalmente asociado un alto costo en la implementación. Los sistemas tradicionales de gestión de bases de datos (DBMSs) basados en el modelo "almacenar y luego procesar" no son adecuados para el procesamiento de alta velocidad y baja latencia asociada a los streams [Ste97, BBD+02, CCD+03, hHBR+03, HXcZ07, CZ09].

Como resultado de ello varias nuevas arquitecturas fueron propuestas. Los nuevos motores son conocidos como *Data Stream Management Systems* (DSMS) [Ste97, MWA+03], o *Continuous Query Processors* (CQP) [CCD+03]. Su objetivo es ofrecer servicios de gestión de datos que cubra las necesidades de todas las aplicaciones arriba mencionadas en un único framework. Las aplicaciones de procesamiento de streams son inherentemente distribuidas, y dado que la distribución puede mejorar la performance y escalabilidad del motor de procesamiento, varios trabajos proponen e implementan DSMS distribuidos [CCD+03, CBB+03].

Al trabajar con DSMS distribuidos surgen varios desafíos. En esta tesis nos enfocaremos en dos desafíos en particular: tolerancia a fallas en un DSMS distribuido y gestión de carga en entornos federados. El objetivo de nuestro mecanismo de tolerancia a fallas es posibilitar que un DSMS distribuido sobreviva a fallas en nodos, red, y particiones de red. El objetivo de nuestro sistema de gestión de carga es crear incentivos y un mecanismo para que los participantes (autónomos) tiendan a colaborar con su carga sin tener individualmente los recursos necesarios y maximizando su productividad. Nuestro mecanismo de gestión de carga encuentra su motivación en el procesamiento de streams, sin embargo es aplicable a otros sistemas federados.

En el resto de este capítulo se describen las principales propiedades de las aplicaciones de procesamiento de streams y las principales características de los motores de procesamiento de streams. Además se introducen los problemas de tolerancia a fallas y gestión de carga que se abordan en esta tesis. Luego se delinean las contribuciones más importantes y se discuten algunos de los hallazgos más importantes de esta investigación.

1.1 Aplicaciones de procesamiento de streams

Las aplicaciones de procesamiento de streams difieren significativamente de las aplicaciones tradicionales de gestión de datos. Estas últimas típicamente operan en conjuntos de datos acotados ejecutando consultas sobre los datos persistentemente almacenados. Un ejemplo típico es un negocio que necesita una aplicación para llevar su inventario, ventas, equipamiento, y empleados. Para este tipo de aplicaciones contamos con DBMSs como mysql, postgres, Oracle, Informix, Microsoft SQL Server, donde los datos son inicialmente ordenados e indexados para luego ser procesados mediante consultas (*queries*).

En contraste, como puede apreciarse en la Figura 1.1, en una aplicación de procesamiento de streams, las fuentes de datos producen streams de información de tamaño ilimitado y las aplicaciones ejecutan consultas continuas sobre estos streams. El monitoreo de redes es un ejemplo de una aplicación de procesamiento de streams, en ella las fuentes de datos son los monitores de red que producen información acerca del origen o destino del tráfico hacia las máquinas pertenecientes a las subredes monitoreadas. Uno de los posibles objetivos de esta aplicación es computar estadísticas continuamente sobre estos streams, de forma tal de posibilitar a un administrador de red que observe el estado de la misma, detectando anomalías, como por ejemplo intentos de intrusión.

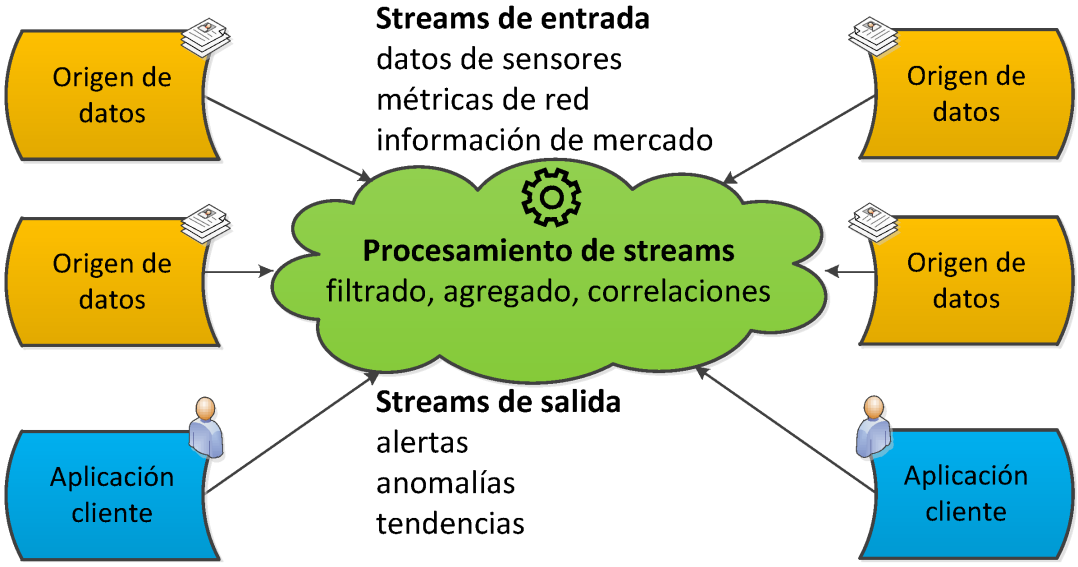


Figura 1.1: Vista de alto nivel del procesamiento de streams

Las aplicaciones de procesamiento de streams presentan las siguientes propiedades [ABB+04, BBD+02, BBC+04a, CCD+03]:

1. **Modelo continuo de procesamiento de consultas:** En un DBMS tradicional, los clientes ejecutan consultas (*one time*) sobre la información almacenada (*e.g.,* "Existe alguna dirección fuente que haya intentado más de 100 conexiones en un período de un minuto?"). En una aplicación de procesamiento de streams, los clientes envían consultas de monitoreo de larga duración que deben ser procesadas continuamente sobre los nuevos datos que van arribando (*e.g.,* "Alérteme si desde una dirección fuente provienen más de 100 conexiones en un período de un minuto?"). Los clientes que envían consultas continuas esperan resultados periódicos o alertas cuando ocurren ciertas entradas.

2. **Modelo de procesamiento *push*:** En una aplicación de procesamiento de streams, una o más fuentes de datos continuamente producen información y la envían (*push*) al sistema para su procesamiento. Las aplicaciones cliente pasivamente esperan que el sistema les envíe resultados periódicos o alertas. Este modelo de procesamiento contrasta con el modelo tradicional, donde los DBMSs procesan localmente la información almacenada, y los clientes activamente traen (*pull*) información sobre los datos cuando la necesitan.

3. **Procesamiento de baja latencia:** muchas aplicaciones de procesamiento de streams monitorean fenómenos en tiempo real y por lo tanto requieren procesamiento de baja latencia sobre estos datos de entrada. Por ejemplo,

para una aplicación de monitoreo de red, información acerca de los ataques de intrusión que están sucediendo en estos momentos es más importantes que información acerca de ataques que sucedieron en el pasado. Los DSMS tratan de proveer baja latencia de procesamiento pero sin garantías duras.

4. **Frecuencia de datos de entrada alta y variable:** en muchas aplicaciones de procesamiento de streams las fuentes de datos producen grandes volúmenes de información. Además puede variar en gran medida la frecuencia de los datos. Por ejemplo, un ataque *Denial of Service* (DoS) puede causar un número elevado de conexiones. Si los monitores de red producen un dato por cada conexión entonces se generarán altas frecuencias de datos sobre los streams durante el ataque. Dado que las frecuencias de datos varían, la carga en un DSMS también varía debido a que fluctúa la cantidad de datos que se tienen que procesar durante las consultas.

1.2 Sistemas de gestión de streams de datos

Los *data stream management systems* (DSMS) introducen nuevos modelos de datos, operadores y lenguajes de consulta. Su arquitectura interna también difiere de la de los DBMSs tradicionales. Mediante consultas continuas procesan los datos según van arribando.

1.2.1 Streams de datos

Existen diversos modelos para el procesamiento de streams [ABW06, ABW02, CBB+03, LWZ04, TMSF03]. Estos modelos se basan en la idea central de que un stream en una secuencia de items de datos tipo *append-only*. Los items de datos están compuestos por atributos, llamados tuplas. Todas las tuplas del mismo stream tienen el mismo conjunto de atributos, los cuales definen el tipo de esquema de un stream. Típicamente cada stream es producido por una única fuente de datos.

La Figura 1.2 muestra un ejemplo de streams, tuplas y esquemas para una aplicación de monitoreo de red. En este ejemplo los monitores de red son la fuente de datos. Ellos producen los streams de entrada, donde cada tupla representa una conexión y tiene el siguiente esquema: `tiempo` en el que la conexión fue establecida, `dirección fuente`, `dirección destino`, y `port destino`. El stream destino presenta un esquema diferente, cada tupla indica cuantas conexiones fueron establecidas por cada dirección fuente en cada período de tiempo predefinido.

En aplicaciones de procesamiento de streams, los streams de datos son filtrados,

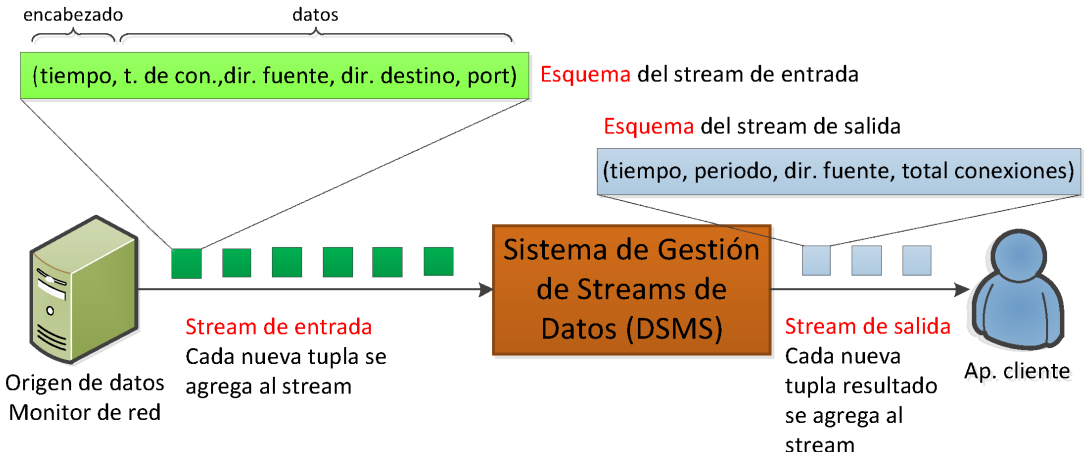


Figura 1.2: Ejemplo de streams y esquemas para una aplicación de monitoreo de red

correlacionados y agregados por operadores para producir salida de interés. Un operador puede verse como una función que transforma uno o más streams de entrada en uno o más streams de salida. Dado que los DSMS están inspirados en los DBMSs tradicionales, soportan operadores análogos a los operadores relacionales como `Select`, `Join`, `Project`, `Union`, y `Aggregate` [ABW06, CCD+03, SPAM91, SL90]. Si bien existen motores capaces de soportar operadores definidos por el usuario, en general la logística de la mayoría de las aplicaciones hace que pueden implementarse directamente empleando las funciones predefinidas [BBC+04a].

Debido a que los streams no tienen tamaño acotado y que las aplicaciones requieren que la salida arribe en el momento esperado, los operadores no pueden acumular estado que crezca con el tamaño de las entradas y no pueden esperar a ver todas las entradas antes de producir un valor. Por esta razón, los operadores de procesamiento de streams efectúan sus cómputos sobre ventanas de datos que se mueven con el tiempo (ventanas deslizantes). Estas ventanas se definen asumiendo que las tuplas en un stream son ordenadas según su estampilla de tiempo, por uno de sus atributos, o insertando tuplas de *puntuación* explícitas que especifican el final de un conjunto de datos [CGM10, TMS03, TMSF03]. Las especificaciones de ventanas hacen que los operadores sean dependientes del orden de las tuplas de entrada.

En algunos sistemas las aplicaciones determinan cómo los streams de entrada deben procesarse a partir de los operadores predefinidos o los definidos por el usuario mediante los cuales se compone un grafo dirigido, libre de bucles y tipo *workflow* denominado *diagrama de consulta*. Otros sistemas directamente usan lenguajes de consulta declarativos del estilo de SQL [ABW06, CCD+03, CJS03, LWZ04] y traducen estas consultas declarativas en diagramas de consulta. La Figura 1.3 ilustra un diagrama de consulta simple.

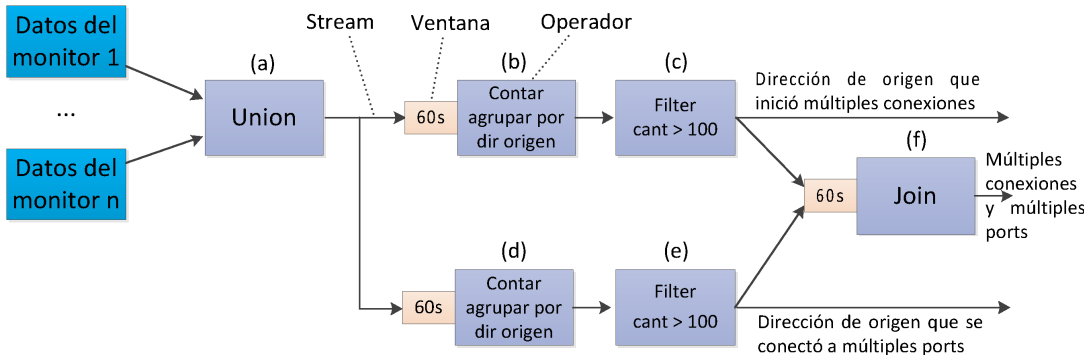


FIGURA 1.3: Ejemplo de un diagrama de consulta para la aplicación de monitoreo de red

La consulta está inspirada en el tipo de procesamiento realizado por herramientas como Snort [Roe99] y Autofocus [ESV03] para monitorear conexiones de red. Cada tupla en los streams de entrada resume una conexión de red: direcciones fuente y destino, tiempo de conexión, protocolo empleado, *etc.* Primero los streams de todos los monitores de red se unen (operador de unión) en un único stream. Luego la consulta transforma ese stream para identificar fuentes que están activas o que tratan de conectarse a distintos *ports* en un período pequeño de tiempo, o ambos. Para contar el número de conexiones y de ports, la consulta aplica operadores de agregación y ventanas: estos operadores bufferean la información de las conexiones por un período T de tiempo. En este ejemplo T es de 60 segundos. Luego los operadores agrupan la información por dirección fuente y aplican la operación de agregación deseada. Los valores agregados son entonces filtrados para identificar el tipo de conexión deseada. En este ejemplo una dirección fuente es etiquetada como activa si establece más de 100 conexiones en un período de 60 segundos o si se conecta a más de 10 ports diferentes. Finalmente se ejecuta un `join` sobre las direcciones fuentes activas que tienen conexiones a muchos ports para identificar fuentes que perteneces a ambas categorías.

Algunos sistemas permiten consultas sobre streams y al mismo tiempo sobre relaciones almacenadas [ABW02, GÖ05, TGNO92]. En este trabajo se restringe el procesamiento a streams de datos *append-only*. Sin embargo, adicionalmente permitiremos operadores de lectura y escritura que realicen un comando SQL de actualización para cada tupla de entrada que reciba y pueda producir streams de tuplas como salida.

1.2.2 Operación Distribuida

Los DSMS son un ejemplo claro de un sistema naturalmente distribuido porque las fuentes de datos se encuentran usualmente distribuidas geográficamente y pertenecen a diversas entidades administrativas. Adicionalmente la distribución puede mejorar la performance y la escalabilidad de un procesador de streams, y posibilitar alta disponibilidad gracias a que los nodos de procesamiento pueden monitorear a los demás y tomar su trabajo cuando se detecta una falla [CCD+03, CDTW00, MSHR02, SDBL07, YG09].

Un DSMS distribuido está compuesto por múltiples máquinas físicas. Cada máquina, es también denominada nodo de procesamiento o simplemente nodo y corre un DSMS. Cada nodo procesa streams de entrada y produce streams de salida que son enviados a las aplicaciones o hacia otros nodos para continuar su procesamiento. Cuando un stream va de un nodo hacia otro, los dos nodos se denominan vecinos *upstream* y *downstream* respectivamente.

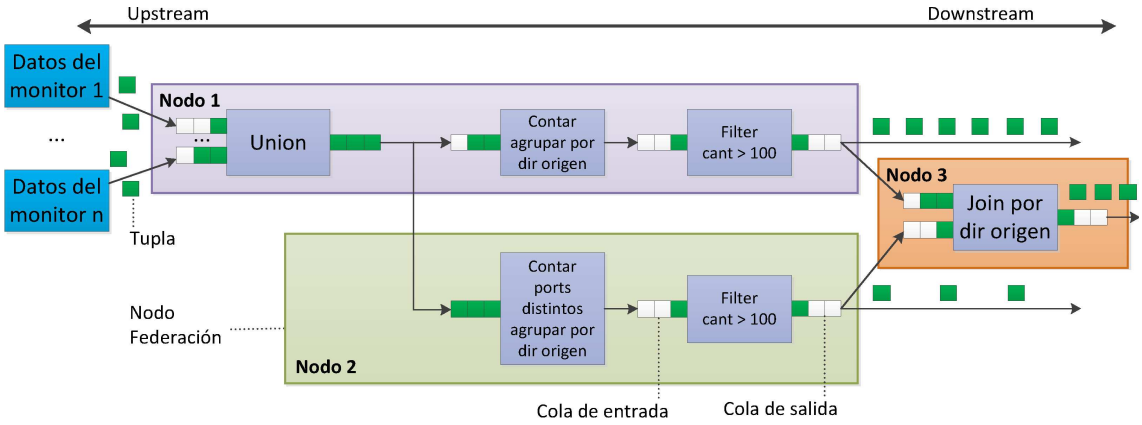


Figura 1.4: Ejemplo de procesamiento de streams distribuido

La Figura 1.4 ilustra un posible despliegue para la consulta de la Figura 1.3. Las fuentes de datos se encuentran distribuidas y remotas, donde se están ejecutando los monitores de red. Todos los streams de entrada se unen en el Nodo 1 antes de ser procesados por los operadores que corren en los Nodos 1, 2 y 3. Cada operador recibe tuplas de entrada de sus colas de entrada. Cuando los streams cruzan los límites del nodo, las tuplas de salida son temporalmente *buffereadas* en colas de salida.

1.3 Desafíos y Contribuciones

Existen varios desafíos relacionados con la construcción un sistema distribuido (posiblemente federado) de procesamiento de streams. En esta tesis atenderemos los

siguientes dos problemas: la tolerancia a fallas en un sistema distribuidos de procesamiento de streams y la gestión de carga en un entorno federado. A continuación plantearemos nuestro enfoque, desafíos y contribuciones en estas dos áreas.

1.3.1 Desafíos en tolerancia a fallas

En un sistema distribuido de procesamiento de streams pueden ocurrir diversos tipos de falla:

- Los nodos de procesamiento pueden fallar y detenerse.
- La comunicación entre los nodos puede interrumpirse.
- El sistema puede particionarse (*split*).

Los tres tipos de fallas pueden ocasionar problemas en el procesamiento de streams: pueden afectar la correctitud de los resultados e incluso pueden interrumpir la producción de los mismos. Para el caso de los nodos nos enfocaremos en fallas del tipo *crash* en lugar de las fallas *Bizantinas*, las cuales causan que los nodos produzcan resultados erróneos.

Idealmente, aun considerando que pueden ocurrir fallas, es de nuestro interés que las aplicaciones cliente reciban los resultados correctos. Para lograr esta propiedad, el sistema replica cada nodo de procesamiento y asegura que las réplicas se mantengan mutuamente consistentes, *i.e.,* que procesen sus entradas en el mismo orden, que progresen a un ritmo similar, que su estado interno de cómputo sea el mismo, y que produzcan las mismas salidas en el mismo orden. Si una falla provoca que un nodo pierda uno de sus streams de entrada, entonces el nodo puede cambiar a una réplica de este nodo que falló o se desconectó y seguir procesando. El estado de las réplicas y la salida "vista" por los clientes coincide con el estado y salida que se hubiese logrado contando con un único nodo de procesamiento y sin fallas.

El desafío está en que algunos tipos de fallas, como las particiones en las redes, pueden causar que un nodo pierda acceso a todas las réplicas que producen uno de sus streams de entrada; entonces para mantener la consistencia el nodo debe bloquearse poniendo al sistema fuera de servicio.

La tolerancia a fallas implementada a partir de técnicas de replicación constituye un área ampliamente estudiada. Lo que se desprende de la bibliografía actual es que no es posible proveer al mismo tiempo consistencia y disponibilidad en presencia de particiones en las redes [Bre01, GL02]. Existen muchas aplicaciones de procesamiento de streams que apuntan al monitoreo de tareas que pueden beneficiarse al contar con resultados preliminares, inclusive si dichos resultados son imprecisos debido a

que están basados en un subconjunto de los streams de entrada. Inclusive algunas aplicaciones pueden preferir contar con resultados preliminares aproximados en lugar de esperar por los resultados correctos. Por ejemplo, aun si sólo se encuentra disponible un subconjunto de monitores de red, el procesamiento de sus datos puede ser suficiente para identificar algunas potenciales intrusiones en la red; en este caso una baja latencia de procesamiento es crítica para mitigar ataques. Nuestra propuesta es entonces que los nodos continúen el procesamiento con las entradas disponibles para mantener disponibilidad. Debido a que las aplicaciones cliente suelen monitorear algún fenómeno y esperar pasivamente a que el sistema les devuelva resultados, definimos *disponibilidad* como una latencia baja de procesamiento por tupla. Esta definición difiere de los esquemas de replicación optimistas tradicionales [SS05].

Si bien es claro que mantener disponibilidad es de suma importancia para muchas aplicaciones, no menos importante es obtener los resultados correctos. Por ejemplo en la aplicación de monitoreo de red mencionada en el párrafo anterior, puede ser importante para un administrador de sistemas conocer todos los ataques que ocurrieron en su red o conocer los valores exactos de diferentes cómputos combinados (*e.g.,* tasa de tráfico por cliente). Como consecuencia de ello el desafío principal relacionado con la tolerancia a fallas que atacaremos es asegurar que la aplicación cliente siempre vea los resultados más recientes pero que eventualmente también reciba los resultados de salida correctos, *i.e.,* el sistema debería mantener disponibilidad y proveer consistencia eventual[1].

Para lograr el objetivo anterior el sistema debe permitir que las réplicas estén temporalmente inconsistentes. Medimos inconsistencia contando la cantidad de resultados imprecisos que deben ser corregidos. Dado que es más costoso procesar que corregir resultados en un DSMS, nuestro segundo objetivo consiste en crear nuevas técnicas que minimicen el número de resultados imprecisos, *i.e.,* buscamos minimizar la inconsistencia y al mismo tiempo mantener el nivel de disponibilidad requerido.

El *trade-off* entre disponibilidad y consistencia varía de acuerdo a los requerimientos de cada aplicación de procesamiento de streams, es por ello que proponemos utilizar dichos requerimientos para minimizar la inconsistencia. Algunas aplicaciones (como por ejemplo el monitoreo de pacientes mediante sensores) pueden no tolerar

[1]Corregir resultados previos requiere que las réplicas eventualmente reprocesen las mismas tuplas de entrada en el mismo orden y mediante ello reconcilien sus estados. Llamamos entonces *consistencia eventual* a la garantía de correctitud eventual y está basada en la misma noción introducida en los esquemas de replicación optimistas empleados en bases de datos y *file systems* [SS05], aunque para los DSMS la consistencia eventual también incluye la corrección de los resultados previos.

resultados inconsistentes, en cambio otras aplicaciones (*e.g.,* la detección de intrusos en una red) pueden preferir recibir siempre los datos más recientes; por último otras aplicaciones (por ejemplo el monitoreo de entornos mediante sensores) puede soportar aumentos (limitados) en la latencia de procesamiento si es que ello ayuda a reducir las inconsistencias. Es por dicha diversidad que proponemos permitir que las aplicaciones asignen su trade-off entre disponibilidad y consistencia especificando la cantidad de tiempo adicional que están dispuestas a esperar sus resultados, siempre que esta demora reduzca las inconsistencias. El sistema tiene entre sus objetivos el de minimizar inconsistencias, pero al mismo tiempo debe producir nuevos resultados dentro del límite de tiempo establecido.

La provisión de un trade-off flexible entre disponibilidad y consistencia distingue a DCAA de técnicas previas de tolerancia a fallas para procesamiento de streams que sólo manejan fallas en los nodos [HBR+05] o estrictamente favorecen consistencia sobre disponibilidad [SDBL07, Rab89, CT96, BHS09]. Es por ello que DCAA es más apropiado para muchas aplicaciones de monitoreo, donde la alta disponibilidad y las respuestas casi de tiempo real son preferidas por sobre las respuestas exactas.

Lograr los dos objetivos principales arriba mencionados trae aparejado un conjunto de objetivos secundarios. Primero debemos diseñar una técnica capaz de mantener a las réplicas mutuamente consistentes en ausencia de fallas. En contraste con los sistemas de archivos y bases de datos tradicionales, un DSMS no opera sobre datos persistentes sino sobre un estado temporal grande y velozmente cambiante. Adicionalmente, en un DSMS distribuido la salida producida por un conjunto de nodos sirve como entrada de otros, y también deben mantenerse consistente. Estas dos propiedades hacen que el problema de la replicación sea diferente a los abordados en trabajos previos. Similarmente, los enfoques tradicionales de reconciliación [KBH+88, Urb03] no son apropiados para reconciliar el estado de un DSMS porque el estado de éste no es persistente y depende del orden en el que el DSMS procesó sus tuplas de entrada. Como consecuencia de ello surge la necesidad de investigar nuevas técnicas para reconciliar estados en entornos de DSMS. Luego, para proveer consistencia eventual es necesario que los nodos de procesamiento empleen buffers para las tuplas intermedias. Para el caso de las fallas prolongadas el sistema puede no ser capaz de *bufferear* la totalidad de los datos necesarios y por ello es necesario un mecanismo que gestione los buffers intermedios y las fallas prolongadas. Finalmente, necesitaremos extender los modelos de streams de datos para poder diferenciar una tupla nueva de una que corrige una anterior.

En resumen, si bien la tolerancia a fallas es un área ampliamente estudiada los requerimientos únicos de un entorno de procesamiento de streams crean un nuevo conjunto de desafíos. Proponemos el diseño de un nuevo enfoque especialmente apropiado para DSMS distribuidos. El aporte principal de nuestro enfoque es el de proveer consistencia eventual mientras se mantiene el nivel de disponibilidad definido en la aplicación. El segundo desafío que enfrentaremos es estudiar nuevas técnicas que logren el primer objetivo minimizando las inconsistencias. La Figura 1.5 resume los tipos de fallas y los objetivos de tolerancia a fallas que nuestro DSMS distribuido deberá manejar.

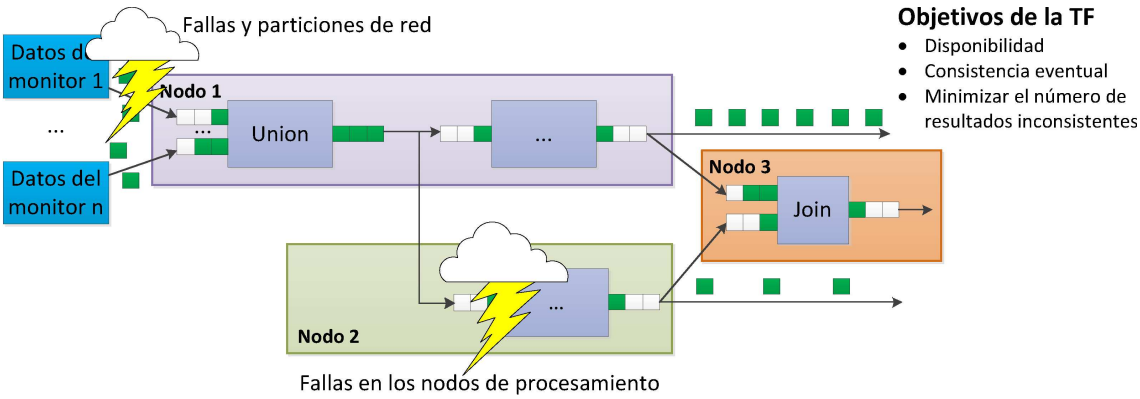


Figura 1.5: Tipos de fallas y los objetivos de tolerancia a fallas

1.3.2 Contribuciones en el área de tolerancia a fallas

La principal contribución en este área es el protocolo DCAA, el cual será descripto en detalle en el Capítulo 4 y logra los objetivos mencionados brevemente en la subsección anterior.

La idea principal de DCAA es favorecer la autonomía de las réplicas. Los nodos detectan independiente las fallas en sus streams de entrada y manejan su propia disponibilidad y consistencia a partir de un autómata finito compuesto por tres estados: estable, falla upstream, y estabilización.

Para asegurar consistencia en las réplicas en el estado estable definimos un operador serializador de datos llamado `SUnion`, el cual toma múltiples streams como entrada y produce un stream de salida con tuplas determinísticamente ordenadas. Adicionalmente empleamos un mecanismo basado en *heartbeats* que permite que un nodo decida a partir de sus entradas cuándo ocurrió una falla. Contando con estas dos técnicas un nodo sólo necesita tener contacto con una réplica (las cuales no necesitan comunicarse entre sí) por cada uno de sus vecinos upstream para mantener la consistencia. Cuando un nodo detecta la falla de uno o más streams de entrada

pasa al estado de falla upstream, donde garantiza que las entradas disponibles sean procesadas dentro del límite de tiempo definido por la aplicación. Al mismo tiempo el nodo trata de evitar (o al menos minimizar) la inconsistencia que se introduce al sistema mediante la búsqueda de réplicas upstream estables, y cuando es necesario, suspender o demorar el procesamiento de nuevos datos. Una vez que la falla se soluciona y un nodo recibe las tuplas faltantes o las entradas inconsistentes, pasa al estado estabilización. Durante la estabilización los nodos reconcilian sus estados y corrigen sus resultados previos, (asegurando consistencia eventual) mientras mantienen la disponibilidad de sus streams de salida. Para lograr consistencia luego de solucionada la falla desarrollamos enfoques basados en técnicas de *checkpoint/redo* y *undo/redo*. Para mantener disponibilidad durante la estabilización desarrollamos un protocolo simple de comunicación inter-réplica que asegura que exista al menos una réplica disponible en todo momento.

Finalmente para soportar DCAA introducimos un modelo de datos de streaming mejorado en el cual los resultados basados en entradas parciales se marcan como tentativos, permitiendo que luego sean modificados; el resto de los resultados se consideran estables e inmutables.

Analizamos en forma teórica que DCAA logra los trade-offs disponibilidad/-consistencia deseados aun frente a diversos escenarios de fallas, incluyendo fallas simultáneas múltiples y fallas durante la recuperación. Descubrimos que lograr que cada nodo se encargue de su propia disponibilidad y consistencia ayuda a manejar estos escenarios de fallas complejos.

Además mostraremos que DCAA presenta un bajo overhead de procesamiento. El principal overhead proviene del buffering de las tuplas necesario para reprocesarlas durante la estabilización. Adicionalmente, para cierto tipo de diagramas de consulta nuestro protocolo soporta fallas largas (tiempo de falla arbitrario), mientras garantiza que una vez solucionada la falla el sistema converja a un estado consistente y las tuplas tentativas más recientes son corregidas.

Si bien para reconciliar el estado de un nodo proponemos el empleo de checkpoint/redo o de undo/redo, emplearemos la primer técnica, porque obtiene como veremos, mejor performance. Empleando checkpoint/redo mostraremos que es posible limitar los checkpoints y reconciliaciones a los *paths* afectados por las fallas y de esta forma evitar overheads en ausencia de las mismas.

DCAA maneja las fallas de corta duración suspendiendo el procesamiento para evitar inconsistencias. De hecho, mostraremos que es posible que el sistema evite inconsistencias frente a cualquier falla de duración más pequeña que la máxima latencia incremental de procesamiento definida en la aplicación, independientemente

del tamaño del sistema distribuido y la ubicación de la falla. Uno de los mayores desafíos de DCAA es garantizar el procesamiento de todas las tuplas disponibles en un período predefinido de tiempo en presencia de fallas prolongadas y al mismo tiempo mantener consistencia eventual. Para lograr esta consistencia para cualquier instante de tiempo es necesario efectuar las repeticiones y correcciones en background. Cada vez que un nodo necesita procesar o incluso simplemente recibir una tupla vieja debe continuar procesando las tuplas de entrada más recientes.

Para finalizar, podemos decir que DCAA encapsula un único esquema capaz de proveer tolerancia a una variedad de fallas a un DSMS distribuido, dejando a discreción de la aplicación elegir el trade-off entre disponibilidad y consistencia.

1.3.3 Desafíos en gestión de carga

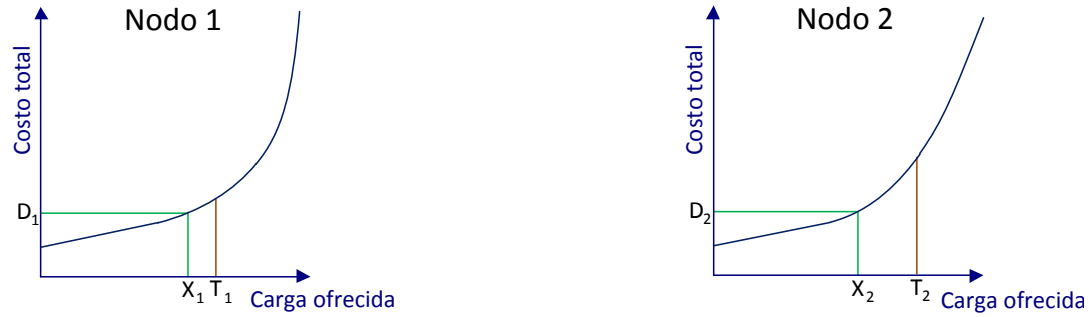
El segundo problema en el que nos enfocamos es el de gestión de variaciones de carga en un sistema distribuido y federado.

Cuando un usuario envía una consulta a un DSMS distribuido se le asignan operadores individuales a varios nodos de procesamiento. Las consultas son continuas y de larga duración y por ende la cantidad de recursos (memoria, CPU, ancho de banda de red) que utiliza cada operador varía durante el tiempo de vida de la consulta. Estas variaciones son típicamente causadas por los cambios en las frecuencias de los datos o en las distribuciones de los datos en los streams de entrada. Los usuarios pueden a su vez agregar o quitar consultas dinámicamente, aumentando aun más las variaciones en la carga. En respuesta a estos desafíos el sistema puede eventualmente necesitar modificar las asignaciones originales de los operadores en los nodos para lograr mejorar la performance global o al menos evitar degradaciones severas de la misma.

El problema de gestionar la carga dinámicamente en un sistema distribuido mediante la reasignación de tareas o recursos ha sido suficientemente estudiado, [ELZ86, Gal77, KS89]. Las técnicas tradicionales definen una función de utilidad global del sistema, como el tiempo total de procesamiento promedio [Gal77, KS89] o el *throughput* [KS89], y a partir de ella buscan una asignación que optimice dicha utilidad. La Figura 1.6(a) muestra un ejemplo de un sistema con dos nodos de procesamiento donde cada nodo tiene una función que a partir de una dada carga total devuelve el costo total de procesamiento[2]. En este ejemplo una asignación de carga óptima minimiza la suma de los costos de procesamiento (*i.e.,* minimiza $D_1 + D_2$); al hacerlo la asignación optimiza la utilidad global por el número de

[2]Este costo podría representar por ejemplo el tamaño de cola medio en un nodo.

(a) Asignación de costo mínimo. El costo total, $D_1 + D_2$, es menor al costo de cualquier otra asignación.



(b) Asignación aceptable. Para cada nodo, la carga X_i se encuentra por debajo de un umbral predefinido T_i.

FIGURA 1.6: Diferentes tipos de asignaciones de carga en un sistema de dos nodos

mensajes encolados en el sistema.

La mayoría de los enfoques previos asumen un entorno colaborativo, donde todos los nodos trabajan juntos para maximizar la performance global del sistema. Hoy en día los sistemas distribuidos suelen desplegarse sobre entornos federados, donde las diferentes organizaciones autónomas son dueñas y administran subconjuntos de nodos de procesamiento y recursos. Nuestro objetivo es lograr gestión de carga en estos sistemas federados en auge, como por ejemplo los *grids* compuestos por computadoras pertenecientes a diferentes dominios [BlDGs03, TL03, FK03, BAG00, BSG+01, FK98, TTL05], plataformas *overlay* como Planetlab [PACR03], *web services* [W3C02, KL03] donde se requiere procesamiento por parte de diferentes organizaciones, y en sistemas *peer-to-peer* [CFV03, DKK+01, LFSC03, NWD03, RD01, SMK+01, VCCS03, LS05, AJ06]. Si bien nuestra motivación surge del procesamiento de streams, dado que el enfoque es general puede ser empleado en cualquier sistema federado.

Muchas son las cuestiones que hacen que la gestión de carga en un sistema federado sean desafiantes. Especialmente el hecho de que los participantes no buscan optimizar una utilidad global del sistema sino que están motivados por intereses propios, colaboran con otros nodos sólo si esta colaboración mejora su utilidad.

Por lo tanto necesitamos diseñar un mecanismo que provea incentivos para que los participantes estén interesados en manejar la carga de sus pares.

Una economía computacional donde los participantes proveen recursos y realizan cómputos para otros participantes a cambio de un pago[3] constituyen las bases de una técnica natural que facilita la gestión de carga colaborativa entre participantes egoístas (*selfish*). Existen varias propuestas de economías computacionales [BAG00, BSG+01, SAL+96, WHH+92], sin embargo dado que ninguna de ellas logró ser aceptada popularmente en la práctica, creemos que las economías computacionales tradicionales no brindan una solución adecuada al problema. Si bien proveen un mecanismo que permite a los participantes mejorar su utilidad al realizar cómputos para otros, esto parece no ser suficiente como para motivar un mayor grado de adopción.

En la práctica, y contrastando las economías computacionales tradicionales, frecuentemente se emplean *acuerdos bilaterales* que posibilitan las colaboraciones entre nodos autónomos. Los *Service Level Agreements* (SLAs) gobiernan las relaciones entre clientes y proveedores de servicios [Rac11, Ver99, Wus02], entre compañías interconectadas mediante *web services* [Cor11, KL03, GS05], o entre *Internet Service Providers* (ISPs) quienes acuerdan transportar el tráfico del otro [FSP00, WS04]. En forma similar a las economías computacionales, los acuerdos bilaterales ofrecen incentivos de colaboración a los participantes, pues ellos acuerdan recompensar a sus pares por el servicio que proveen. Además los acuerdos bilaterales proveen a los partners capacidades adicionales, como privacidad en las interacciones entre ellos y discriminación de precio y servicios [LM02] (donde un participante ofrece diferentes calidades de servicio y diferentes precios a distintos partners). Estos acuerdos se basan en relaciones de larga duración, y por ello al ser de a pares se cuenta con estabilidad y predictibilidad en las interacciones entre participantes. Finalmente, los acuerdos hacen que las interacciones en tiempo de ejecución sean más simples y livianas que las economías computacionales debido a que la mayor parte de las condiciones se negocian previamente, reduciendo el overhead en tiempo de ejecución.

Nuestro objetivo es desarrollar un mecanismo de gestión de carga que permita a un sistema federado lograr una buena distribución de carga que provea los mismos beneficios que brindan los acuerdos bilaterales: incentivos para colaborar, privacidad en las interacciones, predictibilidad en los precios y en los movimientos de carga, estabilidad en las relaciones y simplicidad en tiempo de ejecución.

[3] También son posibles modelos que no contemplan el uso de pagos, como por ejemplo el *bartering* (trueque) [CFV03], como veremos en el Capítulo 2.

1.3.4 Contribuciones en gestión de carga

Proponemos un mecanismo distribuido llamado *Mecanismo de Precio Acotado* (MPA) para la gestión de carga en un sistema federado basado en contratos privados entre pares. A diferencia de las economías computacionales que usan subastas o implementan mercados globales para asignar precios a los recursos en tiempo de ejecución, nuestro mecanismo se basa en la negociación *offline* de los contratos.

Los contratos asignan limites acotados de precios para la migración de cada unidad de carga entre dos participantes y pueden especificar el conjunto de tareas que cada uno de los dos nodos está dispuesto a efectuar en nombre del otro. Bajo MPA las transferencias de carga en tiempo de ejecución ocurren únicamente entre participantes que cuentan con contratos prenegociados y a un precio por unidad dentro del rango contratado. El mecanismo de transferencia de carga es sencillo: un participante migra carga hacia otro sólo si el costo de procesamiento local esperado para el siguiente período de tiempo es mayor que el pago esperado que hubiese tenido que hacer a otro participante para procesar la misma carga (más el costo de migración).

En el Capítulo 5 fundamentaremos por qué un balance de carga óptimo no constituye un objetivo de importancia en un sistema federado, teniendo en cuenta que la mayoría de los participantes contará con recursos suficientes para proveer un buen servicio a sus clientes la mayor parte del tiempo. Nuestro objetivo principal es conseguir una asignación aceptable, es decir una distribución de carga donde ningún participante opera por sobre su capacidad o, si el sistema como un todo se encuentra sobrecargado, entonces todos los participantes operan por encima de su capacidad. La Figura 1.6(b) ilustra una asignación aceptable donde puede verse que una asignación no minimiza la suma de todos los costos de procesamiento sino que asegura que el nivel de carga X_i de cada nodo se encuentre por debajo de su capacidad predefinida T_i. Nuestro trabajo difiere entonces de las investigaciones anteriores en que nos enfocaremos en lograr una asignación aceptable en lugar de buscar un balance de carga óptimo, especialmente en ambientes federados con participantes egoístas.

El mecanismo propuesto provee suficientes incentivos para que participantes egoístas manejen el exceso de carga de otros con el objetivo de mejorar la distribución de carga del sistema. Mostraremos además que el mecanismo distribuye eficientemente el exceso de carga cuando la carga total se encuentra por debajo y por encima de la capacidad total del sistema. Dicho mecanismo asegura una buena estabilidad: maneja la mayoría de las variaciones de carga sin reasignar ninguna.

Veremos que es suficiente para cada participante contar con sólo unos pocos contratos para obtener la mayor parte de los beneficios del sistema. Por otra parte, los contratos sólo deben especificar un pequeño rango de precios para que el mecanismo siempre pueda converger a asignaciones aceptables. A partir de un rango de precios mostraremos que la estrategia óptima para los participantes es acordar rápidamente un precio final, de hecho la mayoría de las veces lo mejor es no negociar. Incluso mostraremos que nuestro mecanismo funciona bien aún si los participantes establecen contratos heterogéneos a diferentes unidades de precio con cada uno de ellos. Por último veremos que nuestro enfoque trae aparejado un bajo overhead en tiempo de ejecución.

En resumen MPA es un mecanismo simple, liviano y práctico que propicia relaciones offline duraderas entre participantes para lograr las siguientes propiedades de gestión de carga: *asignaciones de carga aceptables y estables*. Al mismo tiempo MPA provee a los participantes privacidad, predictibilidad en los precios y posibles movimientos de carga en tiempo de ejecución además de la posibilidad de personalizar servicios.

1.4 Organización de la tesis

El resto de este trabajo se organiza de la siguiente manera. En el Capítulo 2 se presenta un resumen de trabajos relacionados con el procesamiento de streams, sistemas de gestión de datos y gestión de carga en entornos federados. El Capítulo 3 muestra la arquitectura para un sistema de características relevantes a nuestra investigación. El Capítulo 4 describe nuestra propuesta referida a la tolerancia a fallas, y a continuación, el Capítulo 5 describe en detalle a MPA como mecanismo de gestión de carga para sistemas federados. Por último, el Capítulo 6 analiza las conclusiones y propone futuras líneas de trabajo.

1.5 Resumen

Este capítulo comienza presentando algunas nociones básicas relacionadas con el área de estudio. Luego se plantean las propiedades más importantes de las aplicaciones de procesamiento de streams. A continuación se analizan más profundamente los *sistemas de gestión de streams de datos* (DSMS) centralizados y distribuidos. Finalmente se presentan los dos puntos fundamentales de este trabajo: *tolerancia a fallas en un DSMS distribuido y gestión de carga en entornos federados*, en particular se resumen los desafíos y contribuciones desarrollados en estas áreas.

Capítulo 2

Antecedentes y Trabajos Relacionados

Índice

2.1. Los DBMSs tradicionales y sus extensiones 21
2.1.1. Extensiones a los DBMSs 22
2.1.2. Tolerancia a fallas en los DBMSs tradicionales . . . 25
2.2. Data stream management systems (DSMS) 27
2.2.1. Investigación en el área de los DSMS 28
2.2.2. Tolerancia a fallas en los DSMS 31
2.3. Sistemas workflow 32
2.4. Sistemas de Publicación/Subscripción 33
2.5. Redes de sensores 34
2.6. Replicación de máquinas de estados 35
2.7. Recuperación mediante rollback 36
2.8. Gestión de carga 36
2.8.1. Algoritmos de gestión de carga cooperativos 36
2.8.2. Diseño de mecanismo algorítmico distribuido 38
2.8.3. Gestión de carga basada en modelos económicos . . 39
2.8.4. Recursos compartidos en sistemas peertopeer 40
2.8.5. Service Level Agreements 40
2.9. Resumen 40

History is philosophy from examples.

— Dionysius of Halicarnassus

En este capítulo discutiremos detalladamente las diferencias entre los DBMSs tradicionales y los DSMS. También presentaremos trabajos relacionados con tolerancia a fallas en DBMSs, DSMS y otros sistemas de gestión de datos, y trabajos relacionados sobre gestión de carga en sistemas federados.

Más específicamente, en la Sección 2.1, discutiremos las limitaciones de los DBMSs tradicionales para el soporte de aplicaciones de procesamiento de streams, y presentaremos varias de las extensiones que han sido propuestas en el pasado para mejorar las capacidades DBMS. Además discutiremos las técnicas tradicionales para lograr tolerancia a fallas. En la Sección 2.2 se presentan los proyectos de investigación más importantes en el área de los DSMS y se discuten los esfuerzos específicos para lograr tolerancia a fallas en el procesamiento de streams. En las Secciones 2.3 a 2.7, se discute la tolerancia a fallas en los sistemas *workflow*, *publish/subscribe*, *sensor networks*, *state-machine replication*, y *rollback recovery*. Finalmente, en la Sección 2.8, se presentan los trabajos relacionados en el campo de la gestión de carga en sistemas federados.

2.1 Los DBMSs tradicionales y sus extensiones

Las aplicaciones de procesamiento de streams requieren procesar (con baja latencia) continuos y grandes volúmenes de datos. Estos requerimientos se contraponen con las capacidades de los DBMSs tradicionales en tres formas.

La primera es que las aplicaciones de procesamiento de streams requieren modelos de datos nuevos o al menos extendidos, operadores y lenguajes de consulta. Law et al. [LWZ04] estudiaron formalmente los problemas entre las capacidades de los SQL y los requerimientos de las consultas de los DSMS. El problema principal es que el álgebra relacional asume que los datos están acotados en tamaño, lo cual se refleja en el diseño de sus operadores. Por ejemplo, algunos operadores relacionales (*e.g.,* los operadores de agregación, como min, max, o promedio) pueden necesitar procesar toda su entrada para poder producir un resultado. Las aplicaciones de procesamiento de stream típicamente monitorean algún fenómeno que está ocurriendo en ese preciso momento y necesita actualizaciones periódicas en lugar de un único valor al final del stream. Otros operadores relacionales, como join, acumulan estado que crece según aumenta el tamaño de sus entradas. Este tipo de gestión de estado es inapropiado para los streams, donde las tuplas arriban continuamente y la entrada puede ser potencialmente ilimitada en cuanto a tamaño. Para solucionar estos problemas, la mayoría de los DSMS proponen el uso de algún tipo de especificación de ventanas [ABB+04, AM03, CCD+03] para agrupar las tuplas en grupos.

Segundo, el modelo de procesamiento de datos tradicional requiere que los datos sean almacenados persistentemente e indexados antes de que las consultas sean ejecutadas sobre los datos. Los DBMSs asumen que pueden decidir los movimientos de los datos desde y hacia disco y que pueden examinar (de ser necesario) las tuplas múltiples veces. Estas asunciones y este modelo de procesamiento no resisten el procesamiento continuo de streams de entrada. Los DSMS proponen una arquitectura donde los datos se procesan continuamente según arriban antes de que sean almacenados (incluso podrían ni siquiera almacenarse) [ABB+04, CCD+03, CJS03, SH98]. Para medir y comparar la performance de los DSMS puede emplearse *Linear Road* [ACG+04], un *benchmark* que muestra que para el procesamiento de streams, un DSMS especializado obtiene al menos cinco veces más performance que los DBMSs tradicionales.

Finalmente, los DBMSs tradicionales tienen soporte limitado para manejar alertas y consultas continuas, las cuales constituyen tipos de consultas esenciales para el procesamiento de streams. Un tipo de consulta continua es la *vista materializada* [GM95, GMRR01], es una relación derivada definida en términos de relaciones de bases almacenadas. Una vista materializada es entonces como una memoria cache que debe ser actualizada (incrementalmente o mediante re-ejecución) según van cambiando las relaciones base. Sin embargo, no está diseñada para el estilo de procesamiento *push* requerido por las aplicaciones de procesamiento de streams, por el contrario el usuario debe solicitar (*poll*) el estado de una vista materializada.

2.1.1 Extensiones a los DBMSs

Las aplicaciones de procesamiento de streams no son las primeras en romper el modelo de las bases de datos tradicionales. Antes de los DSMS, ya se habían propuesto varias extensiones a los DBMSs. Algunas de estas extensiones, como los *triggers*, procesamiento de memoria principal, procesamiento en tiempo real, soporte para los datos secuenciados, y la noción de tiempo, traen aparejadas similitudes con los objetivos de los DSMS. A continuación se presenta un resumen de estas propuestas iniciales.

Un DBMS tradicional efectúa solamente consultas de única vez sobre las relaciones persistentemente almacenadas. Las bases de datos activas [Han92, PD99, SPAM91, SK91, WF90] apuntan a mejorar un DBMS agregándole capacidad para monitoreo y alertas. En bases de datos comerciales, los *triggers*, [HCH+99, BDR04, LBK01, GSS01, Urb03] constituyen la técnica más frecuente para convertir una base de datos pasiva en una activa. Un trigger es una acción predefinida ejecutada

por el DBMS cuando alguna combinación de eventos y condiciones preestablecidas ocurre[1]. En contraste a los DSMS, las bases de datos activas siguen operando sobre el estado actual de las relaciones localmente almacenadas, simplemente monitorean este estado y reaccionan ante los cambios en el mismo.

Una implementación naive de triggers no escala correctamente debido a que cada evento requiere que el sistema busque todos los triggers relevantes para encontrar las condiciones correspondientes y verificar que cada condición pueda corresponder a una (costosa) consulta. Para lograr una mayor escalabilidad, los triggers pueden agruparse en clases de equivalencia con un índice para cada una de las diferentes condiciones [HCH+99]. Sin embargo, los DSMS presentan un nuevo modelo de procesamiento más escalable y adecuado para el procesamiento de streams que el modelo basado en triggers.

En el modelo relacional, una decisión importante de diseño es la de tratar a las relaciones como un conjunto desordenado de tuplas. Esta elección facilita la optimización de consultas pero falla en soportar aplicaciones que necesitan expresar consultas sobre secuencias de datos en lugar de sobre conjuntos de datos. Un ejemplo de una aplicación de este tipo se encuentra en el área de servicios financieros, donde las aplicaciones necesitan examinar las fluctuaciones en el precio de las acciones. Una solución a este problema la hallamos en las bases de datos de secuencias (*sequence databases*) [JMS95, LS03, Mel02, SLR96].

En las bases de datos de secuencias encontramos nuevos modelos de datos, como por ejemplo los datos *array* [Mel02], y los tipos de datos abstractos relacionados con secuencias, o nuevos operadores, como los que implementan variantes de operadores relacionales dedicados a preservar el orden. Además, estas bases de datos implementan lenguajes de consulta (*e.g.,* soporte para cláusulas `ASSUMING ORDER on` o definiciones de ventanas de cómputo) para soportar secuencias de datos sobre un motor relacional. Las bases de datos de secuencias influenciaron en gran medida el diseño de los lenguajes de consulta para streams (*e.g.,* [ABW02, CCD+03, CDTW00, GÖ05]). Sin embargo, estos últimos enfrentan algunos desafíos adicionales ignorados por las bases de datos de secuencias. En el procesamiento de streams los datos no son almacenados y no están rápidamente disponibles; por el contrario, fluyen continuamente, posiblemente a alta velocidad. Las tuplas pueden arribar al menos fuera de orden pero no es posible esperar hasta el final del stream para ordenar todos los datos y comenzar a procesarlos. Por ello, los DSMS asumen desorden acotado [ABW02] y descartan las tuplas fuera de orden, o utilizan constructores como *puntuación*

[1] Tanto las acciones como los eventos son operaciones de bases de datos.

[TMSF03, TMSS07] para poder tolerar el desorden.

En los DBMSs tradicionales los usuarios consultan el estado *actual* de la base de datos. No existe la noción de línea de tiempo que muestre cuándo ocurrieron las distintas actualizaciones o cuándo tuplas diferentes fueron válidas. Las bases de datos temporales [OS95, SA85] tienen en cuenta este problema y para solucionarlo soportan una o ambas de las siguientes nociones de tiempo. El *tiempo de transacción* (*i.e.,* tiempo físico) es el tiempo en el que se almacena la información en la base de datos, empleando esta noción de tiempo, una aplicación puede examinar el estado de la base de datos en diferentes momentos. Por ejemplo, una aplicación puede examinar el estado de la base de datos del último viernes. El *Tiempo de validez* (*i.e.,* tiempo lógico) es el tiempo en el cual la información almacenada modela la realidad y permite que las aplicaciones puedan almacenar y editar (de ser necesario) la historia. Una aplicación puede por ejemplo almacenar la historia del sueldo de un empleado a lo largo del tiempo y modificar el tiempo en el que un empleado recibió un aumento si este tiempo fue inicialmente erróneamente ingresado.

Todas las bases de datos actuales soportan una tercer noción adicional de tiempo, se trata simplemente de un atributo de *estampilla de tiempo* definido por el usuario. Los DSMS no cuentan con este poderoso tipo de operaciones basadas en tiempo. Las tuplas incluyen típicamente una estampilla de tiempo, pero principalmente sirve para determinar el orden de las tuplas más que para cuestiones de tiempo propiamente dicho.

Las *bases de datos de tiempo real* [KGM95, LSHW00, OS95] son bases de datos en las que las transacciones tienen *deadlines* y el objetivo del sistema es completar transacciones cumpliendo con los mismos. Los DSMS buscan latencias de procesamiento bajas pues si bien soportan aplicaciones de monitoreo no son capaces de ofrecer garantías.

Los DBMSs tradicionales operan sobre datos almacenados en disco, a diferencia de las bases de datos de memoria principal [GMS92], donde todos los datos residen permanentemente en memoria y sólo los backups se almacenan en disco. Obviamente los accesos a memoria son mucho más rápidos que los accesos a disco y por ende las bases de datos de memoria principal obtienen gran performance. Este tipo de arquitectura es especialmente aplicable cuando se tienen pequeños conjuntos de datos y pocos requerimientos de procesamiento de tiempo real [GMS92]. Los DSMS actuales constituyen bases de datos de memoria principal y en contraste con propuestas anteriores, su arquitectura apunta a aplicaciones de streaming. Es claro que no se puede asumir que todos los datos entrarán en memoria y por ello a medida que se reciben nuevos datos se van descartando datos viejos.

Las *bases de datos adaptativas* solucionan otro problema presente en los DBMSs tradicionales. Con el advenimiento de Internet los DBMSs dejan de procesar datos almacenados localmente para procesar datos remotos [ILW+00]. Los datos remotos pueden pertenecer a diferentes dominios administrativos y por ende es difícil sino imposible que el optimizador de consultas pueda obtener información estadística precisa sobre los mismos (por ejemplo la cardinalidad o la taza de arribos) para decidir sobre el plan de consultas, por el contrario, el procesador de consultas debe ser capaz de reaccionar y adaptarse a cambios o a condiciones inesperadas [CNB00a, IDR07, UF01]. Esta adaptación debe tener la capacidad de cambiar el orden en el que se procesan las tuplas según progresa la consulta. Dado que los DSMS también reciben datos remotos también deben tener capacidad de adaptación, capacidad desafiante debido a que como las consultas son continuas y de larga duración provocando que sea altamente probable que las condiciones cambien durante la ejecución de la consulta.

La Tabla 2.1 resume las diferentes extensiones a los DBMSs tradicionales enfatizando sus similitudes y diferencias con los DSMS.

Extensión DBMS	**Similitudes con los DSMS**	**Diferencias con los DSMS**
Activa	Capacidad de monitoreo	El modelo de procesamiento tradicional de las bases de datos activas apunta a rates bajos de eventos. Las bases de datos activas operan sobre el estado actual de las relaciones almacenadas localmente.
Secuencia	Noción de ordenamiento de tuplas	Las bases de datos secuenciales asumen secuencias acotadas en tamaño y almacenadas localmente. En los DSMS los datos se envían al sistema continuamente.
Temporal	Noción de tiempo	Las bases de datos temporales permiten consultas tanto sobre estados de bases de datos físicos como lógicos. En contraste los DSMS ordenan las tuplas mediante *timestamps*.
Tiempo real	Procesamiento con deadlines	Las bases de datos de tiempo real tratan de completar las consultas dentro del deadline prefijado. Los DSMS proveen baja latencia de procesamiento de consultas contínuas pero sin brindar garantías.
Memoria principal	Mantienen los datos en memoria	Los DBMS que mantienen los datos en RAM asumen que todas las tuplas caben en memoria. Los DSMS reciben tuplas continuamente y descartan las tuplas viejas.
Adaptiva	Reaccionan a las condiciones cambiantes durante la ejecución de las consultas	Los DBMS adaptivos se diseñan para consultas de única vez (*one time*) sobre conjuntos de datos acotados en tamaño en lugar del procesamiento sobre consultas continuas.

Tabla 2.1: Extensiones a los DBMS y sus similitudes y diferencias con los DSMS

2.1.2 Tolerancia a fallas en los DBMSs tradicionales

La tolerancia a fallas en los DBMSs tradicionales usualmente se implementa mediante la ejecución de una o más réplicas del motor.

La técnica más simple para implementar tolerancia a fallas es tener un server backup esperando (*standby*) que se haga cargo de las operaciones tan pronto el motor principal falle; este modelo típicamente se conoce con el nombre de *pares de procesos* (*process-pair*) [BGH86, Gra85]. Existen variantes de este modelo que difieren en el overhead en tiempo de ejecución y en el nivel de protección que proveen. En la variante *cold-standby* el primario transmite en forma periódica un log de operaciones al backup, y éste último procesa estas operaciones asincronicamente. En este modelo una falla en el primario puede causar la pérdida de todas las operaciones que no se transmitieron al backup. Alternativamente a los logs de operaciones, el primario puede emplear checkpoints incrementales de su estado. En la variante *hot-standby* tanto el servidor primario como el backup realizan todas las operaciones sincronicamente, *i.e.,* ambos efectúan cada actualización antes de devolver el resultado al cliente. El modelo pares de procesos se encuentra ampliamente difundido entre muchos de los DBMSs actuales [CM01, Cor02].

La metodología de pares de procesos protege el sistema solamente contra fallas de nodos y con un único backup sólo puede fallar un único nodo para que el sistema permanezca disponible. Para soportar fallas en la red y un mayor número de fallas en los nodos es necesario un mayor número de réplicas dispersas en la red. Los esquemas de tolerancia a fallas con un mayor número de réplicas pueden clasificarse en: replicación *eager* o replicación *lazy*.

La replicación eager favorece la consistencia al tener una mayoría de réplicas efectuando cada actualización como parte de una única transacción [GMB85, Gif79]; sin embargo sacrifica disponibilidad al forzar que particiones minoritarias se bloqueen. La replicación lazy favorece la disponibilidad por sobre la consistencia. Con dicha metodología de replicación todas las réplicas procesan (posiblemente) actualizaciones conflictivas aún cuando se encuentren inconexas y luego reconcilien su estado mediante reglas de reconciliación [KBH+88, Urb03], por ejemplo preservando solamente la versión más reciente de un registro [GHOS96]. No es sencillo definir reglas para que un DSMS alcance un estado consistente, pues los DSMS operan sobre un gran estado cambiante que depende del orden exacto en el que fueron procesadas sus tuplas de entrada.

Las técnicas de replicación lazy emplean transacciones tentativas durante las particiones y reprocesan las transacciones, posiblemente en un orden distinto, durante la reconciliación [GHOS96, TTP95] logrando que eventualmente todas las réplicas tengan el mismo estado[2]. Nuestra técnica aplica la idea de resultados tentativos al

[2]Y este estado se corresponde a una ejecución serializable en un único nodo

procesamiento de streams, sin embargo no podemos utilizar en forma directa los mismos conceptos y técnicas: las transacciones se procesan en forma asilada mientras que las tuplas en un DSMS pueden ser (posiblemente todas) parte de una única consulta, y por lo tanto no existe el concepto de transacción. Las tuplas se agregan (*Aggregate*) y correlacionan en lugar de procesarse aisladamente; por esta razón no podemos reutilizar la noción de transacción tentativa e introducimos entonces la noción de tuplas estables y tentativas. Adicionalmente definimos disponibilidad como el procesamiento con baja latencia de los datos de entrada más recientes. Para ello debemos diseñar un protocolo que mantenga esta propiedad a pesar de las fallas y las reconciliaciones (los sistemas de replicación lazy no tienen esta restricción). Finalmente, en un DSMS distribuido la salida de los nodos de procesamiento constituye la entrada de otros nodos y eventualmente también éstos deben ser consistentes.

Existen técnicas pertenecientes al ámbito del procesamiento de consultas que, al igual que sucede con nuestro objetivo de baja latencia de procesamiento, producen trade-offs entre velocidad en los resultados y consistencia. El objetivo de la mayoría de las técnicas es producir resultados parciales significativos lo antes posible durante la ejecución de consultas de larga duración. Trabajos anteriores, como [HHW97, TGO99] atacaron el problema de computar agregados en tiempo de ejecución, sin embargo estos esquemas asumen que los datos están disponibles localmente y que el procesador de consultas puede elegir el orden en el que lee y procesa tuplas (por ejemplo puede tomar muestras de las entradas). Esta asunción no se mantiene para las aplicaciones de procesamiento de streams, pues procesan los datos según van arribando, enfocándose en materializar los resultados de salida de a uno por vez [RH02]. Sin embargo estos esquemas requieren que el procesador de consultas *bufferee* los resultados parciales hasta que la consulta se complete, volviéndolos inapropiados para streams de datos ilimitados y fallas de larga duración.

Finalmente algunas técnicas de procesamiento distribuido de consultas ofrecen control de grano fino bajo el trade-off entre precisión en las consultas (*i.e.,* consistencia) y performance (*i.e.,* utilización de recursos) [OJW03, Ols03]. Los usuarios especifican un umbral de consistencia y el sistema trata de minimizar la utilización de recursos mientras asegura la consistencia deseada. Los usuarios también pueden dejar que el sistema optimice la consistencia dado un umbral de utilización máxima de recursos.

2.2 Data stream management systems (DSMS)

Los DSMS solucionan las limitaciones de los DBMSs tradicionales mediante nuevas técnicas que proveen procesamiento de baja latencia de grandes cantidades de consultas continuas sobre streams de datos. La Tabla 2.2 resume las principales diferencias entre los DBMSs y los DSMS.

Característica	DBMS	DSMS
Ubicación de los datos	persistentemente almacenados	streaming
Tamaño de los datos	limitado	potencialmente streams ilimitados
Modelo de datos	conjuntos desordenados	secuencias *append-only*
Operadores	relacionales, pueden bloquearse	en gral. operadores en ventanas
Consultas	en gral. de una vez	en gral. contínuas
Modelo de procesamiento	almacenar, indexar, procesar	procesar al arribar, opcionalmente almacenar

Tabla 2.2: Principales diferencias entre los DBMSs y los DSMS

2.2.1 Investigación en el área de los DSMS

A continuación presentamos algunos de los principales proyectos de investigación el el área de procesamiento de streams y de procesamiento continuo de consultas. La Tabla 2.3 resume estos proyectos en orden cronológico de comienzo junto con sus principales contribuciones. Luego de introducir estos proyectos discutiremos de qué forma tratan de conseguir tolerancia a fallas en los DSMS.

Proyecto	Principales características
Tapestry	Uno de los primeros procesadores de consultas contínuas, basado en un DBMS.
Tribeca	Uno de los primeros DSMS orientados al monitoreo de las redes.
NiagaraCQ	Sistema que aplica consultas continuas y escalables sobre documentos XML. Escalabilidad de hasta millones de consultas simultáneas.
TelegraphCQ	Implementa procesamiento continuo y adaptable de datos a medida que van arrivando. Soporta consultas de datos previos, gestión de carga y tolerancia a fallas en flujos de datos paralelos.
STREAM	Soporta consultas continuas sobre streams y sobre relaciones almacenadas. Explora el procesamiento eficiente en un sitio único.
Aurora	Expresa consultas componiendo diagramas de flujo de datos a partir de operadores. Explora el procesamiento eficiente en un sitio único optimizando QoS en streams de salida.
Gigascope	DSMS diseñado para el monitoreo de redes. Las consultas se compilan en el código. Se enfoca en el procesamiento de streams de alto flujo de datos.

Tabla 2.3: Consultas contínuas y DSMS

Tapestry [TGNO92, ZHS+04, ZKJ01] fue uno de los primeros sistemas que introdujo la noción de consulta continua con estado. Tapestry está construido sobre un DBMS tradicional y por ende primero almacena e indexa datos antes de incluirlos en una consulta continua. Esta implementación presenta la ventaja de que requiere sólo pequeños cambios sobre el DBMS pero también tiene la fuerte desventaja de que no escala con respecto al rate de tuplas en los streams de entrada y el número

de consultas continuas. Este sistema se enfoca únicamente en los repositorios de datos *append only*, como bases de datos de emails. El objetivo del sistema es permitir a los usuarios el envío de consultas continuas que identifiquen documentos de su interés y notifique a los mismos cuando esos documentos son agregados al repositorio. Tapestry soporta solamente consultas monótonas: devuelve todos los registros que satisfacen la consulta en algún momento del tiempo (no indica si un registro ya no la satisface). Las consultas Tapestry se escriben en TQL, un lenguaje similar al SQL que transforma las consultas de usuario en consultas incrementales que se ejecutan periódicamente. Dichas consultas incrementales son significativamente más eficientes que re-ejecutar nuevamente consultas completas. Cada ejecución produce un subconjunto de los resultados finales

Tribeca [Sul96, SH98] fue uno de los primeros DSMS enfocados en el monitoreo de las aplicaciones de red. Sus consultas se expresan en forma de dataflows orientados específicamente a lenguajes de consulta. Las consultas pueden tener solamente un único stream de entrada pero pueden tener múltiples streams de salida. Si bien este sistema tiene soporte limitado para las operaciones de join soporta agregación por ventanas e incluye operadores para el particionado y *merging* de streams (estos operadores son similares a `Group-by` y a `Union` respectivamente).

NiagaraCQ [CDTW00] es un sistema que aplica consultas continuas y escalables sobre documentos XML. Lo más destacado (incluso novedoso) de NiagaraCQ es su capacidad de soportar potencialmente millones de consultas simultáneas agrupándolas dinámica e incrementalmente según las similitudes en sus estructuras. A medida que van llegando nuevas consultas se van incorporando a los grupos existentes o, de ser necesario, los grupos se *refactorean*. Agrupar consultas mejora indudablemente la performance, pues muchas consultas comparten cómputo y porque se tiene una menor cantidad de consultas en ejecución, y por ello es más probable que todas puedas residir en memoria al mismo tiempo y cuando arriban nuevos datos el control de qué consultas deben ser ejecutadas se realiza a nivel del grupo. NiagaraCQ incluso agrupa consultas que deben ejecutarse periódicamente junto con aquellas que se ejecutan cuando ocurre un evento específico. Por último cabe destacar que este sistema estudia nuevas técnicas de optimización sobre streams [VN02, MVLL05], como por ejemplo el uso de *rates* en streams de entrada como métrica en lugar de las técnicas tradicionales, generalmente basadas en información de cardinalidad de las relaciones.

TelegraphCQ [CCD+03] fue uno de los primeros sistemas que implementó el procesamiento continuo de datos a medida que van arrivando en lugar de almacenarlos primero. Las consultas ingresadas por el usuario se expresan en un SQL levemente

modificado con el agregado de una cláusula adicional que define las ventanas de entrada en donde se realiza el cómputo. El énfasis en este sistema está en el procesamiento adaptivo [CNB00a, DH04, MSHR02], el cual se realiza mediante el ruteo de tuplas por módulos de consulta. Los grupos de módulos de consulta se conectan entre sí y van procesando tuplas y tomando decisiones de ruteo basados en los cambios en las condiciones. Esta capacidad de adaptación mejora la performance frente a condiciones variables durante la ejecución de las consultas pero al mismo tiempo aumenta el nivel de overhead debido a que las tuplas deben incluir información de *linaje* y el sistema debe tomar decisiones de ruteo para cada tupla. Un segundo aspecto interesante de TelegraphCQ es que considera el procesamiento de streams como una operación de join entre un stream de datos y un stream de consultas. Las consultas y las tuplas se almacenan en módulos de estado separa y cuando una nueva consulta arriva se usa su predicado para probar los datos y viceversa [CF02]. Esta perspectiva le permite incluir datos viejos en los resultados de una consulta continua recién insertada y por lo tanto presenta buen soporte tanto para consultas sobre datos actuales como viejos [CF02, CF04]. Por último cabe destacar que éste soluciona parcialmente (sólo para flujos de datos paralelos) el problema de la tolerancia a fallas y la gestión de carga bajo procesamiento de streams [BHS09, SHCF03].

El proyecto STREAM [ABB+04] explora varios aspectos relacionados con el procesamiento de streams: un nuevo modelo de datos y un lenguaje de consultas aplicable a streams [ABW02, SW04a], procesamiento eficiente de único site [BBDM03], gestión de recursos [BDM04, SW04b], cómputo de estadísticas sobre streams [AM03], e incluye algunas operaciones distribuidas [HID03a, OJW03]. Además este sistema es capaz de procesar datos a medida que arriban sin tener que almacenarlos y soporta consultas continuas sobre streams y sobre relaciones almacenadas [ABW02]. Para conseguir esta última característica introduce tres tipos de operaciones: *streamtorelation*, *relationtorelation*, and *relationtostream*. Los operadores streamtorelation usan especificaciones de ventana para transformar streams en relaciones que evolucionan en el tiempo. Los operadores relationtorelation realizan la mayor parte de los cómputos sobre las relaciones de entrada y finalmente los operadores relationtostream transforman (de ser necesario) los resultados nuevamente en streams. Los usuarios expresan las consultas en CQL [ABW06, ABW02], un lenguaje basado en SQL99 con el agregado de tres operadores streamtorelation. La combinación de streams y relaciones posibilitan aplicaciones interesantes pero aumentan significativamente la complejidad del sistema; incluso simples consultas de filtrado se traducen en series de tres operadores: un streamtorelation, un relationtorelation y

uno relationtostream. Las capacidades distribuidas de STREAM se limitan al estudio de los trade offs entre utilización de recursos (especialmente ancho de banda) y precisión de los resultados que se obtienen al emplear fuentes de datos distribuidas.

El proyecto Aurora [BBC+04b] también propone un nuevo modelo de datos para el procesamiento de streams donde los usuarios expresan las consultas directamente mediante diagramas (mediante una GUI o una descripción textual) de cajas y flechas que describen la forma en la que se combinan y transforman los streams. En este modelo las cajas representan los operadores (operadores relacionales) y las flechas los streams. Este sistema adopta una interfase dataflow en lugar de una tradicional interfase declarativa en SQL para facilitar la integración de operadores de consulta predefinidos con los definidos por el usuario. Una desventaja de este modelo es que los clientes pueden solicitar cualquier stream intermedio en cualquier momento, potencialmente limitando la optimización de consultas. En términos de arquitectura, al igual que TelegraphCQ y STREAM, Aurora procesa datos a medida que arriban sin almacenarlos, por lo tanto potencialmente soporta rates de entrada más altos que los de los DBMSs tradicionales.

Gigascope [CJS03, CJSS03, CJK+04] es un DSMS designado específicamente para soportar aplicaciones de monitoreo de red que favorece la performance por sobre la flexibilidad. Los usuarios expresan las consultas en GSQL, un lenguaje basado en un subconjunto de SQL pero que incluye operadores definidos por el usuario. Estos operadores permiten la definición de funciones de monitoreo de red basándose en herramientas provenientes de este campo. Las consultas se compilan en módulos escritos en C o C++ y luego se enlazan (*linkean*) en tiempo de ejecución. Si bien esto logra una mejor performance en tiempo de ejecución dificulta el agregado o la eliminación de consultas en tiempo de ejecución. Este sistema tiene una arquitectura de dos niveles: consultas de bajo nivel que se ejecutan en el origen de los datos y que efectúan fuertes reducciones de datos (*e.g.,* agregación y selección) antes de enviar los datos hacia un nodo de consulta de alto nivel capaz de correr consultas más complejas sobre los streams más lentos. Gigascope puede ser considerado como sistema distribuido debido a que cada nodo de consulta también es un proceso.

2.2.2 Tolerancia a fallas en los DSMS

Los pocos trabajos de investigación que persigan alta disponibilidad en DSMS emplean en su mayoría técnicas enfocadas en fallas de tipo *fail-stop* sobre nodos de procesamiento [HBR+05, BHS09]. Estas técnicas no incluyen fallas en la red [HBR+05] o favorecen estrictamente la consistencia requiriendo que exista al menos una copia

del diagrama de consulta para garantizar procesamiento [BHS09]. A diferencia de estos trabajos anteriores nuestra propuesta apunta a manejar eficientemente diversos tipos de fallas en los nodos y en la red subyacente, dándole a las aplicaciones la capacidad de decidir (trade-off) entre disponibilidad y consistencia. Esta propuesta permite que las aplicaciones cliente puedan elegir recibir únicamente resultados correctos (consistencia) o pueden preferir recibir resultados tan pronto como sea posible y luego corregirlos (disponibilidad).

Hay sistemas DSMS que sin enfocarse en tolerancia a fallas pueden soportar ciertos desórdenes y/o demoras en los streams mediante el uso de puntuaciones [CGM10, TMS03, TMSF03], *heartbeats* [SW04a] o *slacks* estáticamente definidos [BBC+04b]. Un operador con un parámetro slack acepta un número predefinido de tuplas fuera de orden empleando un límite de ventana (antes de cerrar dicha ventana de cómputo), posteriores tuplas que arriban fuera de orden se descartan. Una puntuación es un predicado sobre streams que debe evaluarse a falso para cada elemento que siga a la puntuación [TMSF03]. Las puntuaciones son elementos particulares en los streams cuya función es la de desbloquear operadores como Aggregate y Join permitiéndoles procesar todas las tuplas que tengan la misma puntuación. Si las puntuaciones se demoran los operadores se bloquean. El Administrador de Entradas de STREAM [SW04a] utiliza heartbeats periódicos con valores monótonamente crecientes de timestamp para ordenar las tuplas a medida que entran en el DSMS. Sin embargo asume que los heartbeats siempre arriban dentro de un marco temporal predefinido. Si se produce una falla que causa una demora mayor a la esperada las tuplas que arriban tarde no serán incluidas en la ventana de cómputo.

Como puede observarse las tres metodologías anteriores soportan cierto desorden pero se bloquean o descartan tuplas cuando el desorden o la demora excede los límites esperados. El bloqueo reduce la disponibilidad y si bien continuar procesando aún sin contar con algunas tuplas (perdidas) ayuda a mantener la disponibilidad, sacrifica la consistencia sin siquiera informar a los nodos downstream o a las aplicaciones clientes acerca de las fallas que se están produciendo.

2.3 Sistemas workflow

Los sistemas de gestión de flujo (*Workfow Management Systems (WFMS)*) [AM97, ARM97, Hol95, ERS+95] presentan algunas características comunes con los sistemas de procesamiento de streams. En un proceso workflow los datos atraviesan pasos independientes de ejecución que en su conjunto logran el objetivo del sistema. Esta forma de procesamiento es similar al comportamiento de las tuplas a través de un

diagrama de consultas. Desde el punto de vista de la tolerancia a fallas la diferencia más importante entre un WFMS y un DSMS radica en que cada actividad del workflow empieza, procesa sus entradas, produce salidas y finaliza, a diferencia de un DSMS donde los operadores procesan tuplas de entrada, actualizan sus estados temporales y producen tuplas de salida continuamente. Además, los operadores en un diagrama de consulta presentan una granularidad más fina que las tareas en un workflow.

Para lograr alta disponibilidad la mayoría de los WFMSs emplean un servidor de almacenamiento centralizado y efectúan el *commit* de cada paso de ejecución a medida que se ejecuta [KAGM96]. Si ocurre una falla la información *cometida* es suficiente para realizar la recuperación. Los servidores de almacenamiento mismos emplean uno de los mecanismos tradicionales de alta disponibilidad: standby caliente, standby frío, 1-seguro y 2-seguros [KAGM96]. Para abortar un proceso deben deshacerse las operaciones ejecutadas previamente mediante operaciones compensatorias [MAA+95], por ejemplo devolver un asiento reservado para un determinado vuelo compensa la operación de reservar un asiento para ese vuelo. Este modelo de tolerancia a fallas no es adecuado para los DSMS porque no incluye el concepto de transacciones, no hay momentos durante la ejecución de una parte del diagrama de consultas. El estado es temporal y el procesamiento continuo y además almacenar persistentemente el estado de cada operador después de que procesa cada tupla o aún ventanas de tuplas sería prohibitivo.

En lugar de utilizar un servidor central de almacenamiento, existen soluciones más escalables en cuanto a tolerancia a fallas en WFMSs, *e.g.,* logueando mensajes persistentemente e intercambiando datos durante los pasos de ejecución [MAM+95, ARM97]. Es claro que la información transferida durante los pasos de ejecución puede ser grande, algunos WFMSs utilizan un administrador de datos independiente [ARM97]. El administrador de datos es un DBMS distribuido y replicado y por ende presenta las mismas propiedades que los mecanismos de replicación *eager* y *lazy* descriptos anteriormente en este capítulo. Es claro entonces que ni las colas persistentes ni un administrador de datos son aplicables a los DSMS, pues sería necesario almacenar las tuplas antes de enviarlas a los nodos downstream para su procesamiento. El objetivo principal de una arquitectura DSMS es procesar los datos tan pronto van arribando antes de almacenarlos (posiblemente incluso sin hacerlo) para no necesitar descartar datos de entrada.

Por último cabe mencionar que algunos WFMS soportan operaciones *desconectadas* bloqueando actividades antes de la desconexión [AGK+95]. Esta metodología

funciona adecuadamente para las desconexiones previstas, sin embargo nuestro objetivo en esta materia es manejar las desconexiones causadas por las fallas. Además, un DSMS procesa datos a alta velocidad y por ende cualquier desconexión en las entradas hará que el nodo se quede sin tuplas para procesar.

2.4 Sistemas de Publicación/Subscripción

Los sistemas de publicación/subscripción [EFGK03a, JS03, Str04] constituyen un paradigma de comunicación muchos-a-muchos que desacopla los eventos de fuentes y destinos. Los clientes registran los eventos o patrones de eventos que les interesan especificando un tópico (*e.g.,* [EFGK03b]), un tipo de evento, o un contenido determinado, o propiedades de algún evento (*e.g.,* [BCM+99, CRW00, HSB+02]). Cuando el nodo que publica genera un evento que concide con un evento de interés, el cliente recibe una notificación de dicho evento. El middleware de publicación/subscripción provee administración y almacenamiento de las subscripciones, acepta eventos de los nodos que publican y envía notificaciones a los nodos clientes. Este tipo de sistemas es entonces un sistema sin estado respecto del envío de mensajes. Un DSMS puede utilizarse como sistema de publicación/subscripción al transformar subscripciones en consultas continuas formadas exclusivamente por filtros.

Los sistemas más recientes de publicación/subscripción permiten subscripciones que especifiquen correlaciones entre eventos [BMB+00] o que tengan estado [JS03] y por ello en este sentido su similitud con los DSMS va en aumento.

Un aporte reciente a los sistemas de publicación/subscripción con estado es [Str04], el cual brinda tolerancia a fallas y desórdenes restringiendo el procesamiento a *transformaciones monótonas incrementales*. Esta técnica requiere que cada operador mantenga abiertas en todo momento todas las ventanas de cómputo para poder producir el rango actual para el valor final. Dicho rango se va achicando a medida que arriba nueva información, sin embargo sus límites pueden permanecer arbitrariamente grandes si así lo requiere el dominio del atributo en cuestión. Por lo tanto este mecanismo funciona únicamente para períodos cortos de fallas debido a que provoca que la cantidad de operadores crezca a medida que se prolongan las mismas.

2.5 Redes de sensores

Los DSMS tienen como objetivo principal atacar eficiente y confiablemente el problema del procesamiento de altos volúmenes de datos que arriban en forma de streams.

La mayoría de los trabajos en este campo ven el origen de los datos exactamente como tales: orígenes de datos externos al sistema y preconfigurados para introducir tuplas dentro del sistema ya sea periódicamente o cuando ocurren eventos. Para muchas aplicaciones, como las de monitoreo de red o las de servicios financieros esta perspectiva es apropiada.

Sin embargo, cuando los orígenes de los datos son sensores, la performance puede incrementarse significativamente mediante ajustes dinámicos en los sistemas de monitoreo y en el flujo de los datos medidos por los sensores. Las ganancias potenciales son especialmente considerables cuando los sensores se organizan en redes ad-hoc en lugar de estas conectados directamente a una infraestructura estructurada, *e.g.,* cableada. Por esta razón en los últimos años se publicaron diversos trabajos que persiguen el procesamiento de streams mediante redes de sensores (*sensor networks*) [MFHH05, MFHH03, MNG05, YG02, YG03a, ASSC02a].

La principal similitud entre sistemas que ejecutan consultas sobre redes de sensores, como TinyDB [MFHH05, MFHH03] y Cougar [YG02, YG03a], y los DSMS radica indudablemente en que ambos tipos de sistemas procesan streams de datos y asumen que sus entradas son ilimitadas y restringen el procesamiento a operadores no bloqueantes.

Algunas propiedades de las redes de sensores vuelven a este tipo de sistemas difíciles de aplicar al procesamiento de streams. La primera de estas propiedades se refiere a las baterías, fuente de energía típica en sensores, las cuales presentan un tiempo de vida corto (días). La segunda se refiere a los escasos recursos de procesador y memoria de los sensores, los cuales limitan el tipo de cómputo a realizar. La tercer propiedad está relacionada con el modele de comunicaciones de las redes de sensores, dado que la comunicación es inalámbrica, los sensores sólo pueden comunicarse directamente con sus vecinos cercanos; para comunicarse con sus vecinos más lejanos deben recurrir a comunicaciones multi-hop, *i.e.,* los sensores deben reenviar datos provenientes de otros sensores. Adicionalmente la comunicación en estos sistemas no es confiable y presenta bajo ancho de banda.

Las propiedades arriba expuestas dan forma a la investigación para el procesamiento de streams bajo redes de sensores, por ejemplo los lenguajes de consulta para redes de sensores permiten que las aplicaciones ajusten el trade-off entre precisión, latencia y utilización de recursos en la generación de resultados [MFHH05, YG03a], *e.g.,* una consulta puede especificar la velocidad de sampleo o su tiempo de vida en el sistema. En resumen el procesamiento de consultas y las optimizaciones deben obligatoriamente considerar todos los aspectos del procesamiento de streams: cuándo recolectar información, qué recolectar, qué transmitir y cuál es la mejor forma de

transmitir esa información, como agregar los datos a medida que viajan por la red, qué tipo de topología crear para lograr una buena transmisión de los datos, cómo tolerar fallas en las transmisiones, etc. Este método holístico es opuesto al procesamiento realizado en DSMS mediante potentes servidores, los cuales operan a un nivel de abstracción mucho mayor y por lo tanto permiten ignorar el problema de la recolección de datos a bajo nivel y los problemas de comunicación. En este trabajo nos enfocaremos exclusivamente en el procesamiento de streams en redes con dichos nodos potentes.

2.6 Replicación de máquinas de estados

Los DSMS mantienen un estado que se actualiza a medida que reciben entradas, por ello podemos ver a los DSMS replicados como una instancia de una máquina de estados (autómata finito) replicada [Sch90]. En forma similar a la operatoria de una máquina de estados nuestro sistema mantiene la consistencia entre réplicas asegurando que el procesamiento de los mensajes de entrada se efectúa en el mismo orden. Sin embargo, y en contraste con una máquina de estados típica, los mensajes son agregados (*aggregate*) en lugar de procesarlos atómicamente e individualmente uno de otro; por lo tanto no tenemos una relación directa entre mensajes de entrada y de salida. Más importante aún es que las técnicas tradicionales para tolerancia a fallas que emplean replicación de máquinas de estados favorecen estrictamente la consistencia por sobre la disponibilidad, solamente las réplicas que pertenecen a la partición de red mayoritaria continúan procesando en todo momento [CL02] y un *voter* (*e.g.,* el cliente de salida) combina la salida de todas las réplicas para generar el resultado final [Sch90]. A diferencia de este comportamiento, nuestro sistema favorece la disponibilidad, permitiendo que las réplicas continúen procesando aún en presencia de fallas en algunos de los streams de entrada.

La metodología de trabajo de las máquinas de estado es particularmente apropiada para la gestión de fallas Bizantinas, donde un nodo que falla produce resultados erróneos en lugar de dejar de funcionar. Nuestro sistema solamente es capaz de manejar las fallas del tipo *fail crash* en nodos y fallas en la red (enlaces y particiones).

2.7 Recuperación mediante rollback

Existen diversos trabajos que presentan la reconciliación de estado de un nodo de procesamiento mediante combinaciones de *checkpoints*, *undo* y *redo*, *e.g.,* [EAWJ02, GR92, LT03, TTP95]. Nuestro enfoque adapta estas técnicas en el contexto de la

tolerancia a fallas y la reconciliación de estado de un DSMS.

2.8 Gestión de carga

A continuación presentaremos un resumen de los trabajos relacionados en el campo de la gestión de carga en ambientes cooperativos y competitivos. También discutiremos brevemente esquemas de gestión de SLAs (*Service Level Agreements*).

2.8.1 Algoritmos de gestión de carga cooperativos

La mayor parte de los trabajos previos referidos a la gestión de carga en un DSMS se enfocan en los problemas del scheduling eficiente de operadores [BBDM03, CcR+03] y de *load-shedding* [BDM04, DGR03, SW04b, TcZ+03]. Existen pocos trabajos que apunten a la asignación de operadores a nodos de procesamiento minimizando la utilización de recursos (*e.g.,* ancho de banda de red) o la latencia de procesamiento en el contexto de los streams distribuidos [Ac04, PSW+04]. Los últimos trabajos en esta materia comienzan a enfocarse en las reasignaciones dinámicas de operadores entre nodos de procesamiento en respuesta a variaciones de carga [XHcZ06]. Estas técnicas mencionadas asumen un entorno colaborativo y optimizan calidad de servicio o utilidad global del sistema.

La gestión de carga cooperativa y el compartir recursos constituyen áreas de investigación profundamente estudiadas en el campo de los sistemas distribuidos, *e.g.,* [ELZ86, Gal77, KS89, MFGH88, SHCF03]. En este sentido las investigaciones que presentan mayor similitud con nuestra propuesta producen asignaciones cuasi optimales empleando un gradiente descendiente, en el que los nodos intercambian entre ellos carga o recursos generando sucesivamente asignaciones menos costosas.

En particular en [ELZ86] se comparan diferentes algoritmos de balance de carga y se presenta un resultado interesante: el algoritmo basado en un umbral, en el que los nodos con más de T tareas encoladas transfieren las nuevas tareas que van arribando hacia nodos con colas con menos de T elementos, si bien es uno de los más simples se comporta casi tan bien como los que transfieren carga a los nodos menos cargados. Sin embargo ninguno de estos algoritmos investigados se desempeñaron de forma óptima.

Nuestra metodología está basada en contratos y constituye una variante de las basadas en umbral adaptada a tareas continuas y entornos heterogéneos integrados por agentes egoístas. Además permitimos algunas variaciones en los límites de los umbrales, por ejemplo mediante contratos de precio acotado y asumimos que diferentes nodos emplean diferentes umbrales, por ejemplo nodos que tienen diferentes

contratos. Sin embargo y coincidiendo con Eager et al. [ELZ86], encontramos que en muchos casos aún los esquemas de umbral fijo producen buenas distribuciones de carga globales.

En contraste con los trabajos de gestión de carga bajo ambientes cooperativos, el principal desafío de nuestro esquema es su enfoque en ambientes competitivos. En los entornos cooperativos a menos que existan nodos maliciosos o que están experimentando fallas, los nodos siguen un algoritmo predefinido y revelan al resto de los nodos sus condiciones verdaderas de carga con el objetivo de optimizar la performance global del sistema. En un entorno competitivo los participantes se encuentran motivados por sus intereses propios y adoptan estrategias que optimizan sus propias utilidades. Incluso los participantes podrían elegir no colaborar en lo más mínimo con el resto de los mismos.

2.8.2 Diseño de mecanismo algorítmico distribuido

El incremento en la popularidad de los sistemas federados produjo reciente interés en nuevas tecnologías para balance de carga y asignación de recursos, las cuales empezaron a considerar a los participantes como agentes egoístas. La solución típica es el empleo de tecnologías provenientes de la microeconomía y de la teoría de juegos para crear los incentivos correctos para que los participantes egoístas actúen de forma tal de beneficiar a todo el sistema. Estos nuevos esquemas pueden agruparse en dos grandes áreas: *diseño de mecanismos* y *economías de cómputo*.

El objetivo del *diseño de mecanismos* (DM) [Jac01, LP08] es el de implementara una solución global a un problema de optimización distribuida, donde cada participante egoísta tiene cierta información privada que constituye un parámetro del problema de optimización global. Por ejemplo el costo de procesamiento de cada participantes es un parámetro del problema global de distribución de carga. Si bien los participantes son considerados egoístas también se los asume racionales: buscan maximizar su utilidad, la cual calculan como la diferencia entre los pagos que reciben para realizar algún computo y el costo de procesamiento en el que incurren. Los participantes pueden mentir acerca de su información privada si es que ello mejora su utilidad. El mecanismo define las *reglas del juego* y de esta forma impone las restricciones sobre las acciones que pueden realizar los participantes. Los mecanismos más estudiados son los de *revelación directa*, en los cuales se les solicita a los participantes que revelen su información privada directamente a una entidad centralizada que computa la asignación óptima y un vector de pagos de compensación. Los algoritmos que computan la asignación de carga y los pagos a los participantes

se diseñan con el objetivo de optimizar la utilidad del participante una vez que éste revela en forma confiable su información privada.

En contraste con el diseño mecanismos puro, el *diseño de mecanismo algorítmico* (DMA) [NPS03b, NR01] considera adicionalmente la complejidad computacional de las implementaciones de dichos mecanismos, usualmente a expensas de encontrar una solución óptima. El *diseño de mecanismo algorítmico distribuido* (DMAD) [FPSS05, FS02] se concentra en las implementaciones distribuidas de los mecanismos, debido a que en la práctica suele ser imposible implementar un optimizador central. Los trabajos previos relacionados con DMAD incluyen el ruteo basado en BGP [FPSS05] y los árboles multicast de costo compartido [FPS01]. Estos esquemas asumen que los participantes ejecutan correctamente el cómputo de los pagos. Por el contrario, nuestro mecanismo propuesto es una ejemplo de un esquema DMAD que no considera este tipo de asunciones debido a que está basado en contratos bilaterales.

2.8.3 Gestión de carga basada en modelos económicos

Diversos investigadores propusieron el uso de principios económicos y modelos de mercados para el desarrollo de sistemas distribuidos complejos [MD00]. Las economías de cómputo encontraron nicho de aplicación en áreas como las bases de datos distribuidas [SAL+96], aplicaciones concurrentes [WHH+92] y la computación grid [BAG00, BSG+01].

La mayoría de los estudios que se centran en economías de cómputo utilizan un modelo basado en precios [BSG+01, Chu01, FNSY96, San93, SAL+96, WHH+92, ACM04], donde los consumidores poseen diferentes precios para distintos niveles de preferencia de performance, se asignan según presupuesto (*budget*) y se paga a los proveedores de los recursos. Para establecer los precios de los recursos se emplean diversas técnicas [BSG+01], frecuentemente los nodos que proveen recursos efectúan subastas para determinar el precio y la asignación de sus recursos [Chu01, FNSY96, WHH+92]. Alternativamente los proveedores de recursos compiten por tareas [San93, SAL+96] o ajustan sus precio iterativamente hasta que la demanda alcanza a la oferta [FNSY96]. Estas metodologías que emplean economías de cómputo necesitan que los nodos participen en las subastas durante cada movimiento de carga, induciendo de esta forma un gran overhead. La variabilidad en la carga también puede provocar variaciones substanciales en los precios y por ende pueden ocurrir reasignaciones frecuentes [FNSY96]. Si el costo del procesamiento de

tareas agrupadas es diferente del costo acumulativo de tareas independientes entonces los subastas se tornan combinatoriales [NR07, Par02][3], complicando aún más el problema de la asignación.

Si las subastas son realizadas por agentes sobrecargados entonces los agentes con poca carga tienen la posibilidad de participar en más de una subasta simultáneamente, sin embargo estas situaciones vuelven complejo tanto el *clearance* del mercado como los mecanismos de intercambio [NPS03b]. Nuestro esquema evita estas complejidades limitando la variabilidad de los precios de los recursos en tiempo de ejecución y serializando las comunicaciones entre partners. En contraste con nuestro sistema,las economías de cómputo le dificultan a los participantes la posibilidades de ofrecer diferentes niveles de precios y servicios a distintos partners.

Como alternativa al empleo precios las economías de cómputo también pueden basarse en trueques [BSG+01, CFV03, FNSY96]. En particular SHARP [FCC+03] es una infraestructura que permite que los pares intercambien tickets de acceso a los recursos en forma segura sin necesidad de dictar políticas que definan de qué forman deben intercambiarse dichos recursos. Chun [CFV03] propone la utilización de una economía de cómputo basada en SHARP donde los pares descubren en tiempo de ejecución los recursos requeridos e intercambian tickets de recursos. Un ticket es en este contexto una solicitud débil sobre el uso de un recursos y puede ser rechazada, resultando en un valor nulo para quien tiene el recurso. Por el contrario, nuestro esquema de acuerdos entre pares no especifica ninguna cantidad fija de recursos y por lo tanto los pares pagan solamente por los recursos que efectivamente utilizan.

2.8.4 Recursos compartidos en sistemas peertopeer

En los sistemas peertopeer los participantes ofrecen sus recursos al resto de los participantes en forma gratuita, pues los esquemas promueven la colaboración mediante el uso de reputación [LFSC03], contabilidad (*accounting*) [VCCS03], auditorías [NWD03] o pruebas de estrategia [NPS03a] para la eliminación de los *free-riders*[4].

El desafío en el que se enfocan los sistemas peertopeer es que la mayoría de las interacciones involucran a "extraños", es poco frecuente que los mismos participantes interactúen entre ellos en múltiples ocasiones. Nuestro trabajo apunta a un entorno completamente diferente en el cual los participantes controlan con quienes interactúan y pueden producir relaciones entre ellos a largo término tratando de maximizar sus utilidades.

[3] En una subasta combinatorial se venden concurrentemente múltiples items y para cada postor cada subconjunto de estos items representa un valor diferente.

[4] Los free-riders son participantes que utilizan recursos sin ofrecer nada a cambio.

2.8.5 Service Level Agreements

Los *Service Level Agreements* (SLAs) constituyen acuerdos que se emplean para negociar interacciones entre participantes autónomos [Cor11, FV02, CT04, Rac11, KL03, LM02, GS05, Ver99, Wus02]. Trabajos recientes en este campo apuntan a solucionar el problema del monitoreo automático y cómo lograr que se cumplan los SLAs [KL03, GS05]. Estos esquemas permiten que los SLAs incluyan especificaciones de medidas que deben considerarse para el monitoreo, el tiempo y la forma de empleo para tomar las medidas y el partner con el que se interactuará.

El modelo de contrato que proponemos en este trabajo encaja con una infraestructura SLA de este tipo.

2.9 Resumen

En este capítulo presentamos antecedentes y trabajos relacionados con el tema de investigación abordado en esta tesis. Discutimos los DSMS y su motivación. Presentamos esquemas que posibilitan tolerancia a fallas en DBMSs tradicionales, DSMS y otros sistemas relacionados. Además presentamos un resumen de las tecnologías relacionadas con la gestión de carga en sistemas federados.

En el próximo capítulo nos enfocaremos en la arquitectura propuesta como base del trabajo de esta tesis.

Capítulo 3

Arquitectura

Índice

3.1. Modelo de streams de datos 43
3.2. Operadores 45
3.2.1. Operadores sin estado 45
3.2.2. Operadores con estado 47
3.2.3. Operadores persistentes 50
3.2.4. Diagramas de consulta 50
3.3. Arquitectura del sistema 51
3.4. Arquitectura del nodo 52
3.5. Data flow 54
3.6. Resumen 55

Architecture starts when you carefully put two bricks together. There it begins.

— Ludwig Mies van der Rohe

En este capítulo presentamos la arquitectura de *Federación*, un nuevo sistema de procesamiento distribuido de streams. Los nodos pueden encontrarse bajo una misma entidad o pueden organizarse como un sistema federado débilmente acoplado bajo el control de múltiples participantes autónomos.

El resto del capítulo se organiza de la siguiente forma: en la Sección 3.1 presentamos el modelo de streams de datos de Federación, en la Sección 3.2 discutimos sus operadores y diagramas de consulta. En la Sección 3.3 presentamos los principales componentes del sistema y su interfaz. Luego, en la Sección 3.4 discutimos la arquitectura de software de los nodos y por último en 3.5 describimos el flujo de datos durante el procesamiento de streams.

3.1 Modelo de streams de datos

El modelo de datos de Federación define un stream como una secuencia *append-only* de items de datos. Los items de datos están compuestos por valores denominados tuplas. Todas las tuplas del mismo stream tienen los mismos atributos, los cuales definen el tipo o esquema del stream. Por ejemplo podemos pensar en una aplicación que monitorea las condiciones ambientales dentro de un edificio mediante sensores que generan un stream de mediciones de temperatura. Un posible esquema para este stream es $(t.tiempo, t.ubicación, t.temp)$, donde $t.tiempo$ es un campo de tipo *timestamp* que indica el momento en el que fue tomada la medida, $t.ubicación$ es un *string* que indica la ubicación física del sensor y $t.temp$ es un entero que representa el valor de la temperatura.

Emplearemos la siguiente definición de tupla:

Definición 3.1 *Tupla*
Una *tupla* es un ítem de datos contenido en un stream y sigue el esquema $(ET, A_1, \ldots, A_m)$. Una tupla entonces presenta la siguiente forma: $(tiempo, a_1, \ldots, a_m)$, donde $tiempo$ es una estampilla de tiempo y $a_1, \ldots, a_m$ son valores de atributos. Cada tupla perteneciente a un mismo stream sigue idéntico esquema. ■

Las estampillas de tiempo se utilizarán únicamente en trabajos futuros para propósitos como calidad de servicio (QoS) y por lo tanto pueden considerarse como parte del *header* de la tupla, mientras que el resto de los atributos constituyen la parte de datos de la misma.

A este esquema básico se le adicionan dos campos: tipo de tupla (*tipo*) e identificador de tupla (*id*). El identificador de tupla sirve para la identificación unívoca de

una dada tupla dentro de un stream[1]. El campo tipo de tupla le permite al sistema distinguir entre los diversos tipos de tuplas, característica necesaria para el correcto funcionamiento de DCAA, nuestro protocolo de tolerancia a fallas. En el Capítulo 4 describiremos cómo DCAA extiende el encabezado de la tupla con una estampilla de tiempo adicional llamada *stime*, la cual se utiliza para ordenar las tuplas determinísticamente antes de procesarlas. DCAA se basa en la siguiente definición de tupla:

Definición 3.2 *Tupla (nueva)*
Una *tupla* es un ítem de datos contenido en un stream y sigue el esquema $(TIPO, ID, STIME, A_1, \ldots, A$
Una tupla entonces presenta la siguiente forma: $(tipo, id, stime, a_1, \ldots, a_m)$, donde *tipo* es el tipo de la tupla, *id* identifica unívocamente a la tupla en el stream, *stime* es una estampilla de tiempo y $a_1, \ldots, a_m$ son valores de atributos. Cada tupla perteneciente a un mismo stream sigue idéntico esquema. ■

Además utilizaremos las siguientes definiciones:

Definición 3.3 *Stream*
Un *stream* de datos es una secuencia *append-only* de tuplas que siguen el mismo esquema predefinido. ■

Un stream de datos usualmente se produce en un único origen de datos, sin embargo nuestro sistema no fuerza esta condición. Un origen de datos puede producir múltiples streams que sigan el mismo esquema, pero cada uno de ellos debe tener diferente nombre.

Definición 3.4 *Origen de datos*
Un *origen de datos* es cualquier aplicación, dispositivo u operador que produce tuplas continuamente y las inyecta en las aplicaciones cliente (modelo *push*) o en otros operadores en caso de ser necesario procesamiento adicional. ■

Al presentar DCAA en el próximo capítulo, necesariamente deberemos referirnos a secuencias de tuplas en un stream y para ello emplearemos las siguientes definiciones:

Definición 3.5 *Prefijo de una secuencia de tuplas*
Un *prefijo de una secuencia de tuplas* comienza con la tupla más antigua de la secuencia y se extiende hasta alguna tupla t dentro de la misma. ■

[1]Las estampillas de tiempo no pueden emplearse como identificadores dado que no son únicas.

Definición 3.6 *Sufijo de una secuencia de tuplas*
Un *sufijo de una secuencia de tuplas* comienza con alguna tupla t de la secuencia y se extiende hasta la tupla más reciente dentro de la misma. ■

Usualmente consideramos al stream entero como la secuencia de tuplas y hablamos de prefijo y sufijo del stream para referirnos a todos las tuplas que preceden o suceden a una dada tupla dentro del mismo. Adicionalmente emplearemos los términos prefijo y sufijo aplicado a tuplas *tentativas*. En este caso consideraremos a las tuplas tentativas como tuplas que se encuentran aisladas del resto del sistema.

3.2 Operadores

El objetivo de un sistema de procesamiento de streams es poder filtrar, correlacionar, agregar y transformar streams de entrada para producir salidas de interés a las aplicaciones cliente, facilitando incluso el desarrollo de las mismas. Por ejemplo, un DSMS podría disparar una alarma cuando la combinación de temperatura y humedad dentro de una habitación supera un dado umbral de confort predeterminado. La aplicación podría transformar la alarma en un email y notificar a quien corresponda.

Dentro de un DSMS los streams de entrada se transforman en streams de salida cuando atraviesan una serie de "cajas" denominadas *operadores*, los cuales se describen a continuación.

3.2.1 Operadores sin estado

Los *operadores sin estado* procesan de una tupla por vez sin mantener estado entre tupla y tupla. Federación posee tres operadores sin estado: *Filter*, *Map* y *Union*.

Filter es el equivalente al operador relacional de selección y su función es la de aplicar un predicado a cada tupla de entrada de forma tal que aquellas que satisfacen el mismo son enviadas al stream de salida. Por ejemplo, un filtro aplicado a un stream de mediciones de temperatura podría devolver las temperaturas que superen cierto umbral, por ejemplo $temperatura > 32°C$. Las tuplas que no satisfacen el predicado son o bien descartadas o enviadas a otro stream de salida. Un *Filter* puede tener múltiples predicados y en este caso entonces actúa como una sentencia `case`, propagando cada tupla al stream de salida que se corresponda con el primer predicado que se cumpla.

El operador *Map* transforma tuplas de entrada en tuplas de salida aplicando un conjunto de funciones a los atributos de las tuplas. Por ejemplo map puede transformar el stream de mediciones de temperatura expresado en grados celsius

en un stream de grados fahrenheit. Un ejemplo más complejo es el siguiente: dadas tuplas de entrada con dos atributos, e (espacio) y t (tiempo), map podría producir tuplas de salida con un único atributo v que indique velocidad, calculado como $v = e/t$.

Por último, el operador *Union* simplemente mezcla (*Merge*) un conjunto de streams de entrada (todos con el mismo esquema) en un único stream de salida. Este operador mezcla las tuplas según su orden de arribo sin forzar ningún ordenamiento en las tuplas de salida. Luego, las tuplas de salida pueden ordenarse con algún operador *Sort*, específico a este fin, para ello mantendrá un buffer parametrizable de tamaño $n + 1$ y cada vez que llegue una nueva tupla este operador de ordenamiento entregará la tupla de menor valor del buffer.

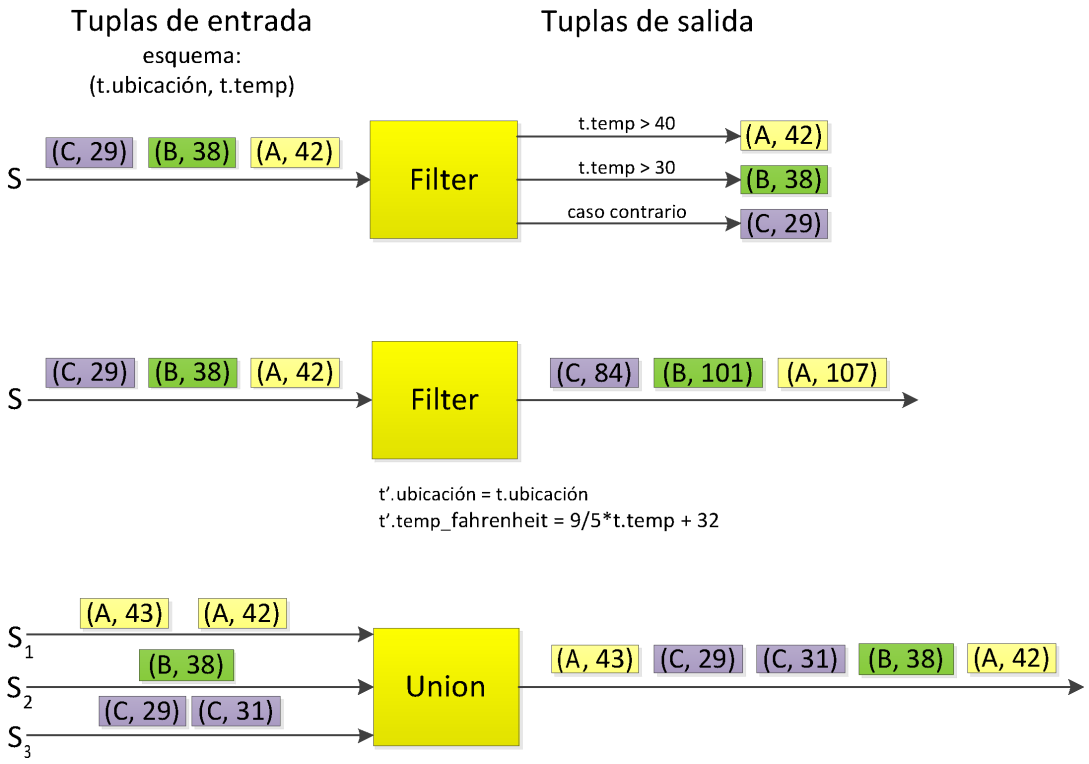


FIGURA 3.1: Ejemplo de salidas de operadores sin estado

La Figura 3.1 muestra ejemplos de la ejecución de los operadores *Filter*, *Union* y *Map* aplicados a un conjunto de tuplas de entrada. En el ejemplo, las tuplas de entrada tienen dos atributos, un identificador de habitación indicado mediante una letra y un entero que indica la lectura de la temperatura. El operador *Filter* tiene dos predicados más un stream de salida para las tuplas que coinciden con alguno de estos predicados; el operador *Map* convierte temperaturas celsius en fahrenheit. Por último, *Union* mezcla las tuplas según su orden de arribo.

3.2.2 Operadores con estado

En lugar de procesar cada tupla en forma aislada del resto, los operadores con estado realizan cómputos sobre un conjunto de tuplas de entrada. Esta primera versión de Federación presenta dos únicos operadores con estado: *Join* y *Aggregate*.

Un operador *Aggregate* computa una función agregada, como por ejemplo promedio, máximo, media, etc. Esta función se computa sobre los valores de un atributo de las tuplas de entrada. Por ejemplo podría producir la temperatura media de un stream de mediciones de temperatura. Antes de aplicar esta función el operador de agregación puede opcionalmente particionar el stream de entrada empleando valores para uno o más atributos, por ejemplo puedo devolver la temperatura promedio de cada habitación. La versión relacional de Aggregate es usualmente bloqueante: el operador puede tener que esperar todos sus datos de entrada antes de producir un resultado. Claramente esta metodología no es apropiada para streams de entrada no acotados, pues esta operación aplicada al procesamiento de streams efectúa cómputo sobre ventanas de datos que avanzan (se "deslizan") a medida que transcurre el tiempo, por ejemplo producen la temperatura promedio cada minuto. Estas ventanas se definen a partir de los valores de algún atributo de las tuplas de entrada, como el momento (tiempo) en el que se midió la temperatura. El tamaño de ventana y que tanto se desliza la misma constituyen parámetros parametrizables y el operador no registra la evolución (historia) de una ventana hacia la próxima. Para poner una ejemplo imaginemos un operador de agregación que calcula la temperatura promedio en una habitación mediante el procesamiento de tuplas de entrada que presentan dos atributos: el momento de la medición y el valor de la temperatura: $(7:06, 20), (7:11, 22), (7:15, 23), (7:19, 21), (7:25, 20), (7:31, 19), \ldots$

El operador de agregación podría realizar este cómputo de diferentes maneras a partir de distintas especificaciones de ventanas. Por ejemplo podría utilizar una ventana *landmark* [CCD+03], manteniendo la temperatura promedio a partir del valor landmark y actualizando el promedio con cada nueva tupla de entrada. Si asumimos que el landmark es 7 : 00, el stream de salida sería el siguiente: (7 : 06, 20), (7 : 11, 21), (7 : 15, 21,67), (7 : 19, 21,5), (7 : 25, 21,2), (7 : 31, 20,83), . . .

Alternativamente, este operador podría emplear una ventana deslizante. Asumiendo una ventana de 10 minutos de duración que avanza de a 10 minutos, el *Aggregate* haría el cómputo de los promedios para las ventanas [7 : 06, 7 : 16), [7 : 16, 7 : 26), . . . generando entonces la siguiente salida: (7 : 06, 21,67), (7 : 16, 21,35), ...

En este ejemplo, el *Aggregate* utiliza el valor de la primer tupla (7 : 06) para asignar los límites de la ventana. Si el operador hubiese comenzado desde la tupla

$(7:19, 21)$, las ventanas hubiesen sido $[7:19, 7:29), [7:29, 7:39), \ldots$. También es posible independizarse del valor de las tuplas y comenzar desde el múltiplo más cercano a 10 minutos (valor de avance), efectuando los cómputos promedios para las ventanas $[7:00, 7:10), [7:10, 7:20), [7:20, 7:30), \ldots$ produciendo la siguiente salida: $(7:00, 20), (7:10, 22), (7:20, 20), \ldots$.

Por último podemos procesar la entrada directamente a partir de un número fijo de tuplas de entrada, para este ejemplo podríamos calcular la temperatura promedio cada 30 mediciones.

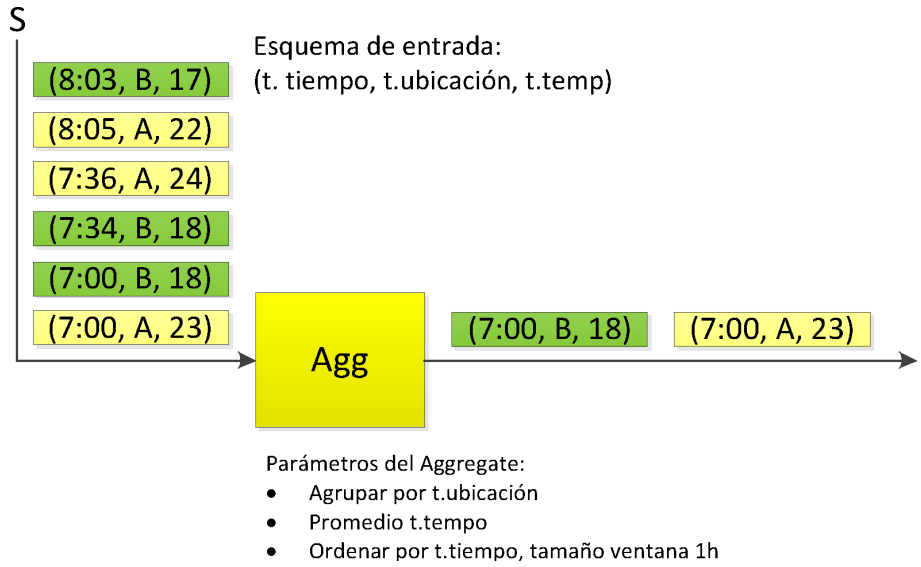


FIGURA 3.2: Ejemplo de salida de un operador Aggregate

Generalmente las especificaciones de ventanas hacen que los operadores con estado se vuelvan sensibles al orden de arribo de las tuplas de entrada. Cada operador asume que las tuplas arriban ordenadas según el atributo empleado en su especificación de ventana, por lo tanto el orden afecta el estado y la salida de dicho operador. Por ejemplo cuando un operador con una ventana de tiempo deslizante de 10 minutos computa la temperatura promedio para las $7:10$ y recibe una tupla con la medición de las $7:21$ entonces el operador cierra la ventana, calcula la temperatura promedio para las $7:10$, produce una tupla de salida y avanza la ventana hasta el próximo intervalo de 10 minutos. Si una tupla con la medición de las $7:09$ arriba luego de que se cerró la ventana, entonces dicha tupla es descartada. Por lo tanto, aplicar un operador de agregación a la salida de un operador *Union* produce resultados aproximados, pues la unión no ordena las tuplas mientras combina sus streams de entrada. La Figura 3.2 ilustra un cómputo *Aggregate* sencillo, donde las tuplas poseen tres atributos: el tiempo de la medición, la ubicación (e.g. letra que identifique una habitación), y un valor de temperatura. La agregación produce cada

hora la temperatura promedio para cada habitación.

Join es otro operador con estado, el cual recibe dos streams de entrada y para cada par de tuplas de entrada (una de cada stream), este operador aplica un predicado a los atributos de las mismas. Si se satisface el predicado entonces el *Join* concatena las dos tuplas y redirige la tupla resultante al stream de salida. Por ejemplo un operador *Join* podría concatenar una tupla que almacena una lectura de temperatura con una que contiene una lectura de humedad cada vez que las dos lecturas provienen de un mismo lugar (ubicación). El operador relacional *Join* acumula estado, el cual aumenta linealmente con el tamaño de sus entrada, haciendo el *matching* para cada tupla de entrada perteneciente a una relación con cada tupla de entrada de la otra. Asumiendo que los streams no tienen cota superior entonces es claro que no es posible acumular estado continuamente. Por el contrario, este operador efectúa el matching únicamente en tuplas que pertenecen a la misma ventana. Sean R y S dos streams de entrada, ambos con atributos hora y tamaño de ventana v, el operador *Join* hace matching con tuplas que satisfacen $|r.hora - s.hora| \leq v$[2]. La Figura 3.3 ilustra una operación simple de *Join* donde las tuplas del stream S presentan tres atributos: el tiempo de la medición, la ubicación (e.g. letra que identifique una habitación), y un valor de temperatura; las tuplas del stream R presentan un valor de humedad en lugar del de temperatura, el resto de los atributos son equivalentes a los de S. El *Join* combina las medidas de temperatura con las de humedad de una misma habitación cuando dichos valores se encuentran a menos de una hora de diferencia.

En resumen, los operadores de estado de *Federación* efectúan sus cómputos empleando ventanas de datos. Dado que los operadores no mantienen historia entre una ventana y su antecesora, en cualquier instante de tiempo el estado de un operador esta constituido por los límites de la ventana actual y el conjunto de tuplas pertenecientes a dicha ventana. Los operadores pueden mantener su estado en forma agregada, por ejemplo, el estado de un operador que calcula promedios puede reducirse a una suma y a un número de tuplas.

Los operadores con estado pueden tener parámetros opcionales (usualmente denominados *slack*), forzándolos a esperar tuplas adicionales antes de cerrar una ventana. Este tipo de parámetros hace posible que lo operadores soporten desorden acotado en sus streams de entrada. Adicionalmente los operadores pueden tener un parámetro *timeout*, el cual le permite a un operador producir un valor y avanzar su ventana aún sin que hayan arribado nuevas tuplas. Los timeouts utilizan la hora

[2]Alternativamente podría utilizarse otra especificación de ventana

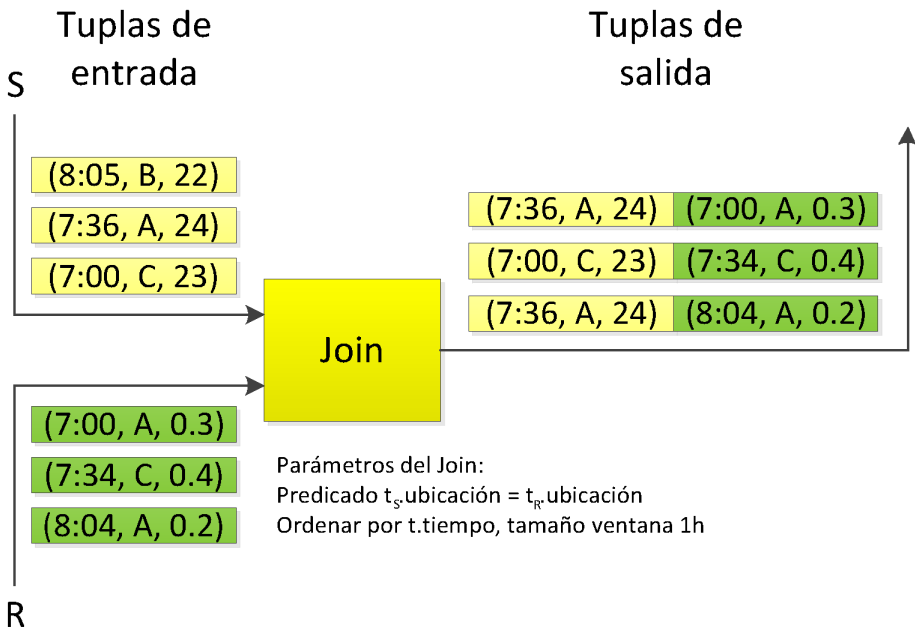


FIGURA 3.3: Ejemplo de salida de un operador Join

local del nodo de procesamiento y se inician cuando el operador abre una ventana de cómputo; si el timer local expira antes de que se cierre la ventana entonces produce un valor.

3.2.3 Operadores persistentes

La mayoría de los operadores de *Federación* efectúan sus cómputos únicamente sobre estado temporal, sin embargo existen dos operadores que tienen acceso a almacenamiento estable: Read y Update. Las tuplas de entrada para estos dos operadores son consultas SQL que leen y actualizan respectivamente el estado del DBMS y la salida de dichas operaciones SQL son enviadas como stream de salida.

3.2.4 Diagramas de consulta

La lógica de las aplicaciones en *Federación* sigue la forma de un *data flow* y las consultas se expresan en función de streams mediante diagramas de consultas. Dichos diagramas, compuestos por cajas y flechas, facilitan el proceso de sintonía de las aplicaciones. La Figura 3.4 reproduce el diagrama de consulta del ejemplo del Capítulo 1.

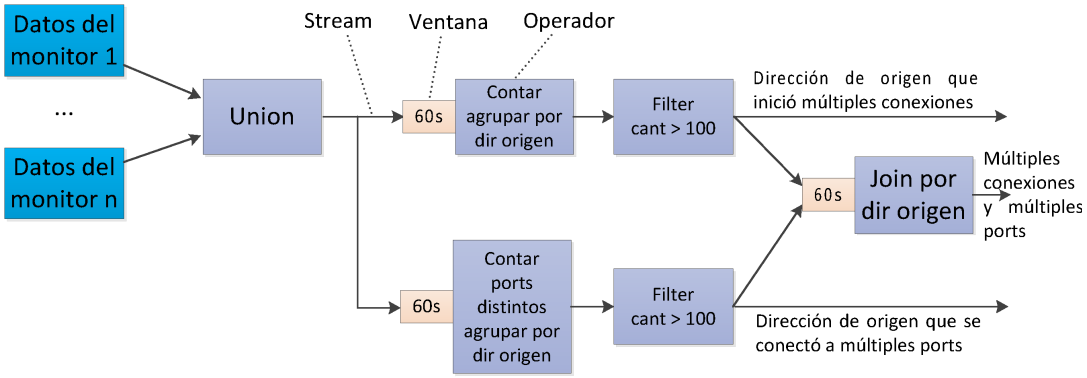


Figura 3.4: Ejemplo de un diagrama de consulta para la aplicación de monitoreo de red

3.3 Arquitectura del sistema

Federación es un DSS distribuido compuesto por múltiples máquinas físicas denominadas nodos de procesamiento (o simplemente nodos) y un catálogo global que presenta las siguientes características de diseño:

- El *catálogo global mantiene información acerca de todos los componentes del sistema*, incluyendo el conjunto de nodos de procesamiento, el diagrama de consulta, la asignación actual de operadores a nodos (*i.e.*, el diagrama actualmente utilizado) y otra información de configuración para los nodos de procesamiento. El catálogo global es una entidad lógica única que si bien puede implementarse como un proceso centralizado también es posible su implementación distribuida. Las aplicaciones cliente se comunican con el catálogo global para cambiar la configuración del sistema, crear o modificar el diagrama de consulta y subscribirse a streams. La subscripción a streams permite a los clientes recibir las tuplas producidas en dicho stream. Los métodos del catálogo deberían esperar argumentos expresados en XML; dicha elección no es mandatoria pero permite a los clientes efectuar descripciones textuales de sus requerimientos y facilita el desarrollo de las aplicaciones (diagrama de consulta y especificación de deployment[3]).

- Los *nodos son quienes realizan el procesamiento de los streams*. Cada nodo ejecuta un fragmento del diagrama de consulta y almacena información acerca de su fragmento en un catálogo que mantiene en forma local. Dicho catálogo local también almacena información acerca de otros nodos, aquellos que o bien envían datos a los streams de entrada del nodo en cuestión o que reciben datos producidos localmente mediante los streams de salida. Los nodos

[3]Descripción en XML que explicita qué grupo de operadores deben correrse en cada nodo.

también recolectan estadísticas sobre sus estados de carga y performance de procesamiento (*e.g.,* latencia de procesamiento). Cada nodo también realiza las tareas necesarias para manejar su carga y asegurar un procesamiento de streams tolerante a fallas. En la Sección 3.4 presentaremos la arquitectura de *Federación*.

- *Aplicaciones cliente*: los desarrolladores crean nuevas aplicaciones capaces de modificar el diagrama de consulta, asignar operadores a nodos de procesamiento. Adicionalmente una aplicación cliente puede actuar como origen de datos o destino de datos, produciendo o consumiendo streams.

- *Orígenes de datos*: los orígenes de datos son aplicaciones cliente que producen streams de datos que luego envían a los nodos de procesamiento.

3.4 Arquitectura del nodo

Cada nodo de procesamiento corre un servidor *Federación*, cuyos componentes de software se muestran en la Figura 3.5. A continuación se describen brevemente dichos componentes.

El *Procesador de Consultas* (PC) constituye la pieza principal donde tiene lugar el procesamiento de streams y está compuesto por:

- Una *interfaz administrativa* encargada de recibir todos los pedidos entrantes. Estos pedidos pueden modificar el fragmento de diagrama de consulta que se encuentra actualmente corriendo o solicitar al PC que traslade algunos operadores a PCs remotos. Los pedidos pueden adicionalmente incluir subscripciones a streams producidos localmente.

- Un *Catálogo Local* que mantiene la información acerca del fragmento actual del diagrama de consulta local. Dicha información incluye operadores locales, streams y subscripciones.

- Los *streams de entrada* alimentan al PC y los resultados se obtienen a partir del *Data Path*, quien rutea las tuplas entre los nodos de procesamiento y los clientes.

- Un *nodo de procesamiento* encargado del procesamiento local del streams de datos. Recibe los streams de entrada provenientes del Data Path, los procesa y produce streams de salida que vuelven luego al Data Path. Para realizar este procesamiento el nodo Federación instancia operadores y planifica su ejecución.

Cada operador recibe tuplas mediante sus colas de entrada (una por cada stream de entrada) y produce resultados en sus colas de salida. El nodo además produce estadísticas acerca de tiempos de ejecución (performance), frecuencia de datos de entrada (*data rates*), utilización de CPU de los operadores, etc. Dichas estadísticas pueden ser accedidas por los otros módulos mediante la *interfaz administrativa*.

- Un *Gestor de Consistencia* encargado de hacer que el procesamiento de streams sea tolerante a fallas. Describiremos su implementación en el próximo capítulo.

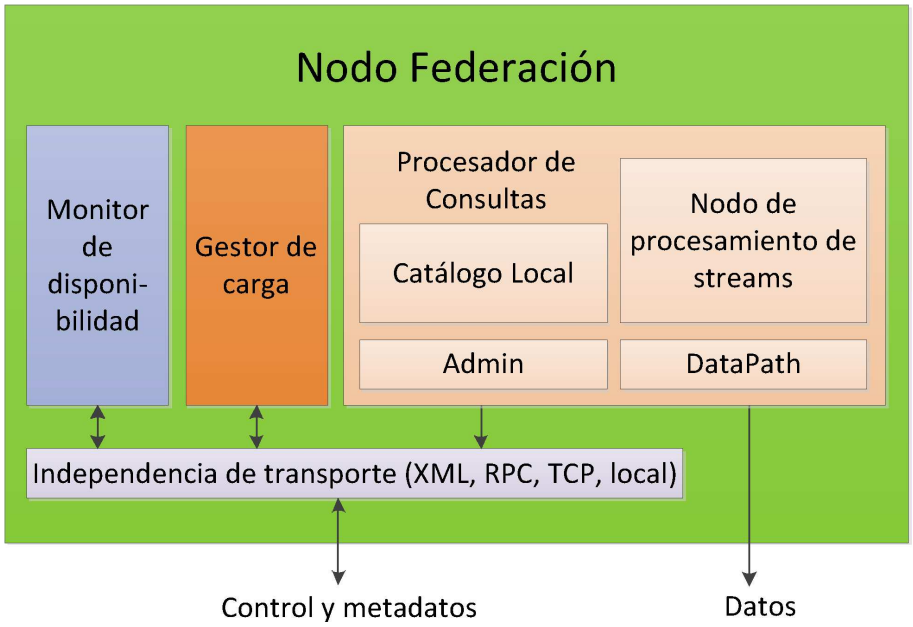


Figura 3.5: Arquitectura de software de un nodo Federación

Además del PC, la arquitectura del nodo Federación incluye módulos que le permiten comunicarse con sus pares en otros nodos y de esta forma poder llevar a cabo acciones colaborativas:

- El *Monitor de Disponibilidad* revisa el estado del resto de los nodos Federación y notifica al Procesador de Consultas cualquier cambio de estado. El Monitor de Disponibilidad es un componente genérico de monitoreo que se utiliza como parte de nuestro protocolo de tolerancia a fallas. Los detalles de este componente pueden encontrarse en el Capítulo 4.

- El *Gestor de Carga* utiliza información de carga local e información de carga proveniente de otros Gestores de Carga para mejorar el balance de la misma entre nodos. El Capítulo 5 desarrolla este tema en profundidad al concentrarse en los mecanismos de gestión de carga.

Los mensajes de control entre los componentes y entre los nodos viajan mediante una capa de transporte que podría implementarse en forma sencilla mediante RPCs (*Remote Procedure Calls*), unificando la implementación de la comunicación local y la comunicación remota. De esta forma las llamadas se traducen automáticamente en mensajes locales o remotos.

3.5 Data flow

Bajo *Federación* las aplicaciones pueden modificar el diagrama de consulta en tiempo de ejecución y pueden solicitar al sistema que se muevan operadores de un nodo hacia otro. Sin embargo, en este trabajo asumiremos que el diagrama de consulta desplegado sobre el conjunto de nodos de procesamiento es estático. Ignoraremos también los pasos involucrados en el despliega y la modificación del diagrama de consulta y describiremos únicamente las interacciones entre los componentes durante el procesamiento de streams.

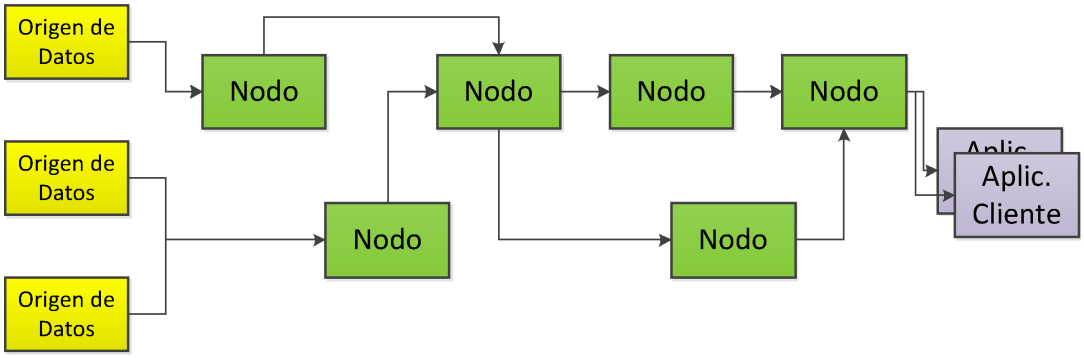


FIGURA 3.6: Data flow en un sistema Federación

El data flow es un flujo continuo de tuplas que provienen del origen de los datos hacia las aplicaciones cliente, pasando a través de los operadores que componen el diagrama de consulta. Por lo tanto, los componentes involucrados en el data flow son: los orígenes de los datos, los nodos de procesamiento y las aplicaciones cliente que se subscriben a los streams de salida. La Figura 3.6 ilustra el data flow, (1) representa el origen de los datos, productor de streams, (2) son los nodos de procesamiento, responsables de transformar los streams de entrada en streams resultado, los cuales son a su vez enviados a otros notros para que continúen el procesamiento, o (3) las aplicaciones cliente. Este flujo de datos de un nodo a otro define lo que en este trabajo denominaremos nodos vecinos, específicamente distinguiremos nodos *upstream* y *downstream*. A continuación precisaremos este concepto:

Definición 3.7 *Vecinos upstream/downstream*

Si un stream proveniente de un nodo de procesamiento N_u es enviado (*push*) hacia un nodo N_d para que continúe con su procesamiento, entonces se dice que N_u es un *vecino upstream* de N_d y que N_d es un *vecino downstream* de N_u . ■

Las aplicaciones que producen streams de entrada deben abrir conexiones (posiblemente TCP [APB09]) hacia los nodos de procesamiento *Federación* correspondientes y enviar los datos. Análogamente, las que reciben streams de salida deben "escuchar" en un determinado port conexiones entrantes y procesar los datos de entrada. Para facilitar la implementación de las comunicaciones y lograr un sistema mantenible es necesario emplear una librería. Si los nodos de procesamiento tuviesen suficiente poder de cómputo sería ideal con una implementación mediante Web Services [ACKM04, WCL+05].

Cuando se crea un diagrama de consulta, cada nodo que corre un fragmento de dicho diagrama recibe, desde el *Catálogo global*, información acerca de la ubicación de sus streams de entrada. Para cada stream se incluye información de identificación (*socket*, *i.e.,* dirección y puerto) del nodo de procesamiento que produce el stream de datos o que recibe a partir de un origen de datos. A partir de esta información el nodo envía un pedido de subscripción a cada nodo que produce cada uno de sus streams de entrada. Al recibir el pedido de subscripción, el nodo upstream abre una conexión hacia el nuevo vecino downstream y envía tuplas tan pronto van arribando. Claramente entonces Federación utiliza un *modelo push*, evitando que los nodos downstream tengan que hacer un *poll* continuo para verificar la disponibilidad de los datos. Adicionalmente cada nodo bufferea las tuplas de salida más recientes que produce para lograr la tolerancia a fallas que se discute en el Capítulo 4.

3.6 Resumen

En este capítulo presentamos la arquitectura de alto nivel de *Federación* para situar el contexto del trabajo. Presentamos el modelo de stream de datos, los operadores, los principales componentes del sistema y el flujo de datos que se produce durante el procesamiento de streams. En los siguientes capítulos presentaremos nuestros mecanismos de tolerancia a fallas y gestión de carga.

Capítulo 4

Tolerancia a Fallas

Índice

4.1. Definición de Falta, Error, Falla 57
4.2. Clasificación de Fallas 58
4.3. DCAA 60
4.4. Definición del problema 61
4.5. Generalidades sobre DCAA 74
4.6. Arquitectura de software extendida 76
4.7. Modelo de datos mejorado 78
4.8. Estado ESTABLE 79
4.9. Estado FALLA_UPSTREAM 87
4.10. Estado ESTABILIZACIÓN 91
4.11. Recuperación del nodo fallado 97
4.12. Gestión del buffer 98
4.13. Aplicaciones cliente y los orígenes de datos 104
4.14. Propiedades de DCAA 106
4.15. Resumen 117

By failing to prepare, you are preparing to fail.

— Benjamin Franklin

La ocurrencia de fallas, especialmente en un sistema distribuido, es habitual. Las fallas más comunes son las relacionadas con los nodos y la red. Una recuperación rápida y completa es crucial para el correcto funcionamiento y disponibilidad del sistema.

En este capítulo presentamos DCAA, un protocolo de tolerancia a fallas para el procesamiento de streams que permite que un DSMS distribuido pueda hacer frente a las fallas, tanto en la red como en los nodos. Este protocolo logra ajustarse a las aplicaciones, las cuales tienen diferentes requerimientos de disponibilidad y consistencia. DCAA minimiza el número de tuplas tentativas mientras garantiza que los resultados correspondientes a cualquier tupla se envíen downstream dentro de un tiempo preestablecido y que las aplicaciones eventualmente reciban los streams de salida en forma completa y correcta.

4.1 Definición de Falta, Error, Falla

El uso de términos como seguridad de funcionamiento, tratamiento de errores, faltas, fallas y averías se presta a confusión, ya que no es igual para todos los autores y es que no existe una terminología en español de los diferentes conceptos asociados a la tolerancia de fallas.

Si bien en [AL81] los autores definen las nociones de falta, error y falla, en este trabajo adaptamos las definiciones dadas por Laprie y Deswarte en [CLL78] y [Desw90] respectivamente. Estos autores consideran la falla de un sistema como la falta a la especificación de servicio pedido. Se define especificación como una descripción autorizada del servicio esperado. Esta especificación debe permitir que toda investigación sobre la causa de una falla concierna solamente a la operación interna del sistema. El fallo ocurre cuando el sistema tiene un comportamiento erróneo.

Avizienis define tolerancia a fallas en su paper *The N Version Approach to Fault Tolerant Software* como: "... preservar la entrega de los servicios esperados sin importar la presencia de errores causados por fallas en el sistema. Los errores son detectados y corregidos, y las fallas permanentes son ubicadas y removidas mientras el sistema sigue entregando servicio aceptable".

El comportamiento de un sistema es la función que realiza, y lo que le permite llevar a cabo dicha función es su estructura. Una estructura esta dividida en estados, donde un estado se define como la razón de ser de acuerdo a un un conjunto de circunstancias. Por lo tanto, para poder encontrar la causa de una falta debemos definir dos conceptos: *transición errónea* y *estado erróneo*.

Una transición errónea de un sistema es una transición de un estado interno del sistema a otro, que más tarde puede provocar una falta. Si la transición no es errónea, entonces es válida. Un estado erróneo es un estado interno que después de una secuencia de transiciones válidas, puede provocar una falta. Si un estado interno

no es erróneo, entonces es válido.

Un error es la parte del estado del sistema (con relación a los procesos de tratamiento) que es susceptible de ocasionar un fallo. La causa supuesta del error es una falta. Un error es entonces la manifestación de una falta en el sistema, y una falla es entonces el efecto de un error sobre el servicio.

Una falta se dice activa cuando produce un error. Una falta es, ya sea una falta interna que estaba precedentemente latente (*i.e.,* no producía error), o que ha sido activada por el proceso de tratamiento, *i.e.,* una falta externa. Una falta interna puede pasar del estado latente al estado activo.

Un error es por naturaleza temporal. Puede estar latente o detectado: un error es latente mientras no haya sido reconocido como tal; es detectado ya sea por los mecanismos de detección de error que analizan el estado del sistema, o por el efecto del error sobre el servicio. Generalmente, un error propaga otros (nuevos) errores, dentro de otras partes del sistema.

Si un sistema es considerado como un conjunto de componentes, la consecuencia del fallo de un componente es una falta interna por el sistema que lo contiene, y también una falta externa por el o los componentes que interactúan con él.

En vista de comprender mejor la terminología tomemos el ejemplo siguiente: un programador que se equivoca tiene un fallo seguido a un error de razonamiento; la consecuencia es una falta latente en el software escrito. Cuando esta falta sea sensibilizada, se volverá activa. La falta activa produce uno o más errores en los datos tratados. Cuando estos errores afectan el servicio liberado, entonces se dice que el sistema falló. Por otro lado, una avería se define como la detención de funcionamiento accidental y momentáneo.

4.2 Clasificación de Fallas

En [Stro84] se clasifican las fallas de la siguiente manera:

- fallas francas;
- fallas transitorias;
- fallas temporales;
- fallas bizantinas

- **Las fallas francas**

Las fallas francas, (*fail-stop* o *crash* en inglés), pueden presentarse en cualquier momento. En el instante mismo en el que un nodo tiene una falla, detiene definitivamente todas las operaciones que estaba haciendo, (a menos que sea reparado y reiniciado). Desde el punto de vista de vías de comunicación, la red no seguirá siendo considerada como operacional si es incapaz de transmitir un mensaje. Desde el punto de vista del proceso, este interrumpe su trabajo y cesa de producir mensajes.

Ejemplos de este tipo de falla son la falla de un proceso, un interbloqueo (*deadlock*) en el sistema, una ruptura en el cable de red, o la falla de la placa de red (NIC).

- **Las fallas transitorias**

En este tipo de falla el proceso puede producir un resultado falso pero continúa trabajando normalmente; un proceso puede perder un mensaje pero libera el resto de los mensajes que pasan por él. En este tipo de falla los resultados no son modificados, responden a la especificación pero pueden estar ausentes.

Encontramos como ejemplos de fallas transitorias a la pérdida de un mensaje, o la eliminación de un proceso para evitar un interbloqueo. Este tipo de falla es conocida también con el nombre de *falla intermitente* o *falla de omisión*.

- **Las fallas temporales**

Las fallas temporales, también llamadas *fallas de timing* o *fallas de desempeño*, se encuentran ligadas a la noción de tiempo y descansan sobre la existencia de una especificación precisa de comportamientos temporales. El mensaje llega antes o después del momento previsto.

El retardo de un mensaje, un reloj muy rápido, o la sobrecarga de un proceso que ocasiona un adelanto de un timeout, son ejemplos de fallas temporales.

- **Las fallas bizantinas**

En este grupo están reunidas todas las fallas no consideradas en las tres categorías anteriores, *i.e.,* provienen de errores físicos no detectados o de errores en el software. Son también conocidas con el nombre de *fallas universales* o *fallas maliciosas*. Todo comportamiento fuera de las especificaciones (principalmente cuando los resultados no están de acuerdo con lo especificado), es calificado como comportamiento bizantino.

Un cambio en una política del orden en las transmisiones es una alteración en las especificaciones y constituye, por lo tanto, una falla bizantina.

4.3 DCAA

Disponibilidad y Consistencia Ajustable a las Aplicaciones (DCAA) es un protocolo de tolerancia a fallas para el procesamiento de streams que permite que un DSMS distribuido pueda hacer frente a fallas, tanto en la red como en los nodos. Este protocolo logra ajustarse a las aplicaciones, las cuales tienen diferentes requerimientos de disponibilidad y consistencia. DCAA minimiza el número de tuplas tentativas mientras garantiza que los resultados correspondientes a cualquier tupla se envíen downstream dentro de un tiempo preestablecido y que las aplicaciones eventualmente reciban los streams de salida en forma completa y correcta.

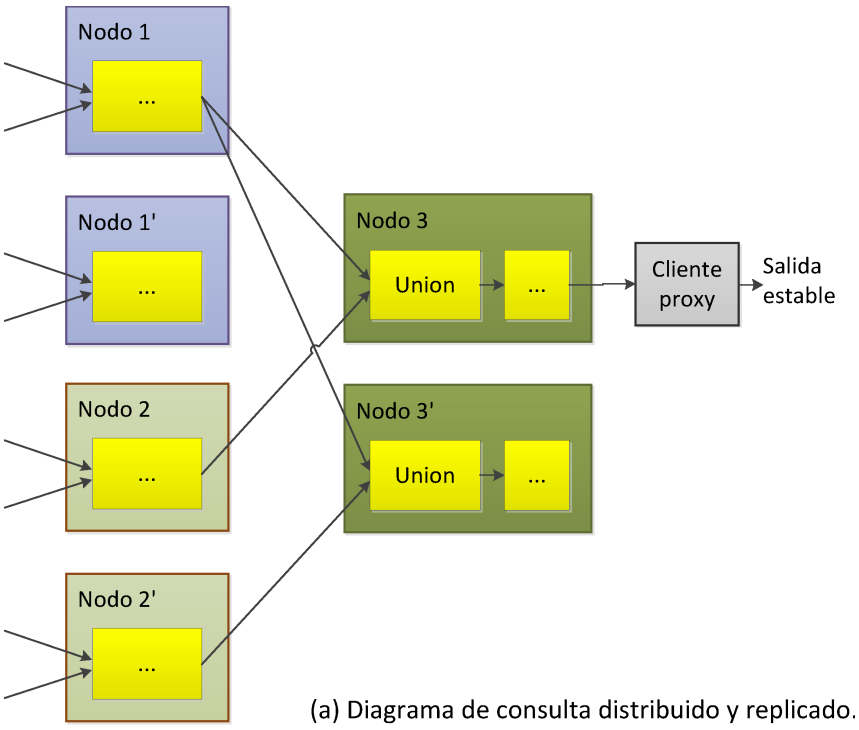


Figura 4.1: Diagrama de consulta distribuido y replicado

La Figura 4.1 ilustra la salida producida por un sistema que emplea DCAA. Como puede verse en dicha figura, en DCAA, todas las réplicas continuamente procesan datos, y un nodo puede utilizar cualquier réplica de sus vecinos superiores (*upstream*) para obtener sus streams de entrada. En ausencia de fallas el cliente recibe una salida estable. En el ejemplo (Figura 4.2), si el nodo 1 falla, entonces el nodo 3 (y su réplica, nodo 3') puede reconectarse con el nodo 1', asegurando que la aplicación cliente continúe recibiendo información actual y correcta. Las tuplas de

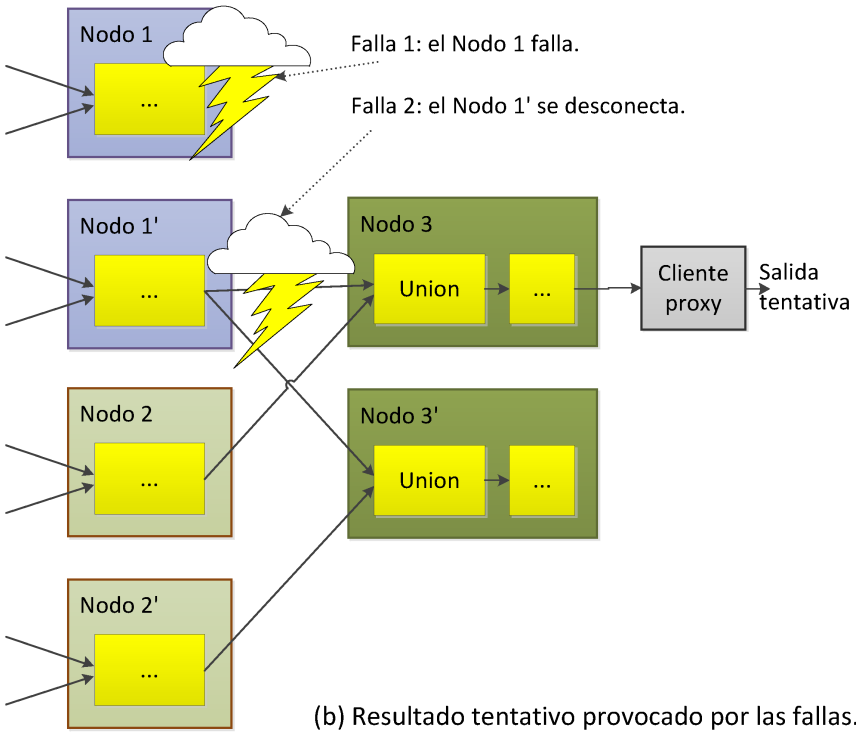


Figura 4.2: Fallas que originan tuplas tentativas

salida continúan siendo etiquetadas como "estables". La falla es enmascarada por el sistema. Sin embargo, si el nodo 1' se desconecta mientras el nodo 1 sigue caído, el sistema será incapaz de enmascarar la falla. Durante un tiempo predefinido el sistema puede suspender el procesamiento y por lo tanto el cliente no recibirá datos. Pero, si la falla persiste, el procesamiento de las entradas que siguen disponibles (streams producidos por el nodo 2) deben continuar para poder asegurar su disponibilidad. Luego el procesamiento continúa, pero como algunos streams de entrada están faltando, las tuplas de salida se etiquetan como "tentativas". Una vez que la falla se soluciona la aplicación cliente sigue recibiendo resultados basados en los datos de entrada más recientes, también etiquetados como "tentativos", mientras se reciben correcciones sobre resultados tentativos anteriores. Este proceso continúa hasta que las correcciones alcanzan los resultados de salida más recientes. En este punto, la aplicación cliente recibe solamente la información más reciente y estable.

4.4 Definición del problema

Para definir el problema es importante identificar los requerimientos de tolerancia a fallas adecuados al procesamiento de aplicaciones de procesamiento de streams, enfatizando las similitudes y diferencias en cuanto a sus objetivos de disponibilidad

y consistencia. Luego presentaremos nuestros objetivos de diseño, enunciando las propiedades deseadas de nuestro sistema. A continuación se exhiben las asunciones acerca del sistema y el tipo de fallas que debe ser capaz de tolerar y finalmente, debido a que algunas características de los operadores afectan las propiedades y overhead de DCAA, presentaremos una clasificación de los operadores. Comenzaremos nuestra discusión considerando los requerimientos de las aplicaciones.

Muchas aplicaciones de procesamiento de streams monitorean fenómenos que están sucediendo en ese preciso momento y requieren resultados producidos con baja latencia (velocidad). Estas aplicaciones incluso muchas veces valoran más el procesamiento con bajas latencias antes que la precisión, obviamente el máximo tiempo de procesamiento soportado depende de cada aplicación. Existen muchos ejemplos de este tipo de aplicaciones:

Monitoreo de redes. Un administrador de sistemas depende del monitoreo de las redes para poder conocer en forma continua el estado de su red y detectar anomalías, como posibles intentos de intrusión, eventos de propagación de worms, o simplemente situaciones de sobrecarga. Administradores de diferentes dominios pueden incluso compartir parte de sus streams de información para mejorar sus capacidades de detección [KKK05], aún si sólo cuentan con un subconjunto de monitores disponibles. Más aún, las bajas latencias de procesamiento son críticas para manejar condiciones anómalas en forma rápida y eficiente: mitigar ataques, detener la propagación de worms tan pronto como aparecen, o reaccionar velozmente a las situaciones de sobrecarga en la red[1].

Monitoreo de entornos mediante sensores. Los sensores se vuelven cada día más baratos y pequeños y es cada vez más frecuente encontrarlos dentro de estructuras como tubos de gas, aire, agua, edificios, etc. Estos sensores monitorean la "salud" de las infraestructuras continuamente. Si ocurre una falla que imposibilita que se procesen datos provenientes de un subconjunto de sensores, continuar procesando con la información restante puede ayudar a detectar problemas al menos tentativamente. Por ejemplo, la temperatura del aire puede ser un poco mayor en una parte de un edificio, lo que significa que existe una falla en el sistema de aire acondicionado o simplemente en ese momento el sol está calentando especialmente ese sector del edificio. Si no hay información de habitaciones adyacentes, el sistema puede tentativamente declarar la aparición de una falla. En este caso sería adecuado enviar un técnico para la valoración

[1] El autor de esta tesis dirige un Proyecto de Investigación centrado en el monitoreo de redes para la detección distribuida de intrusos [Ech08].

final. En contraste con el monitoreo de red, esta aplicación puede tolerar que el procesamiento sea suspendido por unos pocos minutos, si esto ayuda a reducir el número potencial de problemas que luego se tornan benignos.

Equipo de rastreo basado en RFID. En esta aplicación se adjunta un tag RFID a cada pieza de equipo y se instalan antenas en el terreno. Las antenas detectan los tags que aparecen en su vecindad y transmiten las lecturas en forma de streams, posibilitando el seguimiento del equipo. Las fallas pueden causar que el sistema pierda algunas lecturas porque la información producida por un subconjunto de antenas puede estar temporalmente no disponible. Sin embargo, continuar con los datos restantes puede alcanzar a satisfacer un conjunto de consultas acerca de la ubicación de equipos. Por ejemplo, un usuario podría ser capaz de ubicar rápidamente una pieza de equipo que necesite, aún si no hay lecturas actualmente disponibles para uno de los pisos del edificio. En esta aplicación las consultas individuales pueden tener distintos niveles de tolerancia a las demoras de procesamiento.

Las aplicaciones arriba mencionadas no sólo necesitan recibir resultados rápidamente sino que también requieren eventualmente obtener resultados correctos, como se describe a continuación:

Monitoreo de redes. Cuando se procesan datos provenientes solamente de un subconjunto de monitores de red, algunos eventos pueden pasar sin ser detectados, y algunos resultados agregados pueden ser incorrectos. Los eventos que se pierden pueden ser importantes, y por ello todavía pueden requerir algún tipo de acción, *e.g.,* limpieza en una máquina infectada. Los valores finales de resultados agregados corregidos, como por ejemplo utilización de ancho de banda de cada cliente, pueden ser necesarios para futuros análisis. El administrador de la red puede eventualmente requerir el resultado corregido y completo.

Monitoreo de entornos mediante sensores. Cuando una falla se soluciona puede suceder que algunas de las alarmas tentativas sean en realidad falsos positivos mientras que otras alarmas realmente representan problemas. Los valores correctos finales, especialmente aquellos que arribaron poco después de que la falla fuera solucionada pueden ayudar a reasignar técnicos más eficientemente.

Equipo de rastreo basado en RFID. Eventualmente reprocesar los datos de entrada completos y correctos puede ayudar a determinar la precisión de la información de utilización de cada pieza de equipo. Esta información, por ejemplo,

puede ser luego requerida para tareas de mantenimiento.

Muchas otras aplicaciones de procesamiento de streams comparten requerimientos similares: *e.g.,* servicios financieros, aplicaciones militares, monitoreo de tráfico mediante GPS, etc. Estas aplicaciones necesitan nuevos resultados dentro de un tiempo de demora máximo, y además eventualmente necesitan los datos correctos. Obviamente algunas aplicaciones de procesamiento de streams requieren consistencia absoluta, como por ejemplo el monitoreo de pacientes mediante sensores.

DCAA también soporta este tipo de aplicaciones dado que permite que las aplicaciones ajusten el *trade-off* entre disponibilidad y consistencia. Una aplicación puede entonces asignar un umbral infinito indicando que nunca deben producirse resultados inconsistentes. Adicionalmente una aplicación puede descartar todos los resultados tentativos y esperar a los resultados estables.

4.4.1 Objetivos de diseño

Dados los requerimientos de las aplicaciones arriba mencionadas, a continuación se presentan los objetivos de tolerancia a fallas, métricas y propiedades de diseño de DCAA.

4.4.1.1 Objetivo de consistencia

En un sistema replicado, la noción de consistencia más fuerte es la *consistencia atómica* [GL02, Lyn96], también conocida como *single-copy consistency* [SS05], *consistencia linearizable* [GL02, SS05] o *serializable* [GHOS96]. Para proveer consistencia atómica, todos los accesos a un objeto replicado deben aparecer como si se hubiesen ejecutado en una única ubicación siguiendo alguna ejecución serial. En un DSMS la consistencia atómica aseguraría que los clientes recibiesen únicamente tuplas de salida correctas. En el Capítulo 1 mencionamos que para mantener una consistencia atómica se requiere que el sistema sacrifique disponibilidad cuando se producen ciertos tipos de falla, como por ejemplo las particiones de red [GL02].

Para mantener disponibilidad los esquemas de replicación optimista suelen garantizar una noción de consistencia más débil, llamada *consistencia eventual* [SS05]. Para lograr disponibilidad mediante consistencia eventual las réplicas pueden procesar pedidos de los clientes aún si no conocen todavía su orden final, dejando que su estado diverja. Sin embargo, si todas las operaciones de actualización se detienen, las réplicas deben eventualmente converger al mismo estado. Para lograr consistencia eventual, todas las réplicas del mismo objeto deben eventualmente procesar todas

las operaciones en un orden equivalente [SS05]. Si las operaciones son enviadas continuamente la consistencia eventual requiere que el prefijo de las operaciones en el orden final crezca monótonamente en el tiempo en todas las réplicas [SS05]. Un servicio de datos que ofrece consistencia eventual puede modelarse como eventualmente serializable (*i.e.,* manteniendo las operaciones requeridas "en un orden parcial que gravita en el tiempo hacia un orden total" [FGL+96]).

Muchas aplicaciones de procesamiento de streams favorecen disponibilidad sobre consistencia. Sin embargo necesitan eventualmente recibir resultados correctos; es por ello que nuestro objetivo para DCAA es proveer *consistencia eventual*.

En un DSMS, el estado de procesamiento de los nodos es temporal y el stream de salida es continuo, es por esto que definimos la consistencia eventual como el requerimiento por el que todas las réplicas del mismo fragmento del diagrama de consultas eventualmente procesan las mismas tuplas de entrada en el mismo orden. El orden es tal que debe ser igual al que hubiese podido suceder contando con un único nodo de procesamiento sin ocurrencia de fallas.

La consistencia eventual es una propiedad de un objeto replicado. Las respuestas sobre operaciones efectuadas sobre el objeto no tienen que necesariamente ser corregidas después de que las operaciones son procesadas en su orden final [FGL+96]. En un DSMS la salida de los nodos de procesamiento sirve como entrada de los vecinos downstream y por ende es necesario extender la noción de consistencia eventual para incluir a los streams de salida. Para ello es necesario que cada réplica eventualmente procese las mismas tuplas de entrada en el mismo orden y produce las mismas tuplas de salida en el mismo orden.

En resumen, el primer objetivo de DCAA es:

Propiedad 4.1 Asumiendo buffers suficientemente grandes [2], debe asegurarse consistencia eventual. ■

Donde la consistencia eventual se define como:

Definición 4.1 *Consistencia Eventual*
Un DSMS replicado mantiene consistencia eventual si todas las réplicas del mismo fragmento de diagrama de consultas eventualmente procesa las mismas tuplas de entrada en el mismo orden y produce las mismas tuplas de salida en el mismo orden. El orden debe ser igual al que hubiese podido suceder contando con un único nodo de procesamiento sin ocurrencia de fallas. ■

[2] En la Sección 4.12 describiremos la gestión de buffers y las fallas de larga duración.

Una vez que se conoce el orden de procesamiento final de algunas operaciones, se dice que estas operaciones son estables [FGL+96]. Para las tuplas utilizamos esta misma definición. Una tupla de entrada es estable una vez que se conoce su orden de procesamiento final. Cuando una réplica procesa tuplas de entrada estables entonces produce tuplas de salida estables porque los clientes eventualmente reciben versiones estables de todos los resultados.

Todos los resultados intermedios que se producen para obtener disponibilidad pero que no son estables son llamados *tentativos*. En todo momento (cualquier punto en el tiempo), como medida de consistencia empleamos $\mathtt{N}_{\mathtt{tentativas}}$ que es el *número de tuplas tentativas producidas en todos los streams de salida de un diagrama de consulta*. Es posible pensar en $\mathtt{N}_{\mathtt{tentativas}}$ como un (crudo) sustituto del grado de divergencia entre las réplicas del mismo diagrama de consulta cuando el conjunto de los streams de entrada no es el mismo que el de las réplicas. Más en detalle, empleamos la siguiente definición:

Definición 4.2 $N_{tentativas}(s)$
La inconsistencia de un stream s, $\mathtt{N}_{\mathtt{tentativas}}(s)$, el número de tuplas tentativas producidas en s desde la última tupla estable. La inconsistencia, $\mathtt{N}_{\mathtt{tentativas}}$, de un diagrama de consulta es la suma de las tuplas tentativas producidas en todos los streams de salida de un diagrama de consulta desde que se produjo la última tupla estable. ■

4.4.1.2 Objetivo de disponibilidad

La definición tradicional de disponibilidad solamente requiere que el sistema eventualmente produzca una respuesta para cada pedido [GL02]. La disponibilidad puede también utilizarse para medir la fracción de tiempo en la que el sistema está funcionando y procesando pedidos (*i.e.,* el tiempo entre fallas dividido la suma de la duración de la falla, la duración de la recuperación, y el tiempo entre las fallas) [GR92]. Sin embargo, dado que en un DSMS las aplicaciones cliente pasivamente esperan a recibir los resultados de salida, definimos disponibilidad en términos de latencia de procesamiento, donde un bajo nivel de latencia de procesamiento indica un alto nivel de disponibilidad.

Para simplificar el problema medimos disponibilidad en términos del *incremento en la latencia de procesamiento*. Cuando una aplicación envía una consulta al sistema, DCAA permite que la aplicación especifique la disponibilidad deseada, llamémosla X, como la latencia de procesamiento incremental máxima que la aplicación puede tolerar sobre sus streams de salida (éste mismo umbral se aplica a todos los streams de salida dentro de la consulta). Por ejemplo, en un diagrama de

consulta que tarda 60 segundos en transformar un conjunto de tuplas de entrada en un resultado de salida, un cliente puede solicitar que el resultado "no demore más de 30 segundos por sobre el tiempo de procesamiento", y DCAA debe asegurar que los resultados de salida se produzcan en no más de 90 segundos.

Para determinar si el sistema logra cumplir con la solicitud de disponibilidad según la definición anterior, debemos solamente medir el buffering extra y la demora impuesta por sobre el procesamiento normal. Definimos $\texttt{Demora}_{\texttt{nueva}}$ como el incremento máximo en la latencia de procesamiento para toda tupla de salida y expresamos el objetivo de disponibilidad como $\texttt{Demora}_{\texttt{nueva}} < X$. Con esta definición presentamos el segundo objetivo de DCAA como:

Propiedad 4.2 DCAA asegura que mientras esté disponible algún camino de operadores no bloqueantes [3] entre uno o más orígenes (fuentes) de datos y una aplicación cliente, el cliente recibirá resultados dentro del tiempo deseado de disponibilidad: el sistema asegura que $\texttt{Demora}_{\texttt{nueva}} < X$. ■

DCAA divide X entre los nodos de procesamiento, como discutiremos más adelante. Para asegurar la Propiedad 4.2 un nodo que se encuentra experimentando una falla de un stream de entrada debe cambiar a otras réplicas de su vecino superior (upstream), si existe al menos una de estas réplicas dentro de D unidades de tiempo luego del arribo de la tupla más antigua no procesada. Si no existe ninguna réplica, entonces el nodo debe procesar todas las tuplas que todavía se encuentran disponibles dentro de D unidades de tiempo de su arribo, donde D es la latencia de procesamiento incremental máxima asignada al nodo.

$\texttt{Demora}_{\texttt{nueva}}$ mide solamente la disponibilidad de las tuplas resultado que llevan nueva información. Este conjunto de tuplas será denominado `NuevaSalida`, y excluye cualquier resultado estable que corrige cualquier resultado tentativo previo.

Aún aunque solamente midamos las latencias incrementales podemos mostrar como $\texttt{Demora}_{\texttt{nueva}}$ se relaciona con la latencia normal de procesamiento. Definimos $\texttt{proc}(t)$ como la latencia normal de procesamiento de una tupla de *salida*, t, en ausencia de fallas. $\texttt{proc}(t)$ es la diferencia entre el tiempo en el que DSMS produce t y el tiempo en el que la tupla de entrada más antigua que contribuyó al valor de t entró en el DSMS. Dado $\texttt{proc}(t)$ y la latencia de procesamiento actual de una tupla, $\texttt{demora}(t)$, $\texttt{Demora}_{\texttt{nueva}} = max(\texttt{demora}(t) - \texttt{proc}(t))$, donde $t \; \epsilon$ `NuevaSalida`.

[3] Discutiremos los operados bloqueantes y no bloqueantes en la Sección 4.4.3.

4.4.1.3 Minimización de la inconsistencia

El objetivo principal que debe tratar de asegurar DCAA es que el sistema logre un nivel de disponibilidad predefinido mientras garantiza consistencia eventual. Para obtener disponibilidad el sistema produce tuplas tentativas y para mantener consistencia eventual estas tuplas tentativas son luego corregidas mediante tuplas estables. Corregir resultados previos es una tarea costosa en un DSMS y por lo tanto es fundamental poder minimizar la cantidad de tuplas tentativas. En ausencia de fallas es deseable contar con réplicas mutuamente consistentes que mantengan consistencia linearizable y que aseguren que los resultados sean estables. Si ocurre una falla, nuestro protocolo DCAA debe tratar de superarla sin introducir inconsistencias. Si no puede superarla, es deseable que el sistema minimice el número de resultados tentativos. Estos requerimientos se resumen en las siguientes dos propiedades que caracterizan a DCAA:

Propiedad 4.3 DCAA favorece los resultados estables por sobre los tentativos cuando están disponibles ambos tipos de resultados. ■

Propiedad 4.4 Al mismo tiempo que se proveen mecanismos que garanticen las Propiedades 4.1 y 4.2, se buscan métodos que minimicen $\mathtt{N}_{\mathtt{tentativas}}$. ■

4.4.2 Modelo de fallas y asunciones

Antes de presentar DCAA en detalle identificaremos el tipo de fallas que el sistema debe ser capaz de superar y las asunciones fundamentales y de simplificación que inciden sobre él.

4.4.2.1 Asunciones fundamentales

Asumimos que el diagrama de consulta y su despliegue (*i.e.,* la asignación de operadores a los nodos de procesamiento) es estático. También asumimos que el conjunto de réplicas para cada nodo de procesamiento es estático. Los cambios dinámicos sobre el diagrama y sobre el despliegue quedan fuera del alcance de este trabajo y da posibilidades de extensión del sistema en trabajos futuros.

También asumimos que los orígenes (*sources*) de datos o *proxies* que actúan como representantes, han sincronizado relojes (débilmente) y marcan las tuplas que introducen al sistema con estampillas de tiempo.

Cada vez que se juntan (`join`), unen (`union`), o se combinan mediante algún operador dos o más streams DCAA demora las tuplas hasta que las estampillas de tiempo coincidan en todos los streams. Los relojes en las fuentes de datos deben

estar "bien sincronizados", *i.e.,* con un margen de divergencia acotado para así poder asegurar que las demoras en el buffering sean menores que la latencia incremental de procesamiento, X.

Similarmente, cuando los operadores procesan tuplas, asignan estampillas de tiempo en la tuplas de salida. Es posible emplear una variedad de algoritmos, pero se asume que el algoritmo de asignación de estampillas de tiempo elegido combinado con la estructura del diagrama de consulta asegura que las estampillas de tiempo de las tuplas coincidan aproximadamente cada vez que múltiples streams son procesados por el mismo operador. Una vez más, demorar las tuplas hasta que su estampilla de tiempo coincida debe causar demoras que no superen X. Estos requerimientos son similares a aquellos que uno pudiese esperar de una aplicación que tiene un atributo de servicio a partir de especificaciones de ventana.

Además asumimos que cada nodo de procesamiento cuenta con suficientes recursos (CPU, memoria y ancho de banda de red) para soportar un ritmo de tuplas de entrada tal que en ausencia de fallas no se formen colas. Asumimos también que la latencia de red entre cualquier par de nodos es pequeña comparada con la máxima latencia incremental de procesamiento, X.

DCAA soporta fallas de tipo crash [Sch93] de los nodos de procesamiento. Cuando un nodo de procesamiento falla se detiene (*halt*) sin producir resultados erróneos pero el hecho de que el DSMS caiga puede pasar desapercibido por otros DSMS. En particular, un crash del nodo puede no distinguirse de una falla de red.

DCAA también es capaz de soportar fallas y particiones de red. Una falla de red, causante de demoras o incluso pérdida de paquetes, evita la comunicación entre un subconjunto de nodos. Nuestro protocolo interpreta las demoras extensas como fallas. Dado que la versión actual de nuestro sistema maneja únicamente despliegues de diagramas de consulta estáticos maneja solamente fallas temporales. Se asume que si un nodo de procesamiento falla eventualmente recomienza y se reintegra al sistema, partiendo de un estado vacío [4].

Al principio de este capítulo asumimos que todas las tuplas producidas por un nodo de procesamiento son *buffereadas*. Hablaremos de esta asunción en la Sección 4.12, donde discutiremos gestión de buffers y fallas prolongadas.

Excepto para las fuentes de datos, asumimos que los buffers pueden perderse cuando un nodo de procesamiento falla. Además se asume que tanto las fuentes de datos como los clientes implementan DCAA, y que las fuentes de datos emplean logs persistentes (o de otra forma backups) para registrar los datos que producen antes de

[4] En la Sección 4.11 discutiremos el proceso de recuperación de los nodos fallados.

enviarlos al sistema. Con esta asunción, aún luego de fallar y recuperarse, las fuentes de datos pueden asegurar que todas las réplicas del primer nodo de procesamiento recibirán la misma secuencia de entradas. En la Sección 4.13 discutiremos cómo se lleva a cabo este procedimiento.

Si las tuplas se guardan en logs eternamente, DCAA puede resistir fallas en todos los nodos de procesamiento. Si todas las réplicas de un nodo de procesamiento no están disponibles, los clientes no reciben ningún dato. Luego de que los nodos fallados se recuperan (y parten de un estado vacío), pueden reprocesar todas las tuplas logueadas upstream asegurando consistencia eventual. Sin embargo, si los buffers se truncan en algún momento, al menos existirá una réplica de cada nodo de procesamiento que mantenga el estado consistente actual. Este estado comprende al conjunto de tuplas de entrada estable que no se encuentran en buffers upstream y al conjunto de tuplas de salida estable que no se recibieron todavía en todos los vecinos downstream. Entonces, debido al truncado de buffers, DCAA es capaz de resistir la falla (*crash*) simultánea de a lo sumo $R - 1$ de entre R réplicas de cada nodo de procesamiento. Más adelante mostraremos como maneja tanto una única falla como múltiples fallas (fallas que se superponen en el tiempo).

DCAA también brinda tolerancia a fallas de la fuente de datos. Para mantener disponibilidad frente a la presencia de una falla en la fuente de datos, el sistema procesa las tuplas de entrada restantes como tentativas hasta que la fuente de datos se recupere. Una vez recuperada, el DSMS procesa como estables todas las tuplas, incluyendo las tuplas previamente perdidas si la fuente de datos produjo datos durante la falla y es capaz de repetir esos datos. Como en el caso de la falla de los nodos, DCAA únicamente soporta fallas temporales de fuentes de datos. Una falla permanente sería equivalente a un cambio en el diagrama de consulta.

4.4.2.2 Asunciones de simplificación

Asumimos que las réplicas se comunican empleando un protocolo de red confiable y con entrega de paquetes en orden, como por ejemplo TCP [APB09]. Con esta asunción los nodos pueden contar con que las tuplas transmitidas desde upstream arriban en el orden en el que fueron producidas. Un nodo downstream puede, por ejemplo, indicar con una tupla de identificación exactamente qué datos fueron recibidos hasta el momento.

Es posible afirmar que DCAA está diseñado para un bajo nivel de replicación y para una baja frecuencia de fallas, más adelante se desarrollan ambas aseveraciones.

4.4.3 Clasificación de operadores

Existen propiedades en varios operadores que incrementan su overhead y restringen las garantías que DCAA es capaz de proveer. En esta sección se categorizan los operadores según dos ejes: si se bloquean o no cuando faltan algunos de sus streams de entrada y cómo actualizan su estado mientras procesan tuplas de entrada.

4.4.3.1 Operadores bloqueantes y no bloqueantes

Los operadores de procesamiento de streams efectúan sus cómputos sobre ventanas deslizantes de datos mientras arriban nuevas tuplas. Sin embargo hay operadores como por ejemplo `Join`, que se bloquean cuando algunas de sus entradas están faltando y de hecho, si no arriban tuplas en un stream de entrada, eventualmente no habrá tuplas para el Join con las tuplas más recientes en el otro stream. En contraste, la `Union` es un ejemplo de un operador no bloqueante pues puede realizar procesamiento útil aún cuando algunos de sus streams de entrada están faltando. Por supuesto todos los operadores se bloquean si todas sus entradas están ausentes.

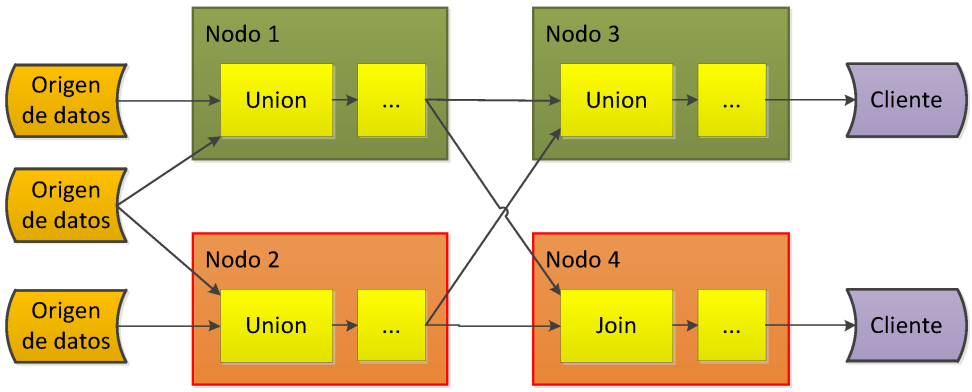


Figura 4.3: Diagrama de consulta compuesto por operadores bloqueantes (Join) y no bloqueantes (Union)

La Figura 4.3 (arriba) ilustra el impacto de los operadores bloqueantes y no bloqueantes con respecto a la tolerancia a fallas mediante un diagrama de consulta desplegado en cuatro nodos. En este ejemplo la falla de la fuente de datos no evita que el sistema siga procesando los streams restantes. La falla de los nodos 1 ó 2 no bloquea al nodo 3, pero efectivamente bloquea al nodo 4. Sólo los operadores no bloqueantes mantienen la disponibilidad cuando algunas de sus entradas están faltando.

4.4.3.2 Determinismo y convergencia

La forma en que los operadores actualizan su estado y producen tuplas de salida en respuesta a las tuplas de entrada influye en los algoritmos de tolerancia a fallas. A continuación presentaremos una taxonomía de operadores basada en esta propiedad. Esta taxonomía es la presentada por Hwang *et.al.* [HBR+05] excepto por nuestra definición de determinismo.

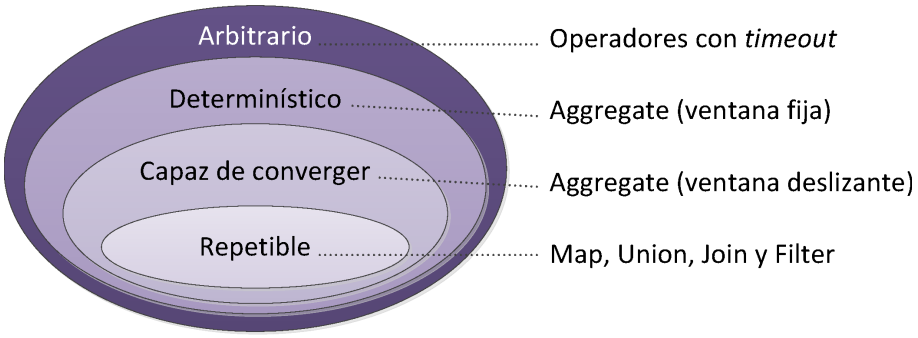


Figura 4.4: Taxonomía de operadores de procesamiento de streams con ejemplos para cada categoría

Distinguimos cuatro tipos de operadores: *arbitrarios* (incluyendo a los no determinísticos), *determinísticos*, *capaces de converger*, y *repetibles*. La Figura 4.4 muestra la relación de inclusión entre estos tipos de operadores y ejemplos para cada categoría.

Un operador es determinístico si produce el mismo stream de salida cada vez que se parte del mismo estado inicial y se procesan las mismas tuplas de entrada en la misma secuencia. La secuencia no sólo define el orden de las tuplas en cada stream de entrada separadamente sino que también establece el orden de procesamiento absoluto en *todas* las tuplas de entrada. Con esta definición existen entonces dos únicas posibles causas para el no determinismo en los operadores: dependencia en tiempo de ejecución o tiempo de arribo de tuplas de entrada (*e.g.*, operadores que con un parámetro de *timeout* producen una tupla de salida cuando no arriban tuplas de entrada durante algún período de tiempo preestablecido) y el uso de mecanismos aleatorios (*e.g.*, un operador que aleatoriamente descarta tuplas para bajar el nivel de carga [TcZ+03]). Por el momento nuestro protocolo soporta únicamente operadores determinísticos, el resto de los operadores podría incluirse en trabajos futuros.

Es importante destacar que un diagrama de consulta compuesto por operadores determinísticos no es por si mismo y automáticamente determinístico. Para que el diagrama sea determinístico debemos asegurar que todos los operadores procesen tuplas de entrada siguiendo un orden determinístico. En la Sección 4.8 presentaremos

una técnica que permite obtener determinismo en las consultas en general.

Debido a que el estado de un operador se define en términos de la ventana de tuplas de entrada que procesa, la manera en la que estas ventanas se mueven determina si el operador es o no convergente.

Un operador determinístico es llamado *capaz de converger* si puede empezar a procesar tuplas de entrada desde cualquier momento (tiempo), y converger siempre al mismo estado consistente luego de procesar suficientes tuplas de entrada (asumiendo por supuesto, que en todas las ejecuciones las tuplas de entrada tienen los mismos valores y llegan en el mismo orden). La convergencia permite mayores posibilidades al momento de elegir una técnica para un operador o un nodo fallado para reconstruir un estado consistente. Mientras presentamos DCAA distinguiremos entre los operadores capaces de converger y otros operadores determinísticos.

Para asegurar convergencia, toda tupla de entrada debe afectar el estado de un operador durante una cantidad de tiempo limitada, y el operador siempre debe converger para procesar el mismo grupo de tuplas de entrada. En la Sección 3.2.2, discutimos diferentes tipos de especificaciones de ventanas. Típicamente un operador con una ventana deslizante es capaz de converger. Por ejemplo, para una ventana de tamaño 100, un avance de 10 y una primer tupla 217, un operador de agregación (*Aggregate*) puede converger su cómputo con límites de ventanas que son múltiplos de 10: [310,410), [320,420), *etc.* Sin embargo, el operador no es capaz de converger si todos los límites de ventana están completamente definidos por el valor de la primer tupla, *e.g.,* [317,417), [327,427), *etc.* Las ventanas fijas (*landmark*) también pueden impedir la convergencia porque un límite de la ventana podría nunca moverse.

Finalmente, decimos que el operador `join` es capaz de converger porque típicamente alinea su ventana con respecto a cada tupla de entrada.

Un operador capaz de converger es también repetible si puede recomenzar a procesar sus tuplas de entrada partiendo de un estado vacío y un punto anterior en el tiempo, y seguir produciendo únicamente tuplas con los mismos valores (duplicados idénticos) y en el mismo orden. Una condición necesaria para que un operador sea repetible es que el operador use a lo sumo una tupla de cada stream de entrada para producir una tupla de salida. Si una secuencia de múltiples tuplas contribuye para una tupla de salida, entonces recomenzar el operador desde la mitad de la secuencia puede producir al menos una tupla diferente en la salida. Por lo tanto, el operador de agregación (*Aggregate*) no es generalmente repetible, mientras que el de filtrado (*Filter*), el cual simplemente descarta tuplas que no coinciden con un dado predicado, y *Map*, que transforma tuplas a partir de la aplicación de funciones a sus atributos, son repetibles, pues tienen un único stream de entrada y procesan

cada tupla independientemente de las otras. *Join*, sin *timeout*, también es repetible si alinea ventanas con respecto a la última tupla de entrada que está siendo procesada.

La propiedad de repetición afecta la facilidad con la que las tuplas duplicadas pueden ser eliminadas si un operador recomienza el procesamiento a partir de un estado vacío y un punto anterior en el stream. A partir de este comportamiento es claro que esta repetibilidad afecta el mecanismo de tolerancia a fallas en general; sin embargo, nuestros esquemas no necesitan distinguir entre operadores capaces de converger y repetibles.

En resumen, dos son las características de los operadores que más impactan en DCAA. La naturaleza bloqueante de algunos operadores afecta la disponibilidad durante las fallas y por otro lado, la naturaleza que posibilita la convergencia en algunos operadores dando mayor flexibilidad en la recuperación de los nodos que presentan fallas y pudiendo colaborar en la reducción del overhead (el cual es discutido en la la Sección 4.12).

4.5 Generalidades sobre DCAA

En esta sección describiremos las generalidades del comportamiento esperado de un nodo de procesamiento. La idea principal de DCAA es favorecer la autonomía de las réplicas. Cada réplica es considerada como un nodo de procesamiento independiente. Cada nodo procesa datos, monitorea el estado de sus streams de entrada, monitorea su disponibilidad, y maneja su consistencia. Estas actividades son posibles gracias al protocolo DCAA, el cual sigue el autómata finito que cuenta con tres estados: ESTABLE, FALLA_UPSTREAM y ESTABILIZACIÓN (Figura 4.5).

Mientras todos los vecinos *upstream* de un nodo producen tuplas estables entonces el nodo está en estado ESTABLE. En este estado el nodo procesa tuplas a medida que van llegando y pasa resultados estables a los vecinos *downstream*. Las réplicas pueden recibir sus entradas en diferente orden y por lo tanto para mantener la consistencia entre las réplicas definiremos un operador de serialización de datos al que llamaremos `SUnion`. La Sección 4.8 discute el estado ESTABLE y el operador `SUnion`.

Si uno de los streams de entrada no está disponible o comienza a producir tuplas tentativas, entonces el nodo pasa al estado FALLA_UPSTREAM, estado en donde trata de encontrar otra fuente estable para ese stream de entrada. Si no encuentra otra fuente estable entonces el nodo tiene tres opciones para procesar el resto de las tuplas de entrada disponibles:

1. *Suspender* el procesamiento hasta que la falla se solucione y que uno de los

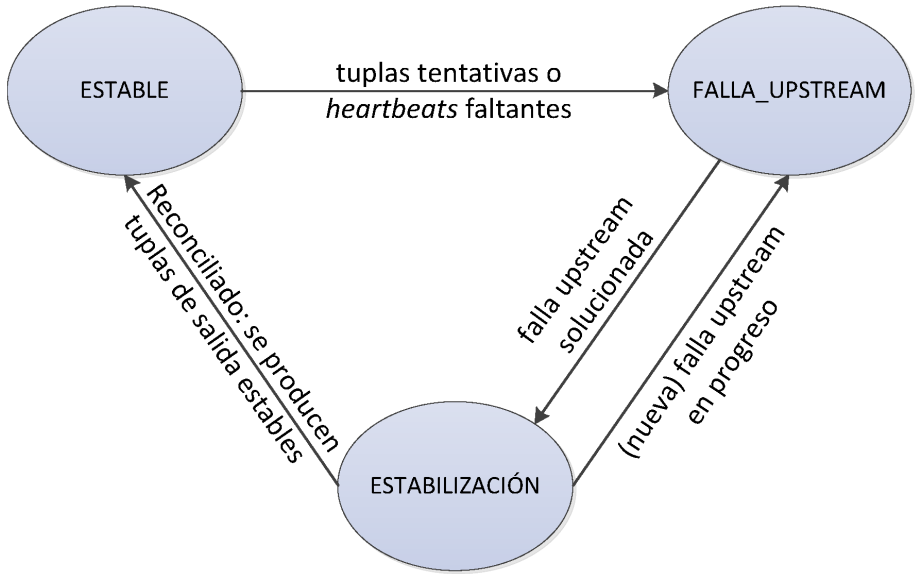


Figura 4.5: Autómata finito de DCAA

vecinos upstream que falló comience a producir datos estables nuevamente.

2. *Demorar* momentáneamente (durante un período corto de tiempo) cada nueva tupla antes de procesarla.

3. *Procesar* cada nueva tupa disponible sin ninguna demora.

La primer alternativa favorece la consistencia, pues no produce ninguna tupla tentativa. Sin embargo, puede ser usada solamente para fallas cortas dado nuestro objetivo de producir nuevas tuplas con demora acotada. Las últimas dos opciones producen tuplas resultado marcadas como "tentativas"; la diferencia entre ellas está en la latencia de los resultados y en el número de tuplas tentativas producidas. La Sección 4.9 discute el estado FALLA_UPSTREAM en detalle.

Una falla se soluciona[5] cuando un vecino upstream previamente no disponible comienza a producir tuplas estables nuevamente o cuando el nodo encuentra otra réplica del vecino upstream capaz de proveer la versión estable del stream. Una vez que un nodo recibe las versiones estables de todas las tuplas previamente faltantes y/o tentativos pasa al estado ESTABILIZACIÓN. En este estado si el nodo procesó al menos una tupla tentativa durante FALLA_UPSTREAM debe ahora reconciliar su estado y estabilizar sus salidas (*i.e.,* corregir las tuplas de salidas que produjo durante la falla). Veremos dos soluciones para reconciliar el estado: un esquema de *checkpoint/redo* y otro de *undo/redo*. Mientras se está efectuando la reconciliación posiblemente sigan arribando tuplas de entrada, en este caso el nodo sigue teniendo

[5]En inglés suele emplearse el término *heals*.

las mismas tres opciones que antes mencionamos para procesar estas tuplas: suspender, demorar, o procesar sin demoras. DCAA posibilita que el nodo reconcilie su estado y corrija sus salidas mientras a su vez asegura que continúe procesando nuevas tuplas. Discutiremos el estado ESTABILIZACIÓN en la Sección 4.10. Una vez culminado el proceso de estabilización, si no existen otras fallas presentes, el nodo pasa al estado ESTABLE, en caso contrario vuelve al estado FALLA_UPSTREAM.

Es claro que un nodo también puede fallar (*fail stop*) en cualquier momento, cuando se recupera el nodo recomienza con un estado vacío y debe reconstruir un estado consistente. Discutiremos la falla y recuperación de un nodo en la Sección 4.11.

El protocolo DCAA requiere que los nodos sean capaces de *bufferear* algunas tuplas. Dado que es imposible que los buffers crezcan sin límites, los nodos deben comunicarse entre sí para truncar periódicamente estos buffers. La gestión de buffers será descripta detalladamente en la Sección 4.12.

Nuestro protocolo de tolerancia a fallas modifica la función de un DSMS en tres niveles, afectando la interacción entre los nodos de procesamiento (*e.g.,* cambiando entre réplicas de un vecino upstream), requiriendo gestiones adicionales sobre tuplas de entrada y salida (*e.g.,* buffereando y replicando tuplas), y finalmente afectando al mismo procesado de tuplas requiriendo la demora y corrección de las correspondientes tuplas tentativas. A continuación presentaremos detalladamente el protocolo en las secciones siguientes.

4.6 Arquitectura de software extendida

La arquitectura de un DSMS debe extenderse para poder correr nuestro protocolo. Estas extensiones se ilustran en la Figura 4.6, donde las flechas indican comunicación entre componentes (tanto mensajes de datos como mensajes de control). Uno de los nuevos componentes es el *Gestor de Consistencia*, encargado de controlar las comunicaciones entre los nodos de procesamiento. El *DataPath* es responsable de llevar cuenta y gestionar los datos que entran y salen del nodo y se extiende con capacidades adicionales de monitoreo y buffereo (*buffering*). Finalmente introduciremos dos nuevos operadores, `SUnion` y `SOutput`, los cuales se agregan al diagrama de consulta para modificar el procesamiento. A continuación presentaremos el rol principal de cada uno de los componentes, los detalles de los mismos serán desarrollados en las próximas secciones mientras presentamos DCAA.

El *Gestor de Consistencia* mantiene una perspectiva global de la situación del nodo de procesamiento dentro del sistema. Conoce acerca de las réplicas del nodo, sus vecinos upstream y sus réplicas, así como también de los vecinos downstream y sus

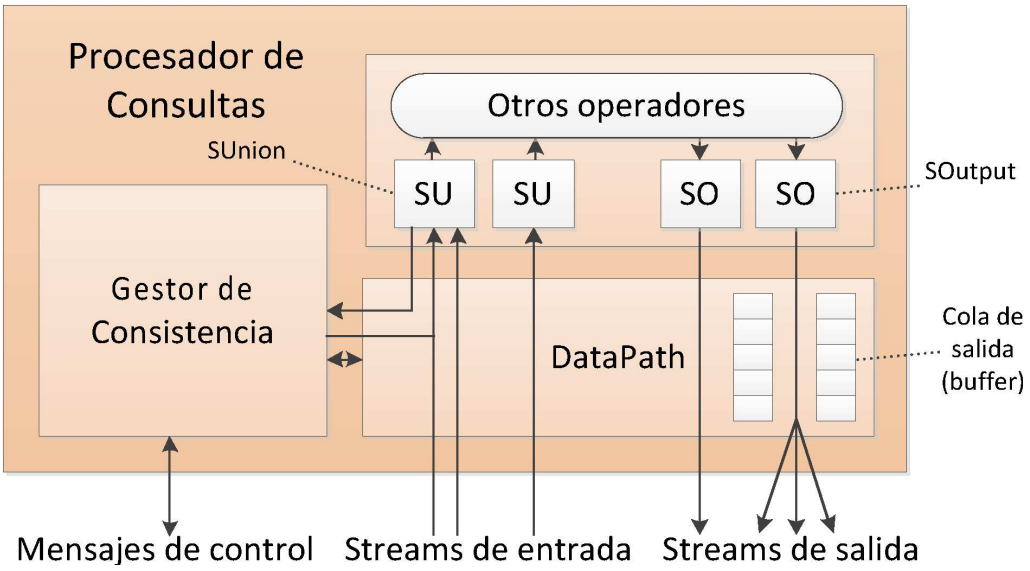


Figura 4.6: Extensiones de software a la arquitecura DSMS para soportar DCAA

réplicas, como consecuencia, el Gestor de Consistencia maneja todas las interacciones entre los nodos de procesamiento. Periódicamente solicita información de estado de los nodos upstream y sus réplicas y decide cuándo cambiar de una réplica a otra. Es claro entonces que el rol global del Gestor de Consistencia hace que las decisiones que tome afecten al nodo de procesamiento como un todo. Por ejemplo, toma la decisión de cuándo efectuar o suspender checkpoints periódicos y cuándo entrar en el estado de ESTABILIZACIÓN. Más adelante veremos que en la implementación del protocolo el Gestor de Consistencia, delega algunas de las funciones anteriores a otros módulos de la arquitectura. En este capítulo veremos este componente como la entidad lógica que realiza todos las tareas antes mencionadas.

El DataPath establece y monitorea los streams de entrada y salida. Este componente solamente conoce el upstream actual y los vecinos downstream. Para los streams de entrada el DataPath lleva cuenta de la entrada recibida y del origen de dicha entrada, asegurando que no entren tuplas indeseadas al DSMS. Para los streams de salida el DataPath asegura que cada cliente downstream reciba la información que necesita, posiblemente respondiendo con datos buffereados.

Finalmente y para lograr un control fino sobre el procesamiento de streams, introducimos dos nuevos operadores, `SUnion` y `SOutput`. El primero asegura que las réplicas se mantengan mutuamente consistentes en ausencia de fallas y gestiona los trade-offs entre disponibilidad y consistencia; bufferea, demora, o suspende el procesamiento de tuplas cuando es necesario. `SUnion` también participa en la ESTABILIZACIÓN. `SOutput` solamente monitorea los streams de salida, descartando posibles duplicados durante el estado de reconciliación. Tanto `SUnion` como `SOutput` envían

señales al Gestor de Consistencia cuando ocurren eventos interesantes, como por ejemplo cuando se procesa la primer tupla tentativa, o cuando se envía downstream la última tupla de corrección. DCAA necesita cambios (discutidos en el Capítulo 5) en todos los operadores del DSMS.

4.7 Modelo de datos mejorado

Con DCAA los nodos y las aplicaciones deben distinguir entre resultados estables y tentativos. Las tuplas estables que se producen luego de la estabilización pueden sustituir a las tuplas tentativas; para ello se requiere que el nodo procese correctamente estas correcciones. Para implementar el mecanismo de corrección de tuplas tentativas extenderemos el modelo de stream de datos tradicional introduciendo nuevos tipos de tuplas.

En el Capítulo 3 discutimos acerca de los streams tradicionales, un stream es una secuencia *append-only* de tuplas de la forma: $(t, a_1, \ldots, a_m)$, donde t es una estampilla de tiempo y $a_1, \ldots, a_m$ son atributos. Para ajustar nuestra nueva semántica de tuplas extenderemos el modelo tradicional:

$$(tipo, id, stime, a_1, \ldots, a_m). \tag{4.1}$$

tipo indica el tipo de la tupla,

id identifica univocamente la tupla dentro del stream,

stime es una nueva estampilla de tiempo.

Tradicionalmente todas las tuplas producen inserciones estables e inmutables. Introduciremos dos nuevos tipos de tuplas: TENTATIVA y DESHACER. Una tupla tentativa es una tupla que resulta del procesamiento de un subconjunto de entradas y que puede subsecuentemente ser corregida mediante una versión estable. Una tupla DESHACER indica que un sufijo de tuplas en un stream debe borrarse y el estado de todos los operadores debe revertirse. Como se ilustra en la Figura 4.7, la tupla DESHACER indica que se debe borrar hasta el *id* de la última tupla que no debe deshacerse. Las tuplas estables que siguen a un DESHACER reemplazan los tuplas tentativas que no fueron completadas. Las aplicaciones que no toleran inconsistencias pueden entonces simplemente desechar las tuplas TENTATIVAs y DESHACER.

Utilizaremos algunos tipos de tupla adicionales pero no provocan cambios radicales en el modelo de datos. La Tabla 4.1 resume los nuevos tipos de tupla.

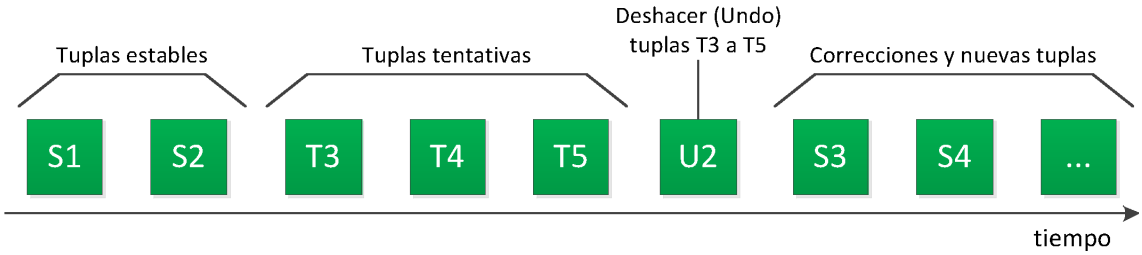


Figura 4.7: Ejemplo del uso de tuplas TENTATIVAs y de DESHACER

Tipo de tupla	Descripción
Streams de datos	
INSERCIÓN	Tupla estable ordinaria.
TENTATIVA	Tupla que resulta de procesar un subconjunto de entradas y que luego puede llegar a ser corregida.
LÍMITE	Todas las tupla subsiguientes tendrán un timestamp igual o mayor que el especificado. Sólo debe asignarse `tipo_tupla` y `stime`.
DESHACER	Las tuplas sufijas deben ser eliminadas y debe revertirse su estado asociado. Excepto por `tipo_tupla` es una copia de la última tupla que no debe deshacerse.
FIN_REC	Tupla que indica el final del estado de reconciliación. Sólo debe asignarse `tipo_tupla`.
Streams de control	
FALLA_UPSTREAM	Se detecta una falla upstream. Si bien sólo debe asignarse `tipo_tupla`, también asignamos `stime` con el momento de comienzo de la falla.
PEDIDO_REC	Se recibió el stream de entrada corregido y está listo para reconciliar estado. Sólo debe asignarse `tipo_tupla`.
FIN_REC	Idem anterior.

Tabla 4.1: Nuevos tipos de tuplas

DCAA también requiere que las tuplas tengan un nuevo campo que aloje una estampilla de tiempo *stime* para establecer un orden serial determinístico para el procesamiento de tuplas que discutiremos a continuación.

4.8 Estado ESTABLE

El estado ESTABLE define la operación de un nodo en ausencia de fallas. Con el objetivo de minimizar las inconsistencias y facilitar la gestión de las fallas, DCAA asegura que todas las réplicas se mantengan mutuamente consistentes en ausencia de fallas mediante el procesamiento de la misma entrada en el mismo orden, pasando por los mismos estados internos de cómputo, y produciendo la misma salida en el mismo orden. A continuación se presenta el mecanismo empleado por DCAA para

mantener la consistencia. En el estado ESTABLE, los nodos deben ser capaces de detectar fallas en sus streams de entrada y así pasar al estado FALLA_UPSTREAM. Luego, en la Subsección 4.8.3 discutiremos el mecanismo de detección de fallas.

4.8.1 Serialización de las tuplas de entrada

Para facilitar el diseño e implementación de DCAA nos limitaremos al empleo de operadores determinísticos. Éstos son los operadores que actualizan su estado y producen salida únicamente basados en los valores y el orden de sus tuplas de entrada (sin timeouts y sin aleatoriedad). Si todos los operadores son determinísticos, para que DCAA logre mantener la consistencia entre las réplicas, debe asegurar que las réplicas del mismo operador procesen datos en el mismo orden, de lo contrario las réplicas generarían resultados divergentes aún sin que existiesen fallas.

Dado que asumimos que los nodos se comunican mediante un protocolo confiable que garantiza paquetes en orden, como por ejemplo TCP, las tuplas nunca son reordenadas dentro de un stream. Dado que cada stream es a su vez generado en un único origen de datos, todas las réplicas de un operador con un único stream de entrada procesan sus entradas en el mismo orden sin ningún mecanismo adicional. Para lograr consistencia solamente necesitamos una forma de ordenar las tuplas determinísticamente cuando existen múltiples streams de entrada para el mismo operador, *e.g.,* `Union` y `Join`.

Si las tuplas sobre los streams estuviesen siempre ordenadas según uno de sus atributos y arribasen a un ritmo constante, el problema de ordenarlas determinísticamente sería sencillo. Cada operador podría *bufferear* tuplas y procesarlas según este atributo en forma creciente, "rompiendo empates" de manera determinística. El desafío está en que las tuplas que pertenecen a un stream no están necesariamente ordenadas por ningún atributo y pueden llegar en cualquier momento y no según un ritmo constante.

Para computar un orden sin el overhead de la comunicación entre réplicas introduciremos tuplas adicionales en los streams. Estas tuplas adicionales serán de tipo LÍMITE y su función será la de marcar extremos entre tuplas [CGM10] y actuar como *heartbeats* [SW04a]. La propiedad de marcar los extremos requiere que no existan tuplas con un *stime* menor que el *stime* del límite que aparece luego del límite del stream[6]. Las tuplas de límite permiten que un operador ordene determinísticamente todas las tuplas (previamente recibidas) con un valor de *stime* menor, ya que

[6]Si un origen de datos no puede producir tuplas de límite o establecer valores *stime*, el primer nodo de procesamiento que "ve" los datos puede actuar como proxy para ese origen de datos, estableciendo tuplas de encabezamiento y produciendo tuplas de límite (ver Sección 4.13).

ello asegura que el operador haya recibido todas estas tuplas. Más específicamente, un operador con i streams de entrada puede ordenar determinísticamente todas las tuplas que satisfacen:

$$stime < \min_{\forall i} (b_i),$$

donde b_i es el valor *stime* de la tupla de la última tupla límite recibida en el stream i. La Figura 4.8 ilustra este mecanismo para tres streams. En el ejemplo, en el tiempo t_0, $min(b_i) = 20$, todas las tuplas con valores *stime* estrictamente menores que 20 pueden ser ordenados. Similarmente, en el tiempo t_1, las tuplas menores que 25 pueden ser ordenadas. En los tiempos t_2 y t_3 solamente las tuplas que se encuentran por debajo de 27 pueden ordenarse, dado que el último límite visto en el stream s_2 tenía un valor de 27.

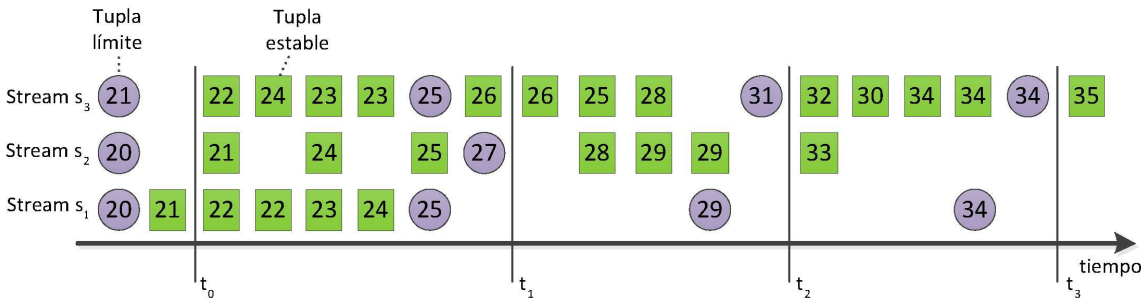


Figura 4.8: Los límites permiten ordenar las tuplas determinísticamente

Para que las réplicas puedan ordenar los pedidos determinísticamente se utiliza una técnica basada en el empleo de un autómata finito para replicación [Sch90], para ello sólo se requiere que los clientes efectúen pedidos periódicamente, posiblemente pedidos nulos. Las tuplas de límite cumplen el rol de pedidos nulos en un DSMS. Dado que las tuplas límite son periódicas se asegura el continuo y sostenido progreso aún en ausencia de datos reales en uno o más streams de entrada.

En lugar de modificar los operadores para ordenar las tuplas antes de procesarlas introduciremos `SUnion`, un simple operador de serialización que toma múltiples streams como entradas y las ordena determinísticamente en una única secuencia. El empleo de un operador separado permite que la lógica del ordenamiento se encuentre contenida dentro de un único operador. Además, `SUnion` gestiona los *trade-offs* entre disponibilidad y consistencia decidiendo cuando las tuplas deben ser procesadas, como discutiremos en detalle en las próximas secciones.

Para lograr mayor flexibilidad al seleccionar la función de ordenamiento y gestionar el *trade-off* disponibilidad-consistencia trabajando con una granularidad mayor, `SUnion` procesa tuplas según la granularidad propuesta mediante cubetas (*buckets*)

de tamaño fijo. `SUnion` utiliza el valor *stime* para ubicar tuplas en buckets de tamaño estáticamente preestablecido. Luego se emplean las tuplas límites para determinar cuando un bucket es estable (no arribarán más tuplas para ese bucket), en ese momento es seguro ordenar las tuplas en el bucket y enviarlas a la salida. La función de ordenamiento de `SUnion` usualmente ordena tuplas por valor creciente de *stime*, pero obviamente se podría emplear otro criterio. La Figura 4.9 ilustra como se utilizan los buckets para determinar cuáles son las próximas tuplas a procesar. En este ejemplo, solamente las tuplas del bucket i pueden ordenarse y enviarse como estables porque arribaron las tuplas límite cuya estampilla de tiempo es mayor que el límite del bucket (se encuentran en el bucket $i+1$). Estas tuplas límite hacen que el bucket i sea estable, pues garantizas que no falta ninguna tupla en este bucket. Por otro lado, ni el bucket $i+1$ ni el $i+1$ pueden ser procesados, dado que a ambos buckets les faltan las tuplas límite, haciendo posible que todavía lleguen tuplas para estos buckets.

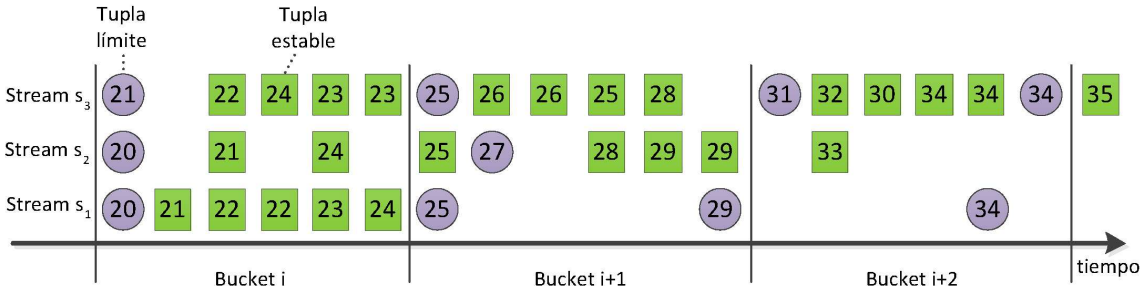
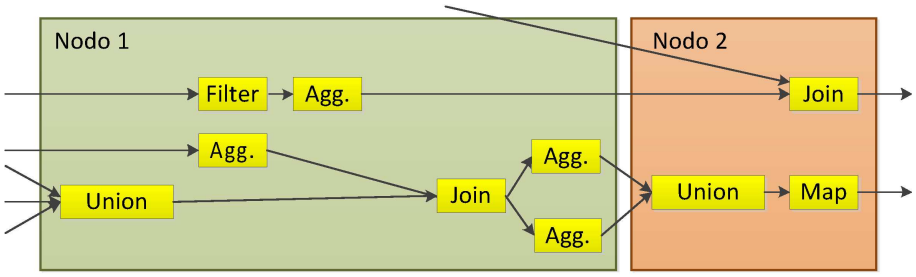


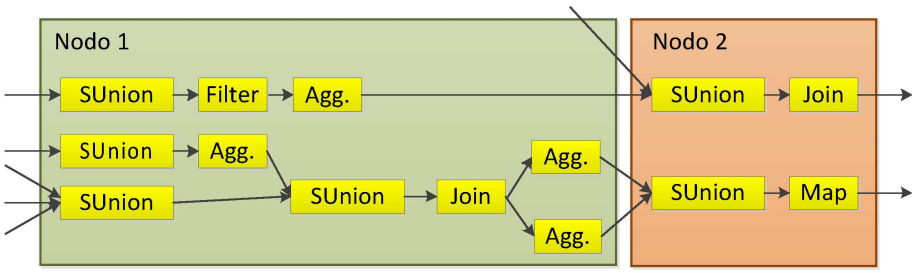
FIGURA 4.9: Ejemplo de tuplas organizadas mediante buckets para intervalo límite $d = 5$

Para mantener las réplicas consistentes debe existir un operador `SUnion` frente a cada operador con más de un stream de entrada. Las Figuras 4.10 (a) y (b) muestran un diagrama de consulta y su versión modificada, en el cual todos los operadores `Union` son reemplazados por `SUnion` y a su vez, estos últimos son ubicados antes de cada `Join`. `Union` y `Join` son los únicos operadores que tienen más de un stream de entrada.

La Figura 4.10 (b) muestra que los operadores `SUnion` pueden aparecer en cualquier ubicación en un diagrama de consulta. Por lo tanto, todos los operadores deben establecer en las tuplas de salida valores *stime* en forma determinística y producir tuplas límite periódicamente con valores *stime* monótonamente crecientes. Los operadores aseguran que el valor *stime* de las tuplas sea siempre determinista usando, por ejemplo, los valores *stime* de las tuplas de entrada para computar el *stime* de las tuplas de salida. Para que los operadores downstream produzcan tuplas límite correctas aún en ausencia de datos de entrada, o cuando las tuplas no están estrictamente ordenadas según sus valores *stime*, las tuplas límite deben propagarse por



(a) Diagrama de consulta inicial.



(b) Diagrama modificado para mantener la consistencia entre réplicas en ausencia de fallas; además posibilita el control entre disponibilidad y consistencia frente a fallas.

Figura 4.10: Ubicación de `SUnion` en un diagrama de consulta

el diagrama de consulta.

`SUnion` es similar al Gestor de Entradas de STREAM [SW04a], el cual ordena tuplas según el orden creciente de sus estampillas de tiempo. Sin embargo, `SUnion` asegura que las réplicas procesen las tuplas en el mismo orden. Los operadores `SUnion` deben aparecer antes que cualquier operador con más de un stream de entrada y no sólo en las entradas al sistema. `SUnion` es a su vez más general que el Gestor de Entradas, y no sólo porque este último no es tolerante a fallas, sino porque nuestro operador puede implementar diferentes funciones de serialización (sólo debe asegurarse de romper empates de manera determinística). `SUnion` asume que las demoras están limitadas y utiliza esta asunción para computar *heartbeats* si las aplicaciones no los proveen. Como discutiremos a continuación, el operador `SUnion` es el responsable de parametrizar el trade-off entre disponibilidad y consistencia.

4.8.2 Impacto de la selección del valor *stime*

La hora local en el origen de los datos constituye la selección natural del valor *stime* de las tuplas. Sincronizar los relojes de las fuentes de datos lograría que las tuplas se procesen aproximadamente en el mismo orden en el que fueron producidas.

El *Network Time Protocol* [Mil85], más conocido por su sigla NTP, es un protocolo estándar que se encuentra implementado virtualmente en cualquier computadora

actual y que permite sincronizar relojes con una precisión de 10ms. Es claro que utilizar relojes reales no es la única opción, de hecho cualquier atributo entero permite definir las ventanas de tiempo que delimitan el cómputo de los operadores. Cuando éste es el caso, los operadores también asumen que las tuplas de entrada se ordenar según dicho atributo. Emplear el mismo atributo para el *stime* de las tuplas y para las ventanas colabora con el cumplimiento del ordenamiento.

El operador `SUnion` demora las tuplas dado que debe bufferearlas y ordenarlas. Esta demora depende de tres propiedades de las tuplas límite. Primero, el intervalo entre tuplas límite con valores crecientes de *stime* y el tamaño del bucket determinan la demora básica de buffering. Segundo, la demora básica puede aumentar con el desorden, estando acotado superiormente por la máxima demora entre una tupla con *stime* t y una tupla límite con *stime* $> t$. Tercero, un bucket es estable únicamente cuando aparecen tuplas límite con *stime* suficientemente alto en todos los streams de entrada para el mismo `SUnion`.

La mayor diferencia entre valores *stime* entre estos streams limita la demora adicional. Dado que el diagrama de consulta usualmente asume que las tuplas están ordenadas según el atributo *stime* es esperable que las demoras originadas en el proceso de serialización sean pequeñas en la práctica. En particular, estas demoras deberían ser significativamente menores que la demora de procesamiento adicional máxima, X.

En resumen, la combinación de operadores `SUnion` y tuplas límite posibilita que las réplicas pertenecientes al mismo nodo de procesamiento procesen las tuplas en el mismo orden y se mantengan mutuamente consistentes. Las `SUnion`es incrementan la latencia de procesamiento porque bufferean tuplas antes de ordenarlas y procesarlas, pero este incremento en la demora es virtualmente despreciable.

4.8.3 Detección de fallas

La propiedad *heartbeat* de las tuplas límite posibilitan que el operador `SUnion` pueda distinguir entre falta de datos en un stream y una falla. Cuando ocurre una falla, un `SUnion` deja de recibir tuplas límite en uno o más de sus streams de entrada, pero puede seguir recibiendo tuplas tentativas. Es claro que no se deben propagar las tuplas tentativas al diagrama de consulta tan pronto como van arribando sino que se deben demorar según el requerimiento de disponibilidad. Para ello deberemos ubicar operadores `SUnion` en cada stream de entrada, aún cuando el stream constituya la única entrada de un operador. La Figura 4.10 (b) también ilustra la modificación necesaria en el diagrama de consulta para controlar la disponibilidad y la consistencia

cuando ocurren las fallas. En este diagrama vemos como se agrega un operador `SUnion` en cada stream de entrada.

En el estado ESTABLE, `SUnion` hace la mayor parte del trabajo. Sin embargo, otros componentes también participan; el DataPath bufferea las tuplas de salida más recientes y el Gestor de Consistencia monitorea los upstreams vecinos y sus réplicas. De hecho, además de contar con tuplas límite para detectar las fallas, el Gestor de Consistencia periódicamente solicita *heartbeats* de respuestas a cada réplica de cada uno de los vecinos upstream. De esta forma si un vecino upstream falla, el Gestor de Consistencia conoce el estado de cada réplica de ese dado vecino y por ende puede elegir utilizar otra réplica.

Los *heartbeats* no sólo indican si una réplica está disponible sino que además incluyen los estados (ESTABLE, FALLA_UPSTREAM, FALLA, o ESTABILIZACIÓN) de sus streams de salida. El Gestor de Consistencia utiliza esta información detallada al momento de seleccionar un vecino upstream. El algoritmo 4.1 es responsable del monitoreo de los streams de entrada. El Gestor de Consistencia periódicamente (cada P_2 unidades de tiempo) envía un mensaje a cada réplica de cada vecino upstream. Una vez que recibe el mensaje proveniente de una réplica r, el Gestor de Consistencia actualiza el último estado conocido para cada stream producido por r. Si hay más de P_1 pedidos sin responder, el Gestor de Consistencia asume que esa réplica falló. Los valores de los parámetros P_1 y P_2 constituyen el trade-off entre la demora en la detección de la falla y la sensibilidad del algoritmo que monitorea las fallas temporales (*transient*) que pueden provocar que un nodo no procese uno o más pedidos consecutivos. Valores posibles podrían ser $P_1 = 3$ y $P_2 = 100ms$.

Además de monitorear los streams de entrada, el Gestor de Consistencia tiene la responsabilidad de anunciar el estado correcto de todos los streams de salida. Para determinar el estado de un stream de salida, el operador `SOutput` podría enviar un mensaje al Gestor de Consistencia cada vez que detecta un cambio en el estado del stream. Sin embargo, existen dos desventajas en esta metodología: la primara es que desde el momento en que ocurre una falla en una entrada puede tomar tiempo hasta que las tuplas tentativas se propaguen hasta la salida. La segunda es que, al observar un stream, `SOutput` no sería capaz de distinguir entre una falla parcial (que resulta en tuplas tentativas) y una falla total que bloquea la salida. Por el contrario, el Gestor de Consistencia puede computar el estado de los streams de salida directamente a partir del estado de los streams de entrada. Presentamos a continuación dos posibles algoritmos.

Algoritmo 1. El algoritmo más simple consiste en poner el estado de todos los

streams de salida igual al estado del nodo de procesamiento. Definiremos el estado del nodo de procesamiento de la siguiente forma: si existe al menos un stream de entrada en el estado FALLA_UPSTREAM, entonces el nodo pasa al estado FALLA_UPSTREAM. El nodo permanece en este estado hasta que encuentre réplicas alternativas para los streams de entrada que fallaron o hasta que la falla se haya solucionado y el nodo comience a reconciliar su estado. Durante el estado de reconciliación (vuelta al estado normal), el nodo pasa al estado de ESTABILIZACIÓN. Luego de reconciliar su estado, si no existen nuevas fallas, el nodo vuelve al estado ESTABLE. De otro modo, se vuelve al estado, FALLA_UPSTREAM.

En el resto de este trabajo y por simplicidad de presentación, utilizaremos este algoritmo y siempre asignaremos el estado de los streams de salida a partir del estado del nodo. En este algoritmo Nodos denota el conjunto de todos los nodos de procesamiento participantes del sistema federado, Estados = { ESTABLE, FALLA_UPSTREAM, FALLA, ESTABILIZACIÓN } y constantemente se actualiza `EnEstado` (el estado de los streams de entrada producidos por cada réplica de cada vecino upstream). Por último, todas las estructuras de datos son locales a cada nodo.

```
// Envío de pedidos de heartbeats
Procedimiento Requerir_Estado
Entrada:
   Streams de entrada: conjunto de todos los streams de entrada del nodo.
   Réplicas: vecinos upstream y sus réplicas.
        ∀s ∈ StreamsEntrada, Réplicas[s] = {r1, r2, ..., rn} | ∀i ∈ [1,n], ri ∈ Nodos produce s.
Para la Entrada y la Salida:
   Pendientes: número de pedidos sin atender enviados a los nodos participantes.
       ∀r ∈ Nodos, Pendientes[r] = i | i es el número de pedidos sin atender enviados
            por este nodo a r.
   EnEstado: estados de los streams producidos por los nodos participantes.
      ∀s ∈ StreamsEntrada, ∀r ∈ Réplicas[s],
      EnEstado[r,s] = x | x ∈ Estados es el estado del stream s producido por el nodo r.

01. foreach r ∈ Réplicas
      // si existen más de P1 mensajes pendientes considerar
      // a todos los streams como fallados
02.   if Pendientes[r] > P1
03.       foreach s ∈ StreamsEntrada | r ∈ Réplicas[s]
04.         EnEstado[r,s] ← \textsf{FALLA}
05.   else
06.      enviar pedido de hearbeat a r
07.      Pendientes[r] ← Pendientes[r] +1
08. sleep durante P2

// Recibir una respuesta, Respuesta[r] de r
Procedimiento Recibir_Respuesta
Entrada:
   Respuesta[r]: respuesta al pedido de heartbeat proveniente de la réplica r.
        ∀s ∈ StreamsEntrada | r ∈ Réplicas[s]
```

```
      Respuesta[r][s] = x | x ∈ Estados es el estado del stream s producido por r.
Para la Entrada y la Salida:
   Pendientes: número de pedidos sin atender enviados a una réplica.
   EnEstado: estados de los streams producidos por los vecinos upstream
         y sus réplicas.

01. Pendientes[r] ← 0
02. foreach s ∈ Respuesta[r]
03.    EnEstado[r,s] ← Respuesta[r][s]
```

Algoritmo 4.1: Algoritmo para monitoreo de disponibilidad y consistencia de streams de entrada mediante monitoreo de todas las réplicas de los vecinos upstream

Algoritmo 2. El algoritmo anterior es simple pero únicamente es capaz de devolver información aproximada acerca del estado de los streams de salida. En muchos casos, aún si el nodo está en el estado FALLA_UPSTREAM (de acuerdo con el algoritmo 1), un subconjunto de sus salidas puede no estar afectado por la falla y puede permanecer en el estado ESTABLE. También podemos distinguir entre el estado FALLA_UPSTREAM, donde el stream de salida puede producir tuplas tentativas y el estado de FALLA, donde el stream de salida se bloquea o permanece inaccesible. Esta distinción mejora la selección de la réplica para los vecinos downstream. El Apéndice A presenta el algoritmo detallado para computar el estado de los streams de salida.

4.9 Estado FALLA_UPSTREAM

En esta sección presentamos los algoritmos empleados por cada nodo para manejar las fallas de sus vecinos upstream de acuerdo con la disponibilidad requerida por la aplicación y manteniendo a su vez una consistencia eventual. Estos algoritmos constan de dos partes: cambiar de una réplica a la otra cuando ocurre una falla, y suspender o demorar el procesamiento de nuevas tuplas para reducir inconsistencias.

4.9.1 Cambio de vecinos upstream

Dado que el Gestor de Consistencia monitorea los streams de entrada constantemente, tan pronto como detecta que un vecino upstream ya no se encuentra en el estado ESTABLE, *i.e.,* se encuentra inalcanzable o experimentando una falla, el nodo puede cambiar a otra réplica ESTABLE de ese vecino. Este cambio de réplica permite que el nodo pueda mantener su disponibilidad y consistencia a pesar de la falla. Cuando un nodo cambia réplicas de un vecino upstream es necesario lograr que esta nueva réplica continúe con el envío de datos desde el punto correcto en el upstream; para ello debe indicar cuál fue la última tupla estable que recibió y

si recibió tuplas tentativas luego de las marcadas como estables. Esta información es provista al Gestor de Consistencia mediante el DataPath, quien envía un mensaje de subscripción al nuevo vecino upstream. Este nuevo vecino upstream puede entonces responder a las tuplas que previamente faltaban o incluso corregir tuplas previamente marcadas como tentativas. Claramente, los DataPaths de los vecinos upstreams deben bufferear sus tuplas de salida para poder lograr este tipo de *replays* y correcciones. Discutiremos la gestión de buffers en la Sección 4.12.

Si el Gestor de Consistencia no puede encontrar una réplica ESTABLE para reemplazar a un vecino upstream, al menos debe tratar de conectarse a una réplica que se encuentre en el estado FALLA_UPSTREAM debido a que procesar tuplas de una réplica con estas características colabora en mantener la disponibilidad en el nodo. Si el Gestor de Consistencia no puede encontrar ninguna réplica en el estado ESTABLE o FALLA_UPSTREAM, entonces el nodo no puede mantener la disponibilidad del stream que falla. Si se conecta con una réplica en el estado ESTABILIZACIÓN se logra que el nodo por lo menos pueda empezar a corregir los datos provenientes del stream fallado. La Tabla 4.2 presenta el algoritmo que emplean los nodos para cambiar de una réplica upstream a otra. En este algoritmo un nodo simplemente prefiere vecinos upstreams en el estado ESTABLE y luego aquellos en FALLA_UPSTREAM antes que en el resto de los estados. Réplicas(s) es el conjunto de réplicas que producen el stream s y Actual(s) es el vecino upstream actual de s. El estado de Actual(s) y los estados de los nodos pertenecientes a Réplicas(s) definen las condiciones que pueden disparar un cambio de vecino upstream. Estos cambios, junto con la ocurrencia de fallas (o solución de las mismas) a su vez provocan el cambio en Actual(s). Como muestra la Figura 4.11, el resultado de estos cambios es que cualquier réplica puede reenviar streams de datos hacia cualquier réplica downstream o cliente y las salidas de algunas réplicas pueden incluso no ser empleadas. En dicha figura R_{ij} constituye la $j$ava réplica del nodo de procesamiento i. En la Sección 4.10.3 refinaremos este algoritmo luego de presentar los detalles del estado ESTABILIZACIÓN.

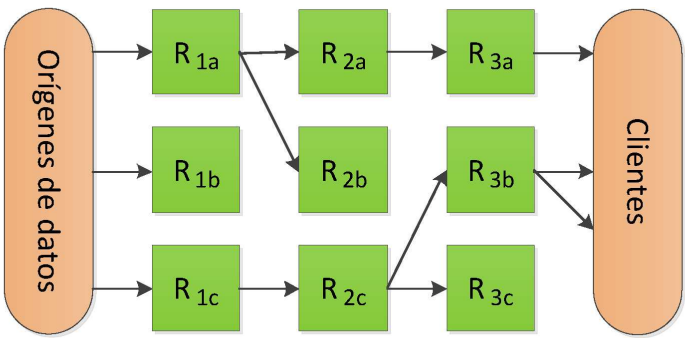


FIGURA 4.11: Ejemplo de un DSMS distribuido y replicado

Estado	Condición: EnEstado(Actual(s), s)	Condición: R = Réplicas(s) − Actual(s)	Acción
1	ESTABLE	–	Permanecer en Estado 1
2	! ESTABLE	$\exists r \in R$, EnEstado(Actual(r),s) = ESTABLE	Dessubscribirse de Actual(s) Actual(s) $\leftarrow r$ Subscribirse a Actual(s) Ir al Estado 1
3	FALLA_UPSTREAM	$\nexists r \in R$, EnEstado(Actual(r),s) = ESTABLE	Permanecer en Estado 3
4	$\in$ { FALLA, ESTABILIZACIÓN }	$\nexists r \in R$, EnEstado(Actual(r),s) = ESTABLE y $\exists r\prime \in R$, EnEstado(Actual(r$\prime$),s) = FALLA_UPSTREAM	Dessubscribirse de Actual(s) Actual(s) $\leftarrow r\prime$ Subscribirse a Actual(s) Ir al Estado 3
5	$\in$ { FALLA, ESTABILIZACIÓN }	$\nexists r \in R$, EnEstado(Actual(r),s) = ESTABLE y $\exists r\prime \in R$, EnEstado(Actual(r$\prime$),s) = FALLA_UPSTREAM	Permanecer en Estado 5

Tabla 4.2: Algoritmo para cambio de réplica de un vecino upstream para mantener disponibilidad

4.9.2 Gestión de la disponibilidad y la consistencia en presencia de fallas

Si un nodo no puede encontrar una réplica upstream que pueda proveerle de las tuplas estables más recientes en un stream, entonces el nodo debe decidir entre suspender el procesamiento de nuevas tuplas mientras dure la falla o puede seguir procesando las (posiblemente tentativas) entradas que permanecen disponibles.

Suspender el procesamiento evita las inconsistencias, por lo tanto es la mejor opción para hacer frente a fallas de corta duración. Para las fallas de larga duración el nodo debe eventualmente procesar las tuplas de entrada más recientes para asegurar la disponibilidad requerida. Si un nodo procesa tuplas tentativas durante una falla, ya sea recibiendo tuplas tentativas o procediendo con entradas faltantes, su estado puede empezar a divergir con respecto a otras réplicas.

Para minimizar las inconsistencias mientras se mantiene la disponibilidad requerida un nodo puede demorar continuamente las nuevas tuplas, mientras no sobrepase su latencia máxima de procesamiento incremental (predefinida). Demorar las nuevas tuplas reduce el número de tuplas tentativas producidas durante la falla, pero procesarlas mientras van arribando permite al nodo demorar o suspender el procesamiento de las nuevas tuplas durante la ESTABILIZACIÓN. En el próximo capítulo discutiremos los trade-offs entre el procesamiento de las tuplas tentativas con o sin demora. A continuación nos limitaremos a presentar únicamente los mecanismos que controlan la disponibilidad y la consistencia.

Los operadores `SUnion` gestionan el trade-off entre disponibilidad y consistencia suspendiendo o demorando las tuplas en presencia de fallas. Más específicamente,

la aplicación cliente especifica la latencia de procesamiento incremental total, X, la cual se divide entre los operadores `SUnion`[7]. A cada stream de entrada de cada operador `SUnion` se le asocia una latencia máxima D. En tiempo de ejecución, cuando recibe la primer tupla de un dado bucket de un stream `SUnion` arranca un *timer*, si el timer expira antes de que las tuplas límite puedan asegurar que el bucket completo esté estable `SUnion` serializa las tuplas disponibles etiquetándolas como TENTATIVAs y buffereándolas para su posterior participación el el proceso de reconciliación. En el ejemplo de la Figura 4.8 podemos ver que si el límite para el stream s_2 no arriba antes de que transcurran D unidades de tiempo a partir de que la primera tupla entra en el bucket $i+1$ `SUnion` reenviará las tuplas restantes desde ese bucket como tentativas. Adicionalmente efectuará el buffering de estas tuplas como preparativo para el posterior proceso de reconciliación. En el estado FALLA_UPSTREAM, los `SUnion`s y el resto de los operadores deben seguir procesando y produciendo tuplas límites, pero estas tuplas también deben marcarse como TENTATIVAs. Estos límites tentativos pueden colaborar con un `SUnion` downstream para determinar qué tan pronto puede procesar un bucket tentativo.

Para asegurar que un nodo tenga tiempo de detectar y reaccionar frente a una falla upstream antes de que `SUnion` comience a procesar sus entradas como tentativas, los parámetros del algoritmo 4.1 deben satisfacer: $P_2 * P_1 << D$, donde P_2 es el intervalo de tiempo entre pedidos para la información de estado y P_1 es el número de pedidos consecutivos sin responder que deben ocurrir antes de que el nodo downstream declare que el vecino upstream falló.

En resumen, cuando un stream de entrada falla, un nodo pasa al estado FALLA_UPSTREAM y usa la información previamente recolectada para encontrar y continuar procesando a partir de la mejor réplica disponible de dicho stream. Si no existe una réplica en el estado ESTABLE, entonces el nodo debe continuar procesando las (posiblemente tentativas) entradas que se encuentran disponibles para lograr mantener la baja latencia de procesamiento. Sin embargo, los operadores `SUnion` pueden suspender o demorar las tuplas más recientes para minimizar las inconsistencias. Luego de solucionada la falla, el nodo pasa al estado ESTABILIZACIÓN, el cual será discutido a continuación.

[7]En el próximo capítulo analizaremos como se divide X entre los operadores `SUnion`.

4.10 Estado ESTABILIZACIÓN

En esta sección presentaremos los algoritmos que sigue cada nodo luego de que una falla se soluciona. Para simplificar este planteo nos enfocaremos en la recuperación de una única falla, sin embargo DCAA soporta fallas múltiples simultáneas, así como también fallas durante la recuperación. Estos escenarios más complejos serán discutidos en la Sección 4.14.

Un operador `SUnion` determina que una falla fue solucionada cuando recibe correcciones a tuplas previamente marcadas como tentativas o una repetición de tuplas provenientes de streams de entrada que anteriormente faltaban. Las correcciones arriban en la forma de una única tupla `DESHACER` seguida de tuplas estables. Cuando se recibe una tupla `DESHACER`, `SUnion` estabiliza los correspondientes streams de entrada reemplazando en su buffer las tuplas inestables por sus contrapartidas estables. Una vez que se corrige y estabilizan las tuplas de entrada de al menos un bucket, `SUnion` notifica al Gestor de Consistencia que se encuentra listo para el estado de reconciliación.

Para evitar corregir tuplas tentativas con otras tuplas tentativas cuando solamente uno de varios streams ha sido solucionado, el Gestor de Consistencia espera la notificación por parte de todos los `SUnion`es que se encontraban en las entradas que habían previamente fallado antes de entrar en el estado ESTABILIZACIÓN.

Para asegurar la consistencia eventual, estando en el estado ESTABILIZACIÓN, un nodo que procesó tuplas tentativas debe reconciliar su estado y estabilizar sus salidas, reemplazando las tuplas tentativas de salida por sus contrapartidas estables. La estabilización de los streams de salida permite a los vecinos downstream reconciliar sus respectivos estados por turnos. En esta sección presentaremos las técnicas para reconciliar estados y estabilizar las salidas. También describiremos cómo cada nodo mantiene su disponibilidad mientras reconcilia su estado.

4.10.1 Reconciliación del estado de un nodo

El estado de un nodo depende de la secuencia exacta de tuplas que procesa y dado que puede suceder que ninguna réplica tenga el estado correcto luego de una falla, proponemos la reconciliación del estado de un nodo a partir de una técnica que revierte el estado a uno anterior, pre-falla, y reprocesa todas las tuplas de entrada que arribaron desde ese momento. Para volver a un estado anterior exploramos dos alternativas: revertir a un estado generado mediante un *checkpoint*, o deshacer los efectos provocados por tuplas tentativas. Ambas alternativas requieren que el nodo suspenda el procesamiento de nuevas tuplas mientras reconcilia su estado.

En este capítulo, por cuestiones de claridad, presentaremos únicamente el esquema *checkpoint/redo*, dejando la técnica basada en *undo* para el Apéndice B.

Bajo el esquema *checkpoint/redo*, un nodo periódicamente realiza checkpoints del estado de sus diagramas de consulta mientras se encuentra en el estado ESTABLE. El Gestor de Consistencia determina cuándo el nodo debe realizar el checkpoint de su estado.

Para realizar el checkpoint, el nodo suspende el procesamiento de cualquier tupla e itera por todos los operadores y colas intermedias haciendo copias de sus estados. Para efectuar este procedimiento, los operadores deben extenderse con un método que permita una *snapshot* de su estado. Adicionalmente estos checkpoints podrían optimizarse mediante la generación de diffs a partir del último, en lugar de generar checkpoints completos.

Como arriba mencionamos, el Gestor de Consistencia determina cuándo el nodo debe realizar el checkpoint de su estado. Para reconciliar su estado, un nodo continúa a partir del último checkpoint antes de la falla y reprocesa todas las tuplas que recibió desde ese momento. Para reiniciar su estado desde el checkpoint, el nodo debe suspender el procesamiento de todas las tuplas e iterar por todos los operadores y colas intermedias reiniciando sus estados a partir del checkpoint. Los operadores deben por lo tanto modificarse para incluir un método que permita reiniciar su estado a partir de un checkpoint. Luego de reiniciar su estado, el nodo reprocesa todas las tuplas de entradas recibidas luego del checkpoint. Para implementar estas repeticiones, los operadores `SUnion` en los streams de entrada deben bufferear las tuplas de entrada (antes, durante y después de la falla) entre checkpoints. Sin embargo, cuando ocurre un ckeckpoint, los operadores `SUnion` truncan todos los buckets que fueron procesados antes del checkpoint.

4.10.2 Estabilización de los streams de salida

Independientemente del método elegido para reconciliar el estado, un nodo estabiliza cada stream de salida borrando un sufijo de un stream con una única tupla `DESHACER` y redirigiendo las correcciones en forma de tuplas estables. Un nodo debe deshacer y corregir todas las tuplas tentativas producidas durante una falla. Una posible optimización es la siguiente: el nodo podría determinar si un prefijo de tuplas tentativas no fue afectado por la falla, en este caso entonces no necesita ser corregido. Sin embargo, como discutiremos en la Sección 4.12, planificar corregir siempre todas las tuplas tentativas ahorra espacio de buffer.

Contando con un esquema de reconciliación *undo/redo*, los operadores procesan y

producen tuplas DESHACER, las cuales simplemente se propagan hacia los streams de salida. Para generar una tupla DESHACER con checkpoint/redo introducimos un nuevo operador de "serialización de salida", SOutput, que será ubicado en cada stream de salida que cruza el límite de un nodo. En tiempo de ejecución, SOutput actúa como filtro *pass-through* que a su vez "recuerda" la última tupla estable que produjo. Luego de reiniciar desde un checkpoint, SOutput, descarta las tuplas estables duplicadas y produce la tupla DESHACER. Alternativamente el DataPath podría monitorear el stream de salida y producir la tupla DESHACER apropiada en lugar de SOutput. Sin embargo, SOutput, encapsula prolijamente al (pequeño) autómata finito necesario para borrar apropiadamente las tuplas duplicadas frente a variados y posiblemente complejos escenarios de falla y recuperación.

Antes de enviar el DESHACER downstream, el DataPath debe buscar el identificador de la última tupla estable que recibió cada vecino downstream. De hecho, cada nodo puede cambiar sus vecinos upstream en cualquier momento antes o durante una falla. Un vecino upstream debe entonces fijarse qué tuplas esta corrigiendo para cada vecino downstream. Primero, el DataPath corrige tuplas en los buffers de salida de forma tal que los mismos contengan solamente tuplas estables. Luego, el DataPath busca la última tupla estable recibida por cada vecino downstream y envía el DESHACER y correcciones apropiados.

El proceso de estabilización se completa cuando el nodo reprocesa todas las tuplas previamente marcadas como tentativas y se pone al día con la ejecución normal (*i.e.*,, limpia sus colas) o cuando ocurre otra falla y el nodo vuelve al estado FALLA_UPSTREAM. Una vez que la estabilización termina el nodo transmite una tupla FIN_REC hacia sus nodos downstream. Los SUnionees ubicados en los streams de entradas producen tuplas FIN_REC una vez que limpian sus buffers de entradas. Las tuplas FIN_REC se propagan por el diagrama de consulta hacia los operadores SOutput, los cuales pasan estas tuplas downstream. Para evitar tuplas FIN_REC repetidas, cada operador SUnion ubicado en medio del diagrama espera una tupla FIN_REC en cada una de sus entradas antes de enviar una (única) FIN_REC downstream.

Los algoritmos anteriores para la reconciliación del estado y para la estabilización de los streams de salida posibilitan que un dado nodo pueda asegurar consistencia eventual: el estado del nodo vuelve a ser consistente y los vecinos downstream reciben los streams de salida en forma completa y correcta. El problema, sin embargo, es que el proceso de estabilización toma tiempo y mientras se reconcilian estados y corrigen salidas el nodo no procesa nuevas tuplas de entrada. Una prolongada demora en la estabilización potencialmente causará que el sistema no pueda cumplir con el

requerimiento de disponibilidad. A continuación analizaremos cómo solucionar este problema.

4.10.3 Procesamiento de nuevas tuplas durante la reconciliación

Durante la estabilización nuevamente nos enfrentamos al trade-off entre disponibilidad y consistencia. Suspender nuevas tuplas durante el estado de reconciliación reduce el número de tuplas tentativas, pero puede eventualmente violar el requerimiento de disponibilidad si la reconciliación dura demasiado tiempo. Para mantener la disponibilidad cuando se está efectuando la reconciliación luego de una falla prolongada, un nodo debe producir tuplas estables corregidas y nuevas tuplas tentativas. Para ello, DCAA emplea dos réplicas de un diagrama de consulta: una réplica se mantiene en el estado FALLA_UPSTREAM y continúa procesando nuevas tuplas de entrada mientras que la otra réplica realiza la reconciliación. Un nodo podría correr ambas versiones localmente, pero DCAA ya emplea replicación y por lo tanto los clientes downstream deben saber de la existencia de todas las réplicas para poder manejar adecuadamente su disponibilidad y consistencia. Como consecuencia de ello, y para evitar duplicar el número de réplicas, proponemos que siempre que sea posible las réplicas se utilicen unas a otras como las dos versiones.

Para lograr que un par de réplicas pueda decidir cuál de ellas debe reconciliar su estado mientras la otra se mantiene disponible, proponemos que los Gestores de Consistencia ejecuten el siguiente protocolo sencillo de comunicación inter-réplicas. Antes de entrar en ESTABILIZACIÓN, el Gestor de Consistencia envía un mensaje a una de sus réplicas elegidas aleatoriamente. Mediante este mensaje se solicita autorización para entrar en el estado ESTABILIZACIÓN. Si el compañero otorga esta autorización, significa que se compromete a no entrar en ESTABILIZACIÓN (asegurando exclusión mutua). Una vez recibida la autorización para reconciliar, el Gestor de Consistencia dispara la reconciliación de estado. Sin embargo, si la réplica rechaza el pedido de autorización entonces el nodo no puede entrar en el estado ESTABILIZACIÓN y el Gestor de Consistencia espera un período corto de tiempo y vuelve a solicitar la autorización, posiblemente a una réplica diferente. Un Gestor de Consistencia siempre acepta los pedidos de reconciliación provenientes de sus réplicas, excepto si ya se encuentra en el estado de ESTABILIZACIÓN o si necesita reconciliar su propio estado y su identificador es menor que el del nodo que hace el pedido. Esta última condición funciona como *tie breaker* cuando múltiples nodos necesitan reconciliar sus estados al mismo tiempo. La Figura 4.12 ilustra la

comunicación que tiene lugar entre dos réplicas que necesitan reconciliar sus estados.

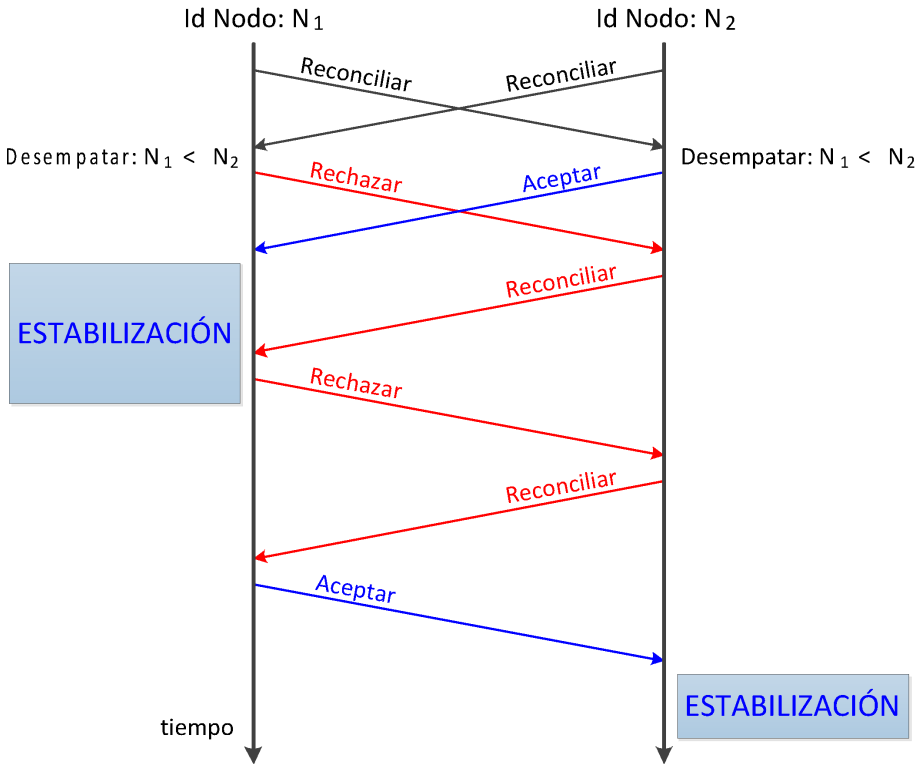


Figura 4.12: Protocolo de comunicación inter-réplica y el estado de ESTABILIZACIÓN

El procesamiento de nuevas tuplas durante la ESTABILIZACIÓN incrementa el número de tuplas tentativas, comparado con suspender su procesamiento. Un nodo puede todavía tratar de reducir las inconsistencias demorando las nuevas tuplas tanto como sea posible. En el Apéndice C compararemos las alternativas de suspender, demorar, o procesar nuevas tuplas sin demoras durante la ESTABILIZACIÓN.

Cada nodo downstream es responsable de detectar cuándo uno de sus vecinos upstream pasa al estado ESTABILIZACIÓN y deja de producir tuplas nuevas para pasar a generar correcciones. Los nodos downstream detectan esta situación mediante *heartbeats* explícitos y porque empiezan a recibir correcciones. En el algoritmo de cambio de réplica de la Tabla 4.2, podemos ver que si uno de los vecinos upstream pasa al estado de ESTABILIZACIÓN, el nodo cambia a una réplica que todavía este produciendo tuplas tentativas hasta que encuentre una réplica que finalmente esté en el estado ESTABLE. En ese momento el nodo vuelva a utilizar la réplica ESTABLE.

4.10.4 Correcciones en background de tuplas de entrada

Luego de solucionada una falla, cuando el nodo finalmente vuelve a una réplica ESTABLE de un vecino upstream, indica dos elementos: el identificador de la última

tupla estable que recibió y un flag para señalar si procesó o no tuplas luego de la última tupla estable. Esta información permite que el nuevo vecino upstream ESTABLE envié correcciones de las tuplas tentativas antes de enviar las tuplas estables más recientes. Si la falla fue prolongada, el sólo envío de las correcciones puede tomar un tiempo considerable y puede causar que el nodo downstream no pueda cumplir con su requerimiento de disponibilidad.

Una posible solución a este problema podría ser que el nodo entre en el estado de ESTABILIZACIÓN antes de cambiar a una réplica ESTABLE y que solicite las correcciones a las entradas tentativas. El problema con este enfoque es que sería necesario que el nodo se comunicase simultáneamente con las réplicas ESTABLEs de todos sus vecinos upstream para luego poder entrar en el estado ESTABILIZACIÓN. Para evitar este requerimiento y también para acelerar la reconciliación de una cadena de nodos, decidimos dejar que los nodos downstream corrijan sus entradas en background (segundo plano), mientras siguen procesando las tuplas más recientes.

Un nodo realiza correcciones en background sobre un stream de entrada conectándose a una segunda réplica que se encuentra en el estado ESTABILIZACIÓN mientras a su vez permanece conectado a una réplica que está en FALLA_UPSTREAM. A partir de estos dos streams de entrada, los operadores SUnion reciben tuplas tentativas y estables en paralelo. Para poder hacer las correcciones en background, SUnion considera que las tuplas tentativas que recibe entre un DESHACER y un FIN_REC son parte de una falla que está actualmente sucediendo y por ende las tuplas estables son correcciones o tuplas previamente marcadas tentativas. Estas tuplas tentativas que aparecen luego de un FIN_REC pertenecen a una nueva falla. El DataPath asegura estas semánticas monitoreando las tuplas que están entrando en el DSMS y desconectando las entradas tentativas tan pronto como aparezca una tupla FIN_REC en el stream.

La Tabla 4.3 muestra el algoritmo extendido para el cambio de vecinos upstream usando un vecino Actual(s) para obtener las entradas más recientes y Background(s) para obtener las correcciones que se hicieron en background. La idea básica del algoritmo es que el nodo elija su vecino actual según el algoritmo anterior (Tabla 4.2). La réplica elegida es la mejor réplica desde el punto de vista de la disponibilidad. Dada la opción de Actual(s), el nodo elige una réplica para que sirva como Background(s)[8]. Si Actual(s) está en FALLA_UPSTREAM, el nodo se conecta (en caso de existir) a un vecino Background(s) en el estado ESTABILIZACIÓN.

Para poder implementar estas correcciones en background es necesario un cambio

[8]Por cuestiones de presentación en la Tabla 4.3 nos referiremos a Background(s) como Bg(s).

en el algoritmo de monitoreo del nodo upstream (Algoritmo 4.1). Si un nodo permanece en el estado FALLA_UPSTREAM por más de P_3 unidades de tiempo, entonces el Gestor de Consistencia debe asumir que todos los vecinos upstream ESTABLEs en realidad están en el estado ESTABILIZACIÓN. De hecho, si el nodo se conecta a una de estas réplicas ESTABLEs tomará tiempo recibir y procesar las repeticiones (*replays*) o las correcciones. Como arriba mencionamos, una vez que un nodo upstream termina de enviar todas las correcciones y se pone al día con la ejecución actual, envía una tupla FIN_REC. Tan pronto como el DataPath en el nodo downstream recibe esta tupla se desconecta del stream background asegurando que no existan tuplas tentativas que sigan a la de FIN_REC. A continuación el Gestor de Consistencia marca nuevamente al correspondiente vecino upstream como ESTABLE.

Discutiremos más profundamente el cambio de vecinos upstream en los distintos estados de consistencia en la Sección 4.14, donde analizaremos las propiedades de DCAA.

En resumen, una vez que la falla se soluciona, para asegurar la consistencia eventual un nodo pasa al estado ESTABILIZACIÓN cuando reconcilia su estado y corrige la salida que produjo durante la falla. Para mantener la disponibilidad mientras se efectúan estas correcciones, el nodo corre un protocolo de comunicación inter-réplica para lograr su reconciliación con el resto de las réplicas. Dado el potencial elevado número de tuplas que se deben corregir en un stream, un nodo debe corregir sus entradas en background mientras continúa procesando los datos tentativos más recientes. Luego de corregir su estado y ponerse al día con la ejecución actual, si no se producen nuevas fallas durante la ESTABILIZACIÓN, el nodo pasa al estado ESTABLE. De otra forma, vuelva al estado FALLA_UPSTREAM.

4.11 Recuperación del nodo fallado

En las secciones anteriores describimos las fallas y su recuperación cuando un nodo pierde temporalmente uno o más de sus streams de entrada. Independientemente de estas fallas, un nodo puede también fallar (*crash*). Un nodo que falla reinicia con un estado vacío que debe llevarse a un estado consistente antes de considerarse a sí mismo ESTABLE nuevamente. Cuando un nodo se recupera, no debe responder a ningún pedido (incluyendo los pedidos heartbeats) hasta que alcanza nuevamente el estado ESTABLE.

La reconstrucción del estado de un nodo caído puede hacerse de dos formas, dependiendo del tipo de los operadores presentes en el diagrama de consultas. El estado de los operadores capaces de converger depende solamente de una ventana

finita de tuplas de entrada y es actualizada de forma tal que siempre converge hacia un estado consistente anterior. Si un diagrama de consultas consiste solamente de operadores de este tipo, el estado del nodo siempre convergerá hacia un estado consistente luego de que el nodo procese suficientes tuplas de entrada. Para reconstruir un estado consistente, un nodo debe simplemente procesar tuplas y monitorear su progreso a través de los operadores de estado. Una vez que el primer conjunto de tuplas procesadas ya no afecta el estado de ningún operador, un nodo puede reprocesar algunas de las tuplas ya buffereadas en los nodos upstream.

Si los operadores son determinísticos pero no capaces de converger entonces sus estados pueden depender de tuplas de entrada arbitrariamente viejas. Para reconstruir un estado consistente en este caso un nodo debe procesar todas las tuplas de entrada que alguna vez procesó o debe obtener una copia del estado consistente actual. Dado que asumimos que existe al menos una réplica de cada nodo de procesamiento que mantiene el estado consistente actual todo el tiempo (*i.e.*,, a lo sumo $R - 1$ réplicas de R fallan al mismo tiempo), cuando un nodo se recupera, siempre podrá solicitar una copia del estado actual a alguna de las otras réplicas. Esta técnica de recuperación es similar a la propuesta por Shah *et.al.*en [BHS09].

4.12 Gestión del buffer

En las secciones previas presentamos los detalles del protocolo DCAA de tolerancia a fallas. Para el correcto funcionamiento del protocolo es necesario que los DataPaths buffereen las tuplas de salida y que los SUnions buffereen las tuplas de entrada. En la Sección 4.10.1 discutimos como los SUnions pueden truncar sus buffers luego de cada checkpoint. Sin embargo, por el momento asumiremos que los buffers de salida pueden crecer indefinidamente. Aún si las tuplas viejas se guardan a disco, no es deseable que los buffers crezcan sin límites.

A continuación presentaremos los algoritmos que gestionan estos buffers de entrada y de salida. En primer lugar veremos cuáles tuplas y dónde deben bufferearse para que el protocolo funcione. Luego analizaremos cómo la corrección de todas las tuplas tentativas luego de una falla (aún aquellas que no cambiaron) reduce significativamente los requerimientos de buffering. Presentaremos un algoritmo simple para truncar periódicamente los buffers de salida en ausencia de fallas y cómo el sistema puede manejar fallas de larga duración empleando únicamente espacio limitado de buffers para consultas capaces de converger.

4.12.1 Requerimientos de buffering

Al comienzo de esta sección mencionamos que los DataPaths deben bufferear las tuplas de salida y que los SUnions deben bufferear las tuplas de entrada. A continuación veremos cuándo es necesario cada tipo de buffer.

Buffers de salida: un nodo debe bufferear las tuplas de salida que produce hasta que todas las réplicas de todos los vecinos downstream reciban estas tuplas. De hecho, en cualquier instante de tiempo, cualquier réplica de un vecino downstream puede conectarse con cualquier réplica de un vecino upstream y pedirle todas las tuplas de entrada que todavía no recibió. Los buffers de salida son necesarios durante las fallas y los nodos upstream deben bufferear tuplas mientras los nodos downstream no puedan recibirlas.

Los buffers de salida también son necesarios en ausencia de fallas porque los nodos no pueden procesar y transmitir datos al mismo tiempo. Asumimos que los nodos tienen capacidades de procesamiento y ancho de banda separados (ver Sección 4.4.2). De esta forma los nodos puede mantener el throughput en las entradas sin atrasarse. Sin embargo, es posible que algunos nodos se adelanten respecto a otros; estos nodos deben bufferear sus tuplas de salida hasta que los otros nodos (atrasados respecto a ellos) produzcan las mismas tuplas de salida y las envíen downstream. Las Figuras 4.13 y 4.14 ilustran la necesidad del buffering de las tuplas de salida. En el ejemplo, el Nodo 1 está adelantado respecto de su réplica Nodo 1' y ya produjo la tupla cuyo *id* es 50, mientras que el Nodo 1' sólo produjo la tupla 30. Si el Nodo 1' falla, el Nodo 2' solicitará al Nodo 1 todas las tuplas desde la tupla 30 en adelante. Estas tuplas deben ser de todas formas buffereadas en el Nodo 1.

Buffers de entrada: cuando ocurre una falla los nodos deben bufferear las tuplas de entrada que reciben para reprocesarlas luego durante la ESTABILIZACIÓN. La cantidad de buffering necesaria depende de la técnica empleada para la reconciliación de estados. En la Figura 4.15 y bajo la metodología checkpoint/redo, los SUnions ubicados en los streams de entrada necesitan bufferear las tuplas que reciben entre checkpoints, dado que la reconciliación involucra recomenzar desde el último checkpoint y reprocesar todas las tuplas de entrada que se recibieron desde allí. En dicha figura se recuadran los operadores que bufferean tuplas. Los requerimientos de buffering empleando undo/redo son tratados en el Apéndice B.

4.12.2 Evitando el buffering de tuplas tentativas

Cuando ocurren fallas prolongadas los nodos producen cantidades significativas de tuplas tentativas que luego deberán ser corregidas. El primer paso en la gestión de

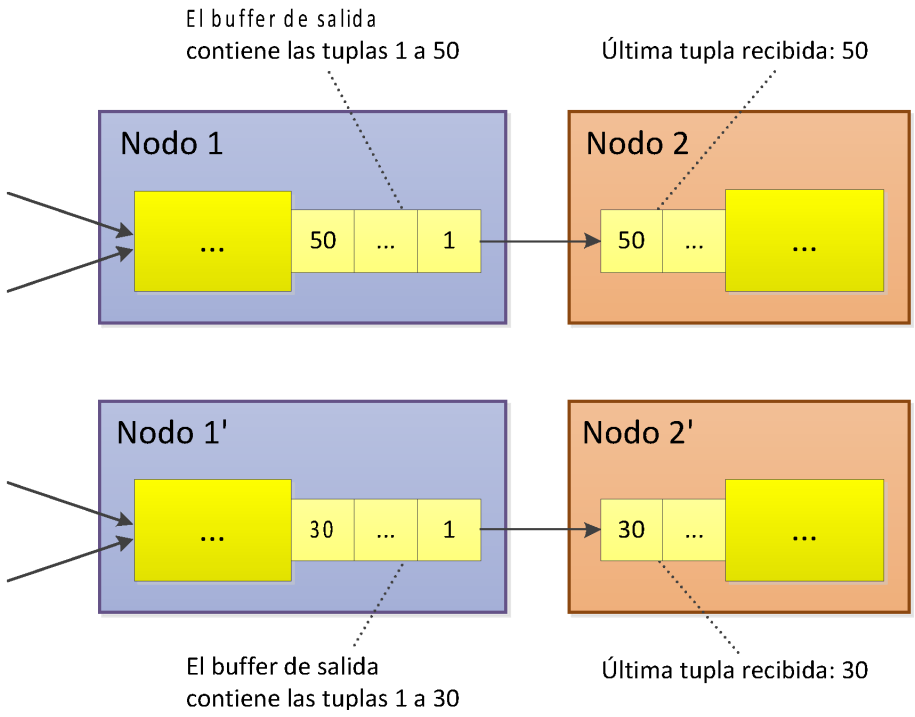


Figura 4.13: Ejemplo de uso de buffers de salida, sin falla de nodo

buffers es decidir si los nodos necesitan o no bufferear dichas tuplas.

Luego de que la falla se soluciona los nodos estabilizan sus salidas. Sin embargo, frecuentemente las tuplas tentativas más antiguas que se produjeron al principio de la falla son reemplazadas por idénticas tuplas estables. Esto puede ocurrir por ejemplo, cuando los buckets tentativos más antiguos de los streams de entrada ya recibieron todos sus datos y sólo les faltaban las tuplas límite. El modelo de datos de DCAA sumado a sus protocolos posibilitan únicamente la corrección de las tuplas sufijas tentativas que cambiaron. Solucionar esta limitación implica que se buffereen tuplas tentativas en los streams de salida y en los nodos downstream, dado que cualquier subconjunto de ellas no podría corregirse.

En contraste, si los nodos corrigen todas las tuplas tentativas, entonces estas no necesitarían ser buffereadas, con el consiguiente ahorro de espacio de buffer:

1. El DataPath todavía debe bufferear tuplas de salida hasta que todas las réplicas de todos los vecinos downstream puedan recibir dichas tuplas. Sin embargo, sólo necesita bufferear tuplas estables, pues las tuplas tentativas se descartan luego de ser enviadas.

2. Bajo la técnica de checkpoint/redo, los SUnions ubicados en los streams de entrada deben bufferear únicamente las tuplas estables producidas luego del último checkpoint antes de la falla. Ver Figura 4.16; nuevamente se recuadran

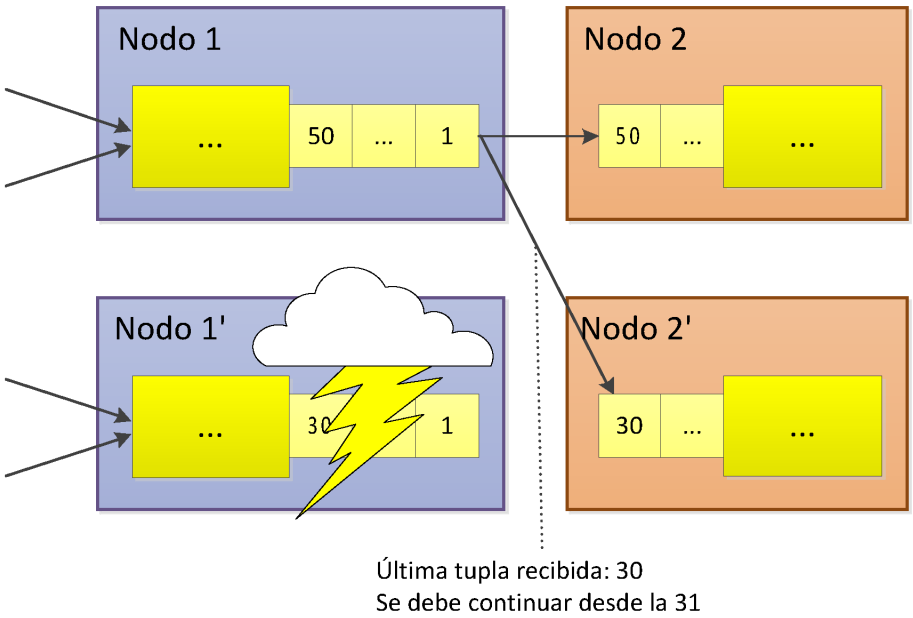


Figura 4.14: Ejemplo de uso de buffers de salida, con falla de nodo

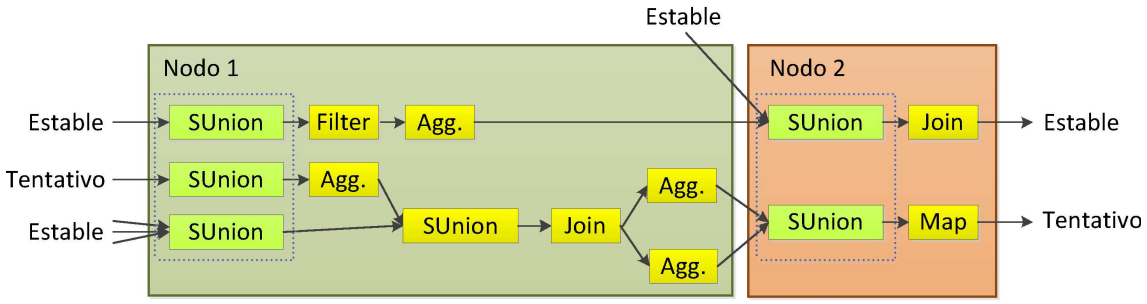


Figura 4.15: Ubicaciones donde las tuplas se bufferean con checkpoint/redo

los operadores que bufferean tuplas.

4.12.3 Algoritmo básico de gestión de buffer

Los buffers dentro de los SUnions crecen solamente en presencia de fallas. Sin embargo, los buffers de salida del DataPath deben truncarse explícitamente. Una técnica posible para truncar los buffers de salida consiste en que los nodos downstream envíen *acknowledgments* (reconocimientos) a todas las réplicas de sus vecinos upstream luego de recibir tuplas de entrada y que los nodos upstream borren tuplas de los buffers de salida una vez que fueron reconocidas por todas las réplicas de todos los vecinos downstream [HBR+05, BHS09].

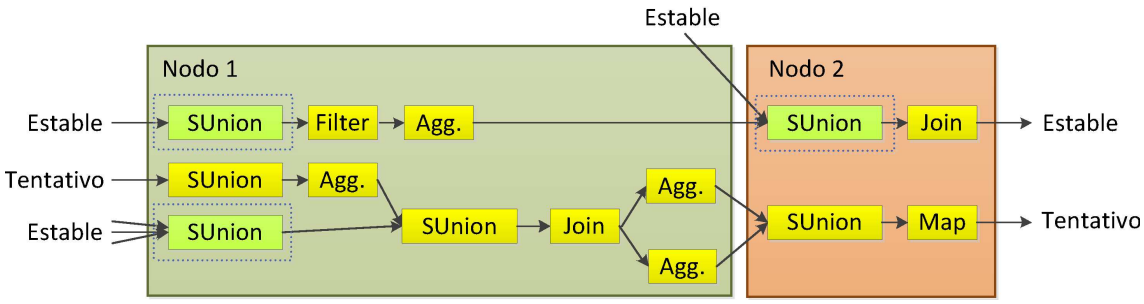


FIGURA 4.16: Ubicaciones donde las tuplas se bufferean con checkpoint/redo cuando todas las tuplas tentativas son corregidas

4.12.4 Gestión de las fallas prolongadas

El algoritmo de truncado de buffer anterior elimina tuplas de los buffers periódicamente mientras no se producen fallas. Si los nodos downstream fallan (*crash*) o se desconectan no son capaces de seguir enviando acknowledgments, forzando así el crecimiento de los buffers en función de la duración de la falla. Las tuplas de los buffers pueden almacenarse en disco y por lo tanto aún sin mecanismos adicionales, DCAA puede tolerar fallas relativamente prolongadas (dependiendo de las capacidades de almacenamiento de los nodos). Sin embargo, dada la razonabilidad de tratar de limitar el tamaño de todos los buffers, discutiremos a continuación un mecanismo capaz de conseguir este objetivo.

Cuando los nodos bufferean y reprocesan todas las tuplas a partir del principio de la falla, aseguran que los clientes reciban eventualmente las correcciones a todos los resultados tentativos anteriores, pero todavía más importante que esto, son capaces de reconciliar sus estados.

Limitar los tamaños de los buffers imposibilita la reconciliación de los estados de los nodos en el caso de producirse una falla suficientemente prolongada. La mejor forma de manejar las fallas de larga duración depende del tipo de operadores presentes en el diagrama de consulta.

Operadores determinísticos: en el peor caso el estado de un operador determinístico puede depender de todas las tuplas que el operador procesó. Con este tipo de operadores, cualquier tupla perdida durante una falla puede evitar que los nodos vuelvan a ser consistentes. Este tipo de situación se conoce con el nombre de *system delusion* [GHOS96]: las réplicas son inconsistentes y no existe una forma obvia de reparar el sistema. Para evitar este grave problema, cuando los operadores no son capaces de converger, proponemos mantener la disponibilidad solamente mientras exista espacio en los buffers. Una vez que se llenan los buffers de un nodo, éste se bloquea, creando presión hacia atrás hasta llegar a los orígenes de los datos, donde

se empiezan a descartar tuplas sin insertarlas en el sistema. Esta técnica mantiene la disponibilidad solamente mientras el tamaño del buffer lo permita pero asegura consistencia eventual y evita *system delusion*.

Operadores capaces de converger: tienen la ventajosa propiedad de que cualquier tupla de entrada afecta su estado solamente durante una cantidad finita de tiempo. Cuando un diagrama de consulta consta únicamente de este tipo de operadores, es posible calcular (para cualquier ubicación en el diagrama de consulta) un tamaño máximo de buffer, S, tal que garantice que pueda bufferearse la cantidad suficiente de tuplas capaz de reconstruir el último estado consistente y corregir las tuplas tentativas más recientes. Con este tratamiento, el sistema puede soportar fallas de duración arbitraria empleando buffers de tamaño finito.

Los operadores capaces de converger efectúan cómputos sobre ventanas de datos que se deslizan (*slide windows*) mientras nuevas tuplas arriban. Para obtener S en cualquier punto del diagrama de consulta, en el peor caso, será necesario sumar los tamaños de ventana de todos los operadores downstream. Para corregir una ventana W que contiene las últimas tuplas tentativas podemos adicionar este valor a la suma del tamaño de todas las ventanas. Cuando se usa el mismo atributo para las especificaciones de ventana y para valores *stime* (Sección 4.8), S se traduce en el máximo rango de valores *stime*. Dado S y una tupla límite con valor B, el DataPath y/o los SUnions sólo necesitan satisfacer:

$$stime > B - S$$

Dado que sólo las tuplas estables son buffereadas, es posible truncar buffers inclusive durante las fallas.

La Figura 4.17 muestra un ejemplo de asignaciones de tamaño de buffers. El tamaño de cada buffer se identifica con S, las etiquetas de los operadores fueron reemplazadas por ejemplos de tamaños de ventana. El usuario quiere recibir las versiones estables de las últimas $W = 10$ tuplas tentativas.

Por lo tanto, con buffers limitados y operadores capaces de converger, DCAA mantiene la disponibilidad en presencia de fallas de duración arbitrariamente larga. DCAA es incapaz de poder seguir asegurando que todas las tuplas tentativas serán corregidas luego de que una falla se solucione. Por el contrario, una aplicación especifica una ventana W y solamente las últimas W tuplas tentativas serán corregidas. Por ejemplo, una aplicación de monitoreo de red puede especificar que necesita ver todas las intrusiones y demás anomalías que ocurrieron durante la última hora. Los diagramas de consulta con operadores capaces de converger son entonces más efectivos para aplicaciones que necesitan mantener disponibilidad durante fallas de

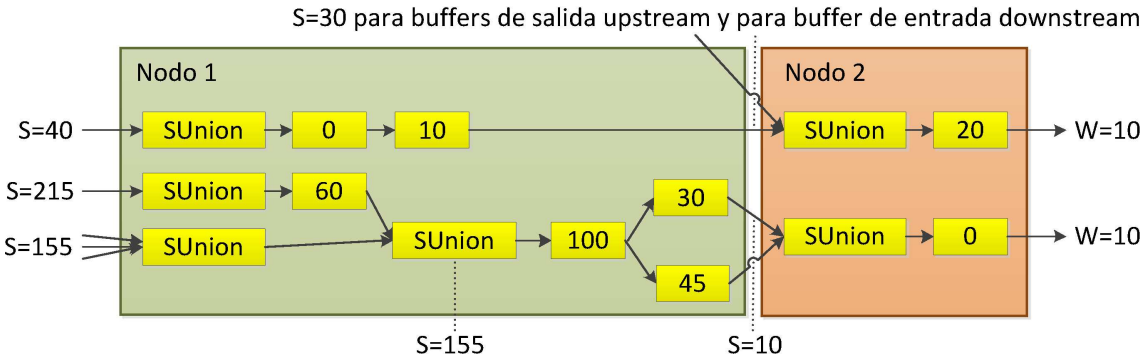


FIGURA 4.17: Ejemplo de asignación de tamaño de buffer con operadores capaces de converger

duración arbitrariamente largas, pero también necesitan arribar a un estado consistente una vez que la falla se soluciona. Para otros operadores determinísticos, los tamaños de los buffers determinan la duración de las fallas que el sistema puede soportar manteniendo la disponibilidad.

El último componente de DCAA es la gestión del buffer. En la próxima sección, discutimos brevemente el impacto de DCAA en las aplicaciones cliente y los orígenes de datos.

4.13 Aplicaciones cliente y los orígenes de datos

DCAA necesita que las aplicaciones cliente, y más importante todavía, que los orígenes de datos participen del protocolo de tolerancia a fallas. Esto se lleva a cabo haciendo que aplicaciones y orígenes de datos empleen una librería de tolerancia a fallas o haciendo que se comuniquen con el sistema mediante un *proxy* (o nodos de procesamiento cercanos) que implementen la funcionalidad requerida.

A continuación describiremos un proxy de este tipo. La Figura 4.18 ilustra el uso de proxies para los orígenes de datos y las aplicaciones cliente.

4.13.1 Proxies

Cuando un proxy recibe un pedido de comunicación proveniente de una aplicación cliente subscripta a un stream, se subscribe al stream en lugar del cliente. El proxy monitorea todas las réplicas de sus vecinos upstream y cambia (*switch*) entre las réplicas manteniendo la disponibilidad y asegurando a su vez consistencia eventual. El proxy corre un filtro *pass-through* (en lugar de un SUnion) y envía todas las tuplas de salida directamente a la aplicación cliente. La Figura 4.18 (a) muestra que el cliente recibe en forma mezclada tuplas tentativas y correcciones a tuplas

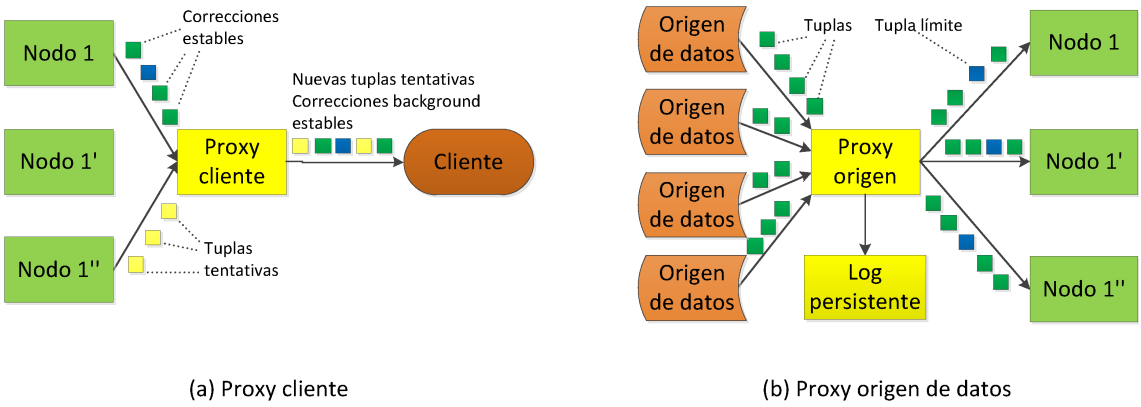


Figura 4.18: Proxies cliente y origen de datos

tentativas anteriores. El proxy asegura que no aparezcan tuplas tentativas luego de un `FIN_REC` a menos que indiquen el principio de una falla. Idealmente el proxy debería correr en la misma máquina que el cliente, de forma tal que si el proxy se desconecta del resto del sistema, el cliente también quede desconectado. Una falla en el proxy se considera equivalente a una falla en el cliente.

4.13.2 Proxies en el origen de los datos

DCAA necesita que los orígenes de los datos realicen las siguientes funciones:

1. Enviar sus streams a todas las réplicas de los nodos que procesan esos mismos streams.

2. Bufferear tuplas de salida hasta que todas las réplicas de todos los vecinos downstream confirmen (ACK) que las recibieron.

3. Establecer los valores de tupla *stime*.

4. Insertar periódicamente tuplas límite.

La Figura 4.18 (b) muestra como un proxy puede realizar las tareas anteriores en nombre de uno o más orígenes de datos. El proxy debe ubicarse cerca del origen de los datos, pues cualquier falla de red entre el origen de los datos y el proxy es considerada una falla en el origen de los datos. La falla del proxy también es considerada como una falla en el origen de los datos.

Como parte del protocolo de tolerancia a fallas, un proxy puede asignar valores de tupla *stime* y puede insertar periódicamente tuplas límite en nombre de sus orígenes de datos, si estos últimos no pueden hacerlo por ellos mismos. Dado que el proxy se encuentra ubicado junto al origen de los datos se asume que si no recibe tuplas

quiere decir que no hay datos que enviar, ya sea porque no hay datos disponibles o porque falló el origen de los datos). El proxy también puede fallar y por lo tanto debe registrar (*log*) tuplas persistentemente antes de transmitirlas al sistema para poder asegurar que todas las réplicas vean eventualmente las mismas tuplas de entrada. Una técnica alternativa es que el proxy tenga una réplica *hot-standby* que tome su lugar frente a una falla.

En la Sección 4.4.2, mencionamos que DCAA soporta únicamente fallas temporales en el origen de los datos o sus proxies. Las fallas permanentes son equivalentes a un cambio en el diagrama de consulta.

4.14 Propiedades de DCAA

En la Sección 4.4 delineamos cuatro propiedades que DCAA debe asegurar (Propiedades 4.1 á 4.4). A continuación revisaremos y refinaremos estas propiedades para luego mostrar como DCAA las cumple. La Tabla 4.4 resume las propiedades revisadas.

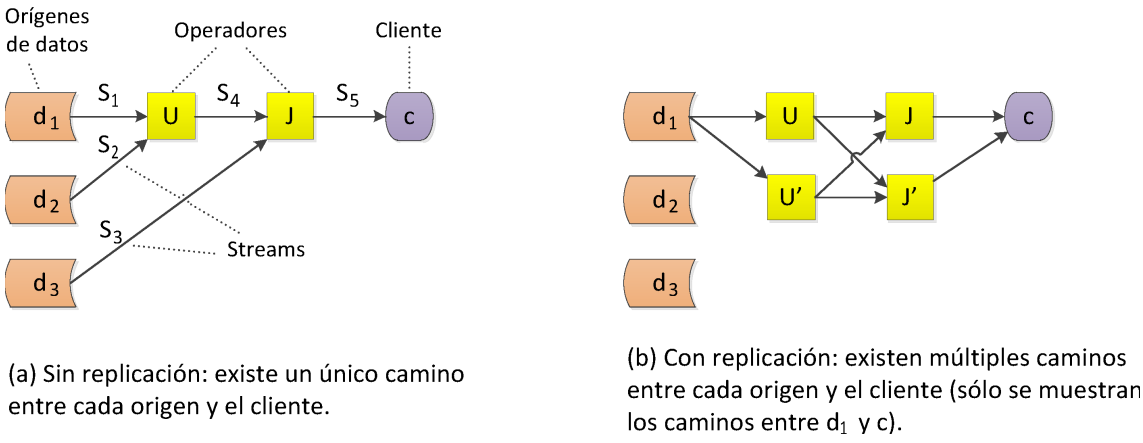


FIGURA 4.19: Paths en un diagrama de consulta

Decimos que un origen de datos contribuye con un stream s si produce un stream que se vuelve s luego de pasar por una secuencia de operadores denominada *path*. La Figura 4.19 muestra ejemplos de paths en un diagrama de consulta con y sin replicación. En el ejemplo, el origen d_1 contribuye con los streams s_1, s_4, s_5, mientras que d_2 contribuye con s_2, s_4, s_5 y d_3 con $s3, s5$. Sin replicación (Figura 4.19 (a)), existe un path entre cada origen y el cliente c. Mientras que si replicamos la `Union` y el `Join`, ahora existen cuatro paths entre d_1 y c. También encontramos cuatro paths desde d_2 a c y dos desde d_3 a c.

La unión de paths que conectan un conjunto de orígenes con un destino (un cliente o un operador) forma un *árbol*. Un árbol es válido si los paths que recorren el

mismo operador también recorren la misma réplica de ese operador, de otra forma el árbol es inválido. Un árbol válido es estable si el conjunto de orígenes de datos en el árbol incluye todos los orígenes que contribuyen con el stream recibido por el destino. Un árbol estable produce tuplas estables durante su ejecución. Si el conjunto de orígenes no incluye a todos aquellos que contribuyen con el stream y cualquiera de los orígenes faltantes se conecta con el árbol a través de operadores no bloqueantes entonces el árbol es *tentativo*, si no es éste el caso entonces se lo llama *bloqueante*. La Figura 4.20 muestra un ejemplo de cada tipo de árbol empleando un diagrama de consulta replicado de la Figura 4.19.

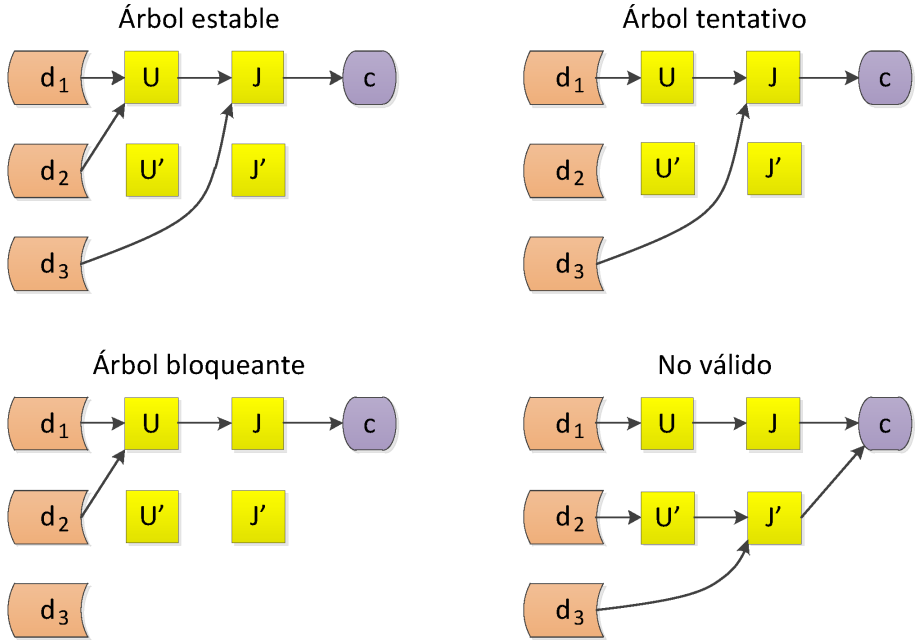


Figura 4.20: Tipos de árboles posibles en un diagrama de consulta distribuido y replicado

4.14.1 Propiedades de disponibilidad y consistencia

El objetivo principal de DCAA es mantener la disponibilidad mientras al mismo tiempo asegura consistencia eventual. En esta sección se muestra cómo se logran estos objetivos.

En primer lugar nos enfocaremos en la disponibilidad. En la Sección 4.4, la Propiedad 4.2 establece que si existe un camino de operadores no bloqueantes entre el origen de los datos y una aplicación cliente, el cliente recibe los resultados dentro del límite de tiempo especificado. La Propiedad 4.3 establece que DCAA favorece los resultados estables por sobre los tentativos. Contando con una noción de árbol más precisa reveremos las cuatro propiedades para poder probarlas o en su defecto presentar argumentos sólidos para cada una de ellas.

Propiedad 4.1 **Procesar las entradas disponibles a pesar de las fallas:** en un estado de falla estático, si existe un árbol estable, hay un destino que recibe tuplas estables. Si existen únicamente árboles tentativos entonces el destino recibirá tuplas tentativas provenientes de uno de los árboles tentativos. En otro caso, el destino puede bloquearse.

Precondición: asumimos que todos los nodos comienzan en el estado ESTABLE y son capaces de comunicarse entre sí cuando ocurren fallas. Luego, no ocurren otras fallas y ninguna de las fallas se soluciona.

Demostración: Esta propiedad comprende tres algoritmos: el algoritmo que determina el estado de los streams de salida a partir del estado de los streams de entrada (asumimos que los nodos utilizan el algoritmo detallado de la Figura A.1), el algoritmo de monitoreo de los streams de entrada (Algoritmo 4.1), y el algoritmo de cambio de vecino upstream de la Tabla 4.3. Probaremos esta propiedad mediante inducción en la profundidad del árbol.

1. **Caso base:** los nodos que se comunican directamente con una fuente de datos reciben tuplas estables si y solo si la fuente está disponible (y por lo tanto existe un árbol estable). Si la fuente de datos falla o se desconecta entonces no existe ni árbol estable ni tentativo, y por lo tanto los nodos no reciben ninguna tupla. La propiedad se cumple para el caso base.

2. **Hipótesis inductiva (parte 1):** si se trata de un nodo downstream perteneciente a un árbol estable, emplea un stream estable como entrada. En cambio si se trata de un nodo downstream perteneciente a un árbol tentativo y no existe ningún árbol estable, entonces el nodo utiliza entradas tentativas. De otra forma el nodo se bloquea. Asumimos por el momento (y lo discutiremos en la parte 2), que el nodo etiqueta sus salidas apropiadamente para reflejar el tipo de árbol al que pertenece cada stream:

   - la salida de un árbol estable es ESTABLE,
   - la salida de un árbol tentativo es etiquetada como FALLA_UPSTREAM,
   - y la salida de un árbol bloqueado es etiquetada como FALLA.

3. **Argumento para el paso inductivo (parte 1):** los nodos downstream periódicamente solicitan el estado de los vecinos réplica upstream empleando el algoritmo 4.1. Utilizando esta información, los nodos cambian entre réplicas de vecinos upstream siguiendo el algoritmo de la Tabla 4.3. Si existe una réplica estable de un vecino upstream, el nodo cambia a esta réplica (Tabla 4.3, estado

2). Si no existe una réplica estable pero si una en estado FALLA_UPSTREAM, el nodo cambia a esta otra réplica (Tabla 4.3, estados 7 y 9). Si no existe un árbol estable pero si uno tentativo, los nodos eligen como entrada la salida del árbol tentativo. En otro caso sólo existen árboles bloqueados y por lo tanto el nodo se bloquea.

4. **Hipótesis inductiva (parte 2):** los nodos etiquetan sus salidas correctamente según el tipo de árbol al que pertenecen.

5. **Argumento para el paso inductivo (parte 2):** los nodos utilizan el algoritmo A.1 para etiquetar sus streams de salida. Si todas las entradas son estables, entonces todas las salidas se etiquetan como estables (líneas 5 y 7). Si un nodo se bloquea a sí mismo porque está reconciliando su estado, entonces las salidas afectadas se bloquean (línea 3). Para el resto de los casos debemos determinar si un stream de salida afectado por una falla es la salida de un árbol bloqueante o tentativo. Para que un stream sea salida de un árbol bloqueante debe ser un downstream de un operador bloqueante con al menos una entrada bloqueada (línea 9) o un downstream de un operador con todas sus entradas bloqueadas (línea 11). En otro caso la salida es tentativa (línea 14).

Por lo tanto dado que la propiedad se mantiene para los nodos que reciben sus entradas directamente a partir de las fuentes de datos, los nodos etiquetan sus salidas según el tipo de árbol al que pertenecen. Asimismo dado que los nodos downstream siempre seleccionan sus entradas a partir del mejor árbol disponible, esta propiedad se mantiene para todos los nodos.

Mientras un nodo mantiene la propiedad anterior, puede a su vez corregir tuplas de entrada tentativas empleando un segundo stream de entrada al que llamamos Background(s). Es importante destacar que estas correcciones en background no afectan la disponibilidad.

Nuestro objetivo no se limita únicamente a asegurar que los clientes reciban los mejores resultados posibles sino que además los reciban con una latencia de procesamiento acotada.

Propiedad 4.2 **Mantener baja la latencia de procesamiento de las entradas disponibles:** si existe un árbol estable o tentativo, el nodo destino recibe los resultados con una demora que satisface $\texttt{Demora}_{\texttt{nueva}} < kD$, donde D es la demora asignada a cada operador `SUnion` y k es el número de `SUnion`s en el path más largo del árbol.

Demostración: la Propiedad 1 establece que si existe un árbol estable o tentativo entonces un nodo siempre elige la salida de ese árbol por sobre la de un árbol bloqueado. En DCAA, un operador `SUnion` nunca bufferea una tupla de entrada durante un lapso de tiempo mayor al de su asignada demora D. Dado que existen a lo sumo k `SUnion`s en un camino, entonces la demora de procesamiento de una tupla no supera kD. Analizaremos las propiedades de la demora con mayor profundidad en el Capítulo 5, donde además discutiremos como se asignan las demoras a los `SUnion`s.

Otro objetivo de DCAA es minimizar las inconsistencias. Más específicamente lo que se pretende es lograr la siguiente propiedad:

Propiedad 4.3 **Emplear técnicas que reduzcan la inconsistencia.** Para lograr las Propiedades 4.1 y 4.2 empleamos métodos capaces de producir la menor cantidad posible de tuplas tentativas: *i.e.,* $\mathtt{N}_{\mathtt{tentativa}}$ producida por el método elegido es menor que la $\mathtt{N}_{\mathtt{tentativa}}$ resultante de aplicar cualquier otro método.

Para lograr esta propiedad empleamos un método adaptativo capaz de manejar tanto fallas cortas como prolongadas. Diferiremos esta discusión para el Capítulo 5.

Las propiedades anteriores apuntan a lograr uno de los principales objetivos de DCAA, mantener la disponibilidad a pesar de las fallas y tratar de minimizar las inconsistencias. El otro objetivo principal de DCAA es asegurar la consistencia eventual. Primero presentaremos el caso de una única falla, luego discutiremos las fallas simultáneas en las Propiedades 4.5 á 4.7.

Propiedad 4.4 **Asegurar consistencia eventual:** un conjunto de fallas provoca dos tipos de problemas: caída de nodos y caída de enlaces de comunicación entre nodos. Asumimos que no existen otros tipos de fallas. Si al menos una réplica de cada nodo de procesamiento no falla, cuando todas las fallas se solucionan, el nodo destino recibe el stream estable de salida completo.

Precondición: asumimos que todos los nodos comienzan en el estado STABLE y son capaces de comunicarse con el resto de los nodos cuando ocurre un conjunto de fallas. Luego de este suceso no ocurren más fallas. Los nodos bufferean tuplas y las eliminan de los buffers según describimos en la Sección 4.12. Los buffers son suficientemente grandes (o conversamente las fallas son suficientemente cortas) para asegurar que no se descartan tuplas por falta de espacio.

Demostración: Probaremos esta propiedad por inducción en la profundidad del árbol y usaremos las propiedades del algoritmo de cambio de vecino upstream de la Tabla 4.3.

1. **Caso base:** dado que asumimos que todas las tuplas producidas durante la

falla son bufereadas, cuando una falla se soluciona, entonces todos los nodos que se comunican en forma directa con los orígenes de los datos reciben una repetición (*replay*) de todas las tuplas previamente faltantes. Por lo menos uno de estos nodos tiene que haber permanecido disponible durante la falla. Este nodo pasa a ESTABILIZACIÓN: reconcilia su estado y estabiliza sus salidas. Dado que el nodo reprocesa todas las tuplas desde el principio de la falla, puede entonces también corregir todas las tuplas de salida tentativas que se produjeron durante la falla. Por lo tanto, los vecinos downstream del nodo logran obtener las tuplas estables que corrigen todas las tuplas previamente marcadas tentativas.

2. **Hipótesis inductiva:** si al menos un nodo upstream corrige todos las tuplas tentativas que produjo previamente, entonces un nodo downstream que permaneció online durante la falla puede reconciliar su estado y estabilizar su salida, corrigiendo secuencialmente todas las tuplas tentativas que generó.

3. **Argumento para el paso inductivo:** si al menos una réplica de un nodo upstream corrige todas las tuplas tentativas previamente generadas, entonces la réplica vuelve al estado ESTABLE. En este punto todas las fallas se solucionaron, los nodos downstream detectan la transición y cambian hacia la réplica estable como nodo upstream (algoritmo de la Tabla 4.3, estado 2). Cuando un nodo cambia a una réplica estable como nodo upstream, envía el identificador de la última tupla estable que recibió. Dado que asumimos que los buffers pueden almacenar todas las tuplas producidas durante la falla, el nuevo vecino upstream puede corregir todas las tuplas tentativas que siguen a la tupla identificada (como la última estable) por el nodo downstream. En este momento el nodo recibió los streams de entrada en forma completa y correcta. Los nodos pueden corregir sus entradas en background mientras el nodo upstream se encuentra en ESTABILIZACIÓN, como puede verse en la Tabla 4.3, estados 3 y 6.

   Una vez que el nodo recibe la versión estable de todas las tuplas previamente marcadas tentativas (o entradas faltantes), puede ir a ESTABILIZACIÓN. El nodo puede reprocesar todas las tuplas desde el principio de la falla, reconciliando su estado y corrigiendo todas las tuplas tentativas que produjo.

Por ello, luego de solucionada una falla, los nodos que se comunican con los orígenes de los datos reciben una repetición de todas las tuplas previamente faltantes. Cada nodo reconcilia su estado y estabiliza sus salidas, corrigiendo todas las tuplas

tentativas producidas durante la falla. Cada vez que un nodo estabiliza sus salidas, sus vecinos downstream pueden corregir sus entradas, reconciliar sus estados y estabilizar sus salidas secuencialmente. Debido a que asumimos que existe al menos una réplica de cada nodo de procesamiento que permanece online durante la falla, este proceso se propaga a las aplicaciones cliente.

En la Sección 4.12 discutimos el caso donde las fallas exceden la capacidad de buffer con diagramas de consulta capaces de converger. Los nodos descartan viejas tuplas de los buffers y no pueden corregir todas las tuplas tentativas, sólo las más recientes. Para otros diagramas de consulta, una vez que los buffers se llenan DCAA se bloquea evitando que nuevas entradas ingresen al sistema. En este momento el sistema no puede seguir manteniendo disponibilidad, pues debe asegurar consistencia eventual y evitar *system delusion*[9].

4.14.2 Propiedades de las fallas múltiples

Además de las propiedades básicas arriba mencionadas mostraremos que DCAA es capaz de manejar tanto fallas múltiples como fallas durante la recuperación. En la Propiedad 4.5 mostraremos que mientras ocurren las fallas y un nodo cambia múltiples veces entre réplicas de un vecino upstream, las tuplas estables en sus streams de entrada nunca deben ser descartadas, ni son duplicadas. En la propiedad 4.6, mostraremos que cuando un nodo reconcilia su estado, nunca descarta, duplica, o deshace tuplas estables de salida a pesar de fallas simultáneas e incluso fallas durante la recuperación. Finalmente, la Propiedad 4.7 establece que un nodo periódicamente produce tuplas estables de salida, aún cuando sus streams de entrada fallen y se recuperen frecuentemente.

Propiedad 4.5 **Manejar fallas simultáneas múltiples:** cambiar entre árboles nunca causa que las tuplas de entrada estables deban ser descartadas, duplicadas, o deshechas.

Precondición: los nodos bufferean y remueven de los buffers tuplas según describimos en la Sección 4.12. Todos los buffers son lo suficientemente grandes, o análogamente las fallas son lo suficientemente cortas, como para asegurar que ninguna tupla se descarte por falta de espacio.

Para argumentar esta propiedad estudiaremos cada posible escenario de cambio de vecino:

1. Cambiar de un vecino upstream cuyo estado era ESTABLE antes de que la falla ocurriese a un vecino upstream que todavía está en estado ESTABLE:

[9] La definición de *system delusion* se encuentra en la página 102.

el nodo downstream indica que se recibió el identificador de la última tupla estable, por lo tanto una nueva réplica ESTABLE puede continuar a partir del punto correcto en el stream, ya sea esperando producir la tupla identificada o repitiendo su buffer de salida.

2. Cambiar de un vecino en el estado FALLA_UPSTREAM a uno en el estado ESTABLE: en este caso el nodo downstream indica el identificador de la última tupla estable que recibió y un *tag* que indica que procesó tuplas tentativas a partir de la última tupla estable. Por lo tanto, el nuevo vecino upstream "sabe" que debe estabilizar su salida y además el punto exacto en el stream desde dónde comenzar las correcciones.

3. Cambiar a un vecino upstream que se encuentra en el estado FALLA_UPSTREAM: en este caso el nodo downstream y su nuevo vecino upstream probablemente continuarán en sus estados mutuamente inconsistentes. De hecho, la última tupla estable producida por el nuevo vecino (antes de que pasase al estado FALLA_UPSTREAM) ocurrió en el stream antes o después de la última tupla estable recibida por parte del nodo downstream, tal como lo ilustra la Figura 4.21.

   Si la última tupla estable producida por el nuevo vecino aparece antes de tiempo en el stream, Figura 4.21 Nodo 2 y nodo 2', significa que el nuevo vecino upstream ha permanecido en el estado FALLA_UPSTREAM por un lapso de tiempo mayor que el del anterior vecino upstream. Debido a que los nodos no pueden deshacer tuplas estables, el nuevo par upstream y downstream debe seguir procesando tuplas estando en estados mutuamente inconsistentes.

   Si la última tupla estable producida por el nuevo vecino aparece tardíamente en el stream, como por ejemplo en el caso del Nodo 2' de la Figura 4.21, entonces dicho nuevo vecino puede estabilizar las tuplas tentativas más antiguas antes de continuar con las tentativas más recientes. Sin embargo, el nodo downstream, está conectándose a una réplica en el estado FALLA_UPSTREAM para obtener los datos de entrada más recientes en lugar de las correcciones, por lo tanto la nueva réplica directamente continua con las tuplas tentativas más recientes.

   En ambos casos, el nodo upstream "recuerda" las últimas tuplas estables recibidas por cada uno de los vecinos downstream pero les envía los datos tentativos más recientes. Luego de que la falla se soluciona y el nodo upstream estabiliza su salida, busca la última tupla estable recibida por cada cliente downstream y produce *undo*s y correcciones necesarias para cada uno de ellos.

4. Finalmente, un nodo puede conectarse a una réplica que se encuentra en el estado ESTABILIZACIÓN para corregir los streams de entrada en background. Dado que el nodo indica cuál fue la última tupla estable que recibió, las correcciones en background siempre comienzan en el punto apropiado del stream.

   Adicionalmente, cuando un nodo recibe tuplas de entrada tentativas y correcciones, DCAA necesita que el DataPath monitoree estos streams entrantes, asegurando que solamente el stream de background contenga datos estables.

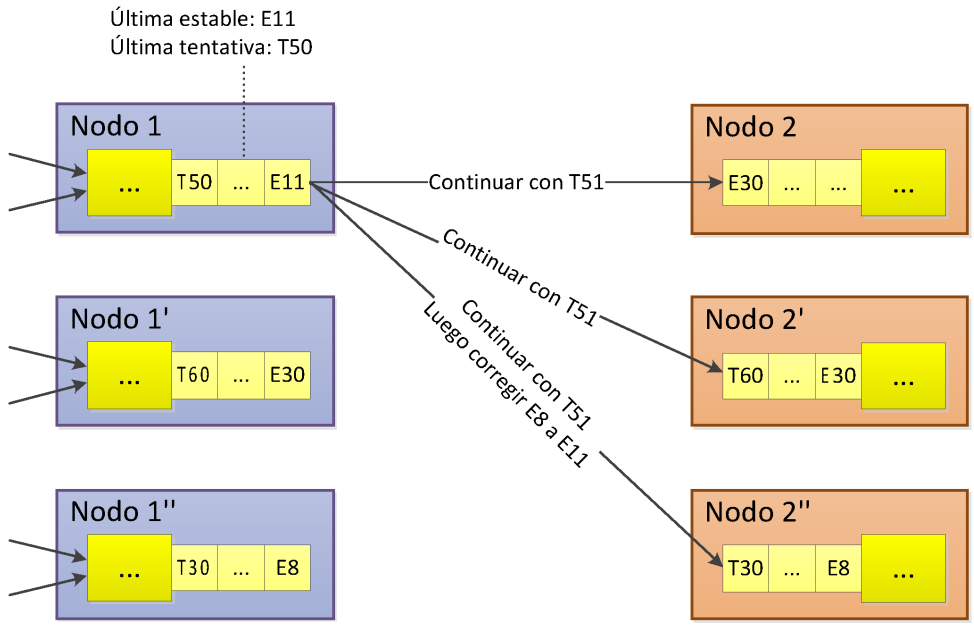


FIGURA 4.21: Cambio entre réplicas de vecinos upstream durante diferentes estados de consistencia

Mostramos que cuando un nodo cambia entre réplicas de vecinos upstream, para todos las combinaciones de consistencia de estos vecinos, el nodo downstream recibe las tuplas de entrada correctas; las tuplas estables nunca se deshacen *undo*, descartan, o duplican. Sin embargo, las tuplas tentativas si pueden duplicarse, incluso es posible que sean descartadas mientras un nodo se encuentra cambiando de vecino upstream, pero ni estas duplicaciones ni estas pérdidas afectan las propiedades esenciales de nuestro protocolo de tolerancia a fallas: disponibilidad y consistencia eventual.

El segundo problema con las fallas múltiples es que los streams de entrada se vuelven tentativos en diferentes momentos y puede suceder entonces que no todos sean corregidos al mismo tiempo. En el caso extremo habrá al menos una entrada fallada. Adicionalmente pueden ocurrir nuevas fallas en cualquier momento, inclusive cuando el nodo está en el proceso de recuperación de una falla previa. Para mantener

consistencia se debe garantizar que la estabilización nunca cause que las tuplas sean duplicadas, descartadas, o deshechas.

Propiedad 4.6 **Manejar fallas durante las fallas y la recuperación:** la estabilización nunca provoca que las tuplas sean duplicadas, descartadas, o deshechas (*undo*).

Precondición: Los nodos bufferean y eliminan tuplas de los buffers según se describió en la Sección 4.12. Todos los buffers son lo suficientemente grandes (o las fallas son lo suficientemente cortas) para asegurar que no se descartan tuplas por falta de espacio.

Argumentaremos sobre esta propiedad examinando nuevamente cada posible escenario de falla para cada metodología de reconciliación de estado. A continuación presentaremos *checkpoint/redo*, dejando para el Apéndice B la técnica *undo/redo* y mostraremos que DCAA es capaz de manejar fallas durante las fallas y el proceso de recuperación sin riesgos de duplicar, descartar, o deshacer tuplas.

Dado que nuestro objetivo es producir pocas tuplas tentativas, para que un nodo entre en el estado de ESTABILIZACIÓN debe haber recibido correcciones a todas las tuplas previamente marcadas tentativas en todos sus streams de entrada. De hecho, si un nodo comienza la reconciliación en el momento en que sólo un stream fue corregido, si el stream se une (join) con otro stream todavía tentativo, es probable que las nuevas tuplas que resulten de la reconciliación sean tentativas. Por lo tanto, cuando un nodo comienza a reconciliar su estado tiene viejas tuplas estables en todos sus streams de entrada. Si ocurre una falla durante la estabilización, las nuevas tuplas tentativas aparecen luego de estas tuplas estables en los streams de entrada.

Debido a que los `SUnion`s se sitúan en todos los streams de entrada, se bufferean todas las tuplas que arriban entre checkpoints y por lo tanto recomenzar desde un checkpoint no causa pérdida de tuplas.

Los operadores `SOutput` garantizan que las tuplas estables no se dupliquen ni necesiten deshacerse (*undo*). Al recomenzar a partir de un checkpoint, `SOutput` entra en el modo *eliminación de duplicados*, esperando la última tupla duplicada hasta que produzca la tupla `DESHACER`, aún si ocurriese otro checkpoint o proceso de recuperación. Luego de producir el `DESHACER`, `SOutput` vuelve a su estado normal, recordando la última tupla estable que ve (*i.e.,* guardando su identificador) durante los checkpoints. Si ocurre una nueva falla antes de que el nodo tenga tiempo de actualizarse (ponerse al día) y producir una tupla `FIN_REC`, `SOutput` fuerza una tupla `FIN_REC` entre la última tupla estable y la primer tupla tentativa que ve.

Por lo arriba descripto, con la recuperación *checkpoint/redo*, las fallas pueden suceder durante el transcurso de otras fallas o durante la recuperación y el sistema

puede seguir garantizando que las tuplas no se dupliquen, descarten, o necesiten deshacerse.

Finalmente abordaremos el problema de las fallas frecuentes y mostraremos que aún si no hay suficiente tiempo para restablecer todos los componentes y las comunicaciones, el sistema puede progresar. Definimos *progreso* como la producción de nuevas tuplas estables. Para lograr este progreso en los streams de salida cada stream del diagrama de consulta debe periódicamente progresar.

Propiedad 4.7 **Gestionar las fallas frecuentes:** aún cuando las entradas de un nodo de procesamiento fallen tan frecuentemente que al menos una esté caída en todo momento, si cada stream de entrada se recupera entre fallas y progresa, entonces el nodo mismo progresa.

Precondición: Los nodos bufferean tuplas y las eliminan de los buffers según describimos en la Sección 4.12. Los buffers son suficientemente grandes (o conversamente las fallas son suficientemente cortas) para asegurar que no se descartan tuplas por falta de espacio.

Un stream de entrada progresa cuando un nodo es capaz de recibir una repetición de tuplas faltantes o correcciones de tuplas de entrada tentativas. Un nodo progresa cuando reconcilia su estado y estabiliza su salida.

Discutiremos esta propiedad mediante un conjunto de aserciones acerca del nodo de procesamiento que presenta fallas en sus entradas. Por claridad asumimos que los operadores `SUnion` ubicados en los streams de entrada bufferean tuplas en buckets identificados con números de secuencia que se incrementan con el tiempo. Asimismo asumimos que para todos los `SUnion`s los buckets con el mismo número de secuencia corresponden aproximadamente al mismo punto en el tiempo.

Aserción 1: para cualquier stream s y bucket i, el nodo eventualmente recibe tuplas estables para ese bucket de ese stream. Argumento: un stream que falla siempre se recupera eventualmente y logra progresar antes de volver a fallar. Cada recuperación posibilita que el nodo downstream corrija (posiblemente en background) al menos un subconjunto de tuplas previamente faltantes o tentativas en ese stream. Por lo tanto, el nodo eventualmente recibe tuplas estables de entrada para todos los buckets en todos los streams de entrada.

Aserción 2: para todo i, todos los operadores `SUnion` tienen eventualmente solamente tuplas estables en su bucket i. Argumento: se desprende directamente de la Aserción 1.

Aserción 3: el nodo eventualmente entra en el estado de estabilización. Argumento: sea j el primer bucket procesado como tentativo por al menos un `SUnion`. Por

la Aserción 2, eventualmente todos los SUnions tendrán tuplas estables en su bucket j, luego el nodo podrá entrar en el estado ESTABILIZACIÓN.

Aserción 4: un nodo progresa mediante el procesamiento de tuplas estables a pesar de las fallas. Argumento: por la Aserción 3, un nodo entra eventualmente en ESTABILIZACIÓN. Asumiendo que el nodo utiliza la técnica de checkpoint/redo[10] en la primer reconciliación de estado, el nodo recomienza desde un checkpoint tomado antes de procesar el bucket j. Luego procesa el bucket j y desde allí continúa, eventualmente aparecen tuplas tentativas en el bucket $k > j$. En la siguiente estabilización, aún si el nodo recomienza desde el mismo checkpoint, procesa desde el bucket j hasta el k antes de comenzar a procesar nuevas tuplas tentativas. Podemos entonces concluir, una vez más, que el nodo progresa. Es claro que para evitar reprocesar las mismas tuplas múltiples veces y tener a SOutput filtrando duplicados, uno nodo debe realizar el checkpoint de su estado antes de empezar a procesar tuplas pertenecientes al bucket k.

Conclusión: dado que un nodo puede entrar periódicamente en el estado de ESTABILIZACIÓN y progresar a partir del procesamiento de tuplas estables, periódicamente produce nuevas tuplas estables en sus streams de salida.

4.15 Resumen

En este capítulo presentamos DCAA, un protocolo de tolerancia a fallas para un sistema distribuido de procesamiento de streams. DCAA introduce un modelo de datos mejorado, donde las tuplas son etiquetadas como tentativas o estables. DCAA se basa en la replicación: cada fragmento del diagrama de consultas se replica en múltiples nodos de procesamiento. Dicha replicación posibilita la tolerancia a fallas de múltiples nodos y enlaces de red, y por ende reduce las probabilidades de una falla total del sistema.

Una de las características más importantes de DCAA es que cada réplica debe manejar su propia disponibilidad y consistencia. Cada nodo implementa el protocolo de tolerancia a fallas DCAA, regido por un autómata finito de tres estados. En el estado ESTABLE las réplicas mantienen consistencia mutua empleando SUnion, un operador simple de serialización de datos, junto con tuplas límite periódicas. Gracias al mecanismo de monitoreo explícito del vecino upstream, los nodos utilizan estas tuplas límites para detectar fallas en sus streams de entrada. Cuando ocurren fallas, los nodos pasan al estado FALLA_UPSTREAM, donde mantienen el

[10] Puede emplearse un argumento similar para la reconciliación undo/redo.

nivel de disponibilidad requerido, mientras al mismo tiempo tratan de minimizar las inconsistencias cambiando entre réplicas de vecinos upstreams, y posiblemente demorando el procesamiento de nuevas tuplas de entrada. Una vez que la falla es solucionada los nodos pasan al estado ESTABILIZACIÓN, donde reconcilian sus estados empleando el mecanismo de *undo/redo* o el de *checkpoint/redo*. Para lograr mantener la disponibilidad durante la ESTABILIZACIÓN, los nodos ejecutan un protocolo de comunicación inter-réplica para lograr la reconciliación.

DCAA requiere que los nodos buffereen tanto sus tuplas de entrada como de salida (gestión de buffers). Para que las aplicaciones cliente logren beneficiarse de la tolerancia a fallas sin implementar protocolos adicionales, las aplicaciones cliente y los orígenes de los datos pueden comunicarse con el sistema a través de un nodo de procesamiento cercano que actúa como *proxy*.

Mostramos que DCAA provee la disponibilidad y consistencia requerida, soportando tanto una única falla como múltiples fallas simultáneas. Se diseñó DCAA a partir de una asunción de una frecuencia de fallas baja, pero vimos que también funciona cuando las fallas ocurren frecuentemente. Por último cabe destacar que DCAA funciona particularmente bien cuando los diagramas de consulta son capaces de converger.

Estado	Condición			Acción
	EnEstado(Actual(s), s)	EnEstado(Bg(s), s)	R = Réplicas(s) − Actual(s) − Bg(s)	
1	ESTABLE	−	−	Dessubscribirse de Bg(s) Bg(s) ← NIL Permanecer en Estado 1
2	! ESTABLE	NIL o ! ESTABLE	$\exists r \in R$, EnEstado(Actual(r),s) = ESTABLE	Actual(s) ← r Ir al Estado 1
3	FALLA_UPSTREAM	ESTABILIZACIÓN	$\forall r \in R$, EnEstado(Actual(r),s) ≠ ESTABLE	Permanecer en Estado 3
4	FALLA_UPSTREAM	NIL o FALLA_UPSTREAM o FALLA	$\forall r \in R$, EnEstado(Actual(r),s) ≠ ESTABLE y $\forall r\prime \in R$, EnEstado(Actual(r′),s) ≠ ESTABILIZACIÓN	Dessubscribirse de Bg(s) Bg(s) ← NIL Permanecer en Estado 4
5	FALLA_UPSTREAM	NIL o FALLA_UPSTREAM o FALLA	$\forall r \in R$, EnEstado(Actual(r),s) ≠ ESTABLE y $\exists r\prime \in R$, EnEstado(Actual(r′),s) = ESTABILIZACIÓN	Bg(s) ← $r\prime$ Subscribirse a Bg(s) Ir al Estado 3
6	ESTABILIZACIÓN	NIL o FALLA	$\forall r \in R$, EnEstado(Actual(r),s) ≠ ESTABLE y $\forall r \in R$, EnEstado(Actual(r),s) ≠ FALLA_UPSTREAM	Dessubscribirse de Bg(s) Bg(s) ← NIL Permanecer en Estado 6
7	ESTABILIZACIÓN	NIL o FALLA	$\forall r \in R$, EnEstado(Actual(r),s) ≠ ESTABLE y $\exists r\prime \in R$, EnEstado(Actual(r′),s) = FALLA_UPSTREAM	Bg(s) ← $r\prime$ Subscribirse a Bg(s) Bg(s) ↔ Actual(s) Ir al Estado 3
8	ESTABILIZACIÓN	ESTABILIZACIÓN	−	El más avanzado de los dos será el nuevo Actual(s) Dessubscribirse de Bg(s) Bg(s) ← NIL Ir al Estado 6 ó 7
9	FALLA	NIL o FALLA	$\forall r \in R$, EnEstado(Actual(r),s) ≠ ESTABLE y $\exists r\prime \in R$, EnEstado(Actual(r′),s) = FALLA_UPSTREAM	Actual(s) ← $r\prime$ Subscribirse a Actual(s) Ir al Estado 4 o 5
10	FALLA	NIL o FALLA	$\forall r \in R$, EnEstado(Actual(r),s) ≠ ESTABLE y $\exists r\prime \in R$, EnEstado(Actual(r′),s) = ESTABILIZACIÓN	Actual(s) ← $r\prime$ Subscribirse a Actual(s) Ir al Estado 6 o 7
11	FALLA	NIL o FALLA	$\forall r \in R$, EnEstado(Actual(r),s) = FALLA	Dessubscribirse de Bg(s) Bg(s) ← NIL Permanecer en Estado 11

Tabla 4.3: Algoritmo para cambio de réplica de un vecino upstream para mantener disponibilidad mientras se corrige entrada en bg

Propiedad	Descripción (resumida)
1	Procesar las entradas disponibles a pesar de las fallas.
2	Mantener baja la latencia de procesamiento de las entradas disponibles.
3	Emplear técnicas que reduzcan la inconsistencia.
4	Asegurar consistencia eventual.
5	Manejar fallas simultáneas múltiples.
6	Manejar fallas durante las fallas y la recuperación.
7	Gestionar las fallas frecuentes.

Tabla 4.4: Resumen de las propiedades del protocolo de disponibilidad y consistencia

Capítulo 5

Gestión de Carga

Índice

5.1. Definición del problema 125
5.1.1. Tareas y carga 125
5.1.2. Utilidad 126
5.1.3. Correspondencia de elección social 129
5.2. Descripción de MPA 130
5.3. Migraciones en tiempo de ejecución 133
5.4. Condiciones de carga dinámica 137
5.5. Contratos offline de precio fijo 140
5.6. Contratos de precio acotado 147
5.6.1. Rango de precio mínimo 148
5.6.2. Negociación del precio final 152
5.7. Aplicación de MPA a sistemas DSMS federados . 158
5.8. Propiedades de MPA 158
5.8.1. Asunciones adicionales 159
5.8.2. Propiedades 160
5.9. Resumen 172

The greatest difficulty with the world is not its ability to produce, but the unwillingness to share.

— *Roy L. Smith*

En este capítulo estudiaremos la gestión de carga, posiblemente el tema de mayor relevancia asociado a los DSMS distribuidos. Describiremos el *Mecanismo de Precio Acotado* (MPA), una técnica que permite que participantes autónomos gestionen las variaciones de sus cargas mediante contratos entre pares negociados offline. Si bien la motivación de este mecanismo está dada por el procesamiento de streams, MPA puede aplicarse a una gran variedad de sistemas distribuidos.

En un DSMS la carga del sistema varía según diversos factores, como por ejemplo los operadores agregados y eliminados por parte de los usuarios en un diagrama de consulta, variaciones en las tasas de entrada y distribuciones de los datos de entrada. Periódicamente el sistema debe cambiar las asignaciones de los operadores (tareas) a los nodos de procesamiento para lograr una buena performance, o por lo menos para evitar una degradación significativa de la misma.

En el Capítulo 1 mencionamos que la gestión de carga dinámica constituye un área ampliamente estudiada (e.g., [ELZ86, Gal77, KS89]), y por lo tanto fueron propuestas numerosas técnicas que posibilitan que un sistema alcance distribuciones de carga que optimizan alguna utilidad global, como por ejemplo el throughput, la latencia de procesamiento, o el tamaño de las colas. Dichas propuestas típicamente asumen un entorno colaborativo, en el cual todos los nodos trabajan cooperativamente para maximizar la utilidad global del sistema. Sin embargo, muchos sistemas distribuidos actuales suelen desplegarse en entornos federados, donde las diferentes organizaciones autónomas (por ejemplo diferentes universidades distribuidas geográficamente) poseen y administran un conjunto de nodos de procesamiento y de recursos.

Los sistemas federados surgen cuando los participantes individuales se benefician a partir de colaborar con otros. Por ejemplo, los participantes podrían colaborar para componer servicios *end-to-end* más completos. Ejemplos de sistemas federados son los *web services* [W3C02, KL03], sistemas *grid de cómputo* [BAG00, BSG+01, FK98, TTL05, **?**] y sistemas *peer-to-peer* [CFV03, DKK+01, LFSC03, NWD03, RD01, SMK+01, VCCS03]. Otro beneficio de los sistemas federados es que las organizaciones pueden crear un *pool* de recursos para poder hacer frente a picos de carga (*load spikes*) sin necesidad de contar, mantener o administrar individualmente los recursos (cómputo, red y almacenamiento) requeridos. En particular, ejemplos federados de este tipo son los grids de cómputo y las plataformas de cómputo basadas en *overlays* como Planetlab [PACR03]. Las aplicaciones de procesamiento de streams son inherentemente distribuidas y federadas, pues los streams de datos suelen provenir de ubicaciones geográficas remotas, por ejemplo *sensor networks* (redes

de sensores) [ASSC02b] instaladas en áreas lejanas, o incluso pertenecientes a diferentes organizaciones, por ejemplo acciones de distintas compañías. Finalmente, los streams de datos pueden estar compuestos de diferentes formas a partir de los servicios generados en distintas organizaciones.

La gestión de carga en un entorno federado constituye un desafío actual, pues los participantes trabajan motivados por un interés propio y no global. Las primeras investigaciones relacionadas con la gestión de carga entre participantes autónomos proponían el uso de economías de cómputo, por ejemplo [BAG00, BSG+01, SAL+96, WHH+92, EAS07]; sin embargo ninguno de estos esquemas alcanzó popularidad masiva. A nuestro entender esto se debe a que estas metodologías previas no solucionan el problema en forma completa, en la práctica los participantes autónomos tienden a colaborar estableciendo acuerdos entre pares [Cor11, FSP00, KL03, Rac11, GS05, Ver99, WS04, Wus02].

Basándonos en los resultados positivos expuestos por los acuerdos entre pares en la práctica proponemos MPA, un mecanismo distribuido para la gestión de carga en un sistema federado basado en contratos privados de a pares. A diferencia de las economías de cómputo, las cuales emplean subastas o implementan mercados globales para establecer en tiempo de ejecución el precio de un recurso [EA09], MPA se basa en la negociación *offline de contratos*. Los contratos asignan límites en los precios para migrar cada unidad de carga entre dos participantes y pueden especificar el conjunto de tareas que cada parte está dispuesta a ejecutar en lugar del otro.

Mediante MPA las transferencias de carga en tiempo de ejecución ocurren solamente entre participantes que negociaron contratos previamente y a un precio unitario que se encuentra dentro del rango contratado. El mecanismo de transferencia de carga es simple: *un participante mueve carga hacia otro sólo si el costo de procesamiento local esperado para el siguiente período de tiempo es mayor que el pago esperado que hubiese tenido que hacer a otro participante para procesar la misma carga (más el costo de migración)* [EA06].

Por lo tanto, en contraste con las propuestas anteriores, MPA provee (1) privacidad para todos los participantes con respecto a los detalles de sus interacciones con otros, (2) facilita la parametrización de servicios y precios, (3) proporciona un mecanismo de gestión de carga liviano basado en precios prenegociados, y (4) presenta buenas propiedades de balance de carga a nivel del sistema. Es posible y potencialmente útil pensar en extender estos contratos con cláusulas adicionales para personalizar los servicios ofrecidos, por ejemplo garantías de performance, seguridad, disponibilidad, etc.

En este capítulo se presenta MPA y sus propiedades. Comenzaremos formalizando el problema en la Sección 5.1 y haremos un resumen de nuestro protocolo en la Sección 5.2. Cada una de las secciones siguientes desarrollarán los componentes de MPA. En la Sección 5.3 primero se presentarán las estrategias y algoritmos responsables de las transferencias de carga en tiempo de ejecución asumiendo que cada participante tiene un conjunto de contratos de precios fijos y que la carga es fija. En la Sección 5.4 veremos cómo extender MPA para manejar un entorno que se ajuste mejor a la realidad, *i.e.,* con variaciones de carga en forma dinámica. Luego, en la Sección 5.5 discutiremos distintas estrategias para establecer contratos offline de precio fijo. Dado que los contratos de precio fijo no siempre llevan a asignaciones aceptables, en la Sección 5.6 propondremos relajar la restricción de precio fijo permitiendo contratos que especifiquen pequeños rangos de precios preacordados. Seguidamente, en la Sección 5.7, veremos de qué forma MPA puede aplicarse a DSMS federados. Finalmente presentaremos las propiedades de MPA en la Sección 5.8.

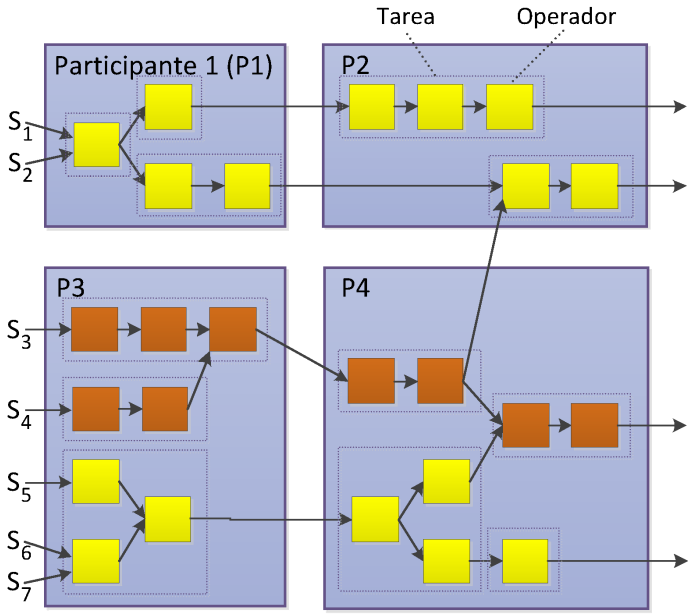
5.1 Definición del problema

En esta sección presentaremos el problema de la gestión de carga, introduciendo el modelo del sistema y definiendo el tipo de asignación de carga que trataremos de manejar. Emplearemos un sistema federado de procesamiento de streams como ejemplo ilustrativo para luego evaluar nuestra propuesta como soporte de cualquier sistema federado.

5.1.1 Tareas y carga

Asumimos que el sistema está compuesto por un conjunto de N *participantes* autónomos. Cada participante posee y administra un conjunto de recursos que utiliza para ejecutar *tareas* enviadas por sus propios clientes. Se asume también un conjunto dinámico K de tareas (el cual varía en el tiempo). Cada tarea en K se origina en algún cliente, quien a su vez la envía a un participante de N. Dado que solamente examinamos interacciones entre participantes utilizaremos los términos *participante* y *nodo* en forma intercambiable.

Una *tarea* es un cómputo de larga duración que requiere uno o más recursos (*e.g.,* memoria, CPU, almacenamiento y ancho de banda de red) y conforma la unidad de movimiento de carga. En un DSMS, una tarea comprende uno o más operadores interconectados y un diagrama de consulta. Éste se construye a partir de una o más tareas, como lo ilustra las Figuras 5.1 y 5.2.



(a) Los participantes P2 y P3 están sobrecargados.

FIGURA 5.1: Necesidad de reasignación de carga en un DSMS de cuatro participantes

El problema del particionamiento óptimo de un diagrama de consulta de tareas se deja como trabajo futuro.

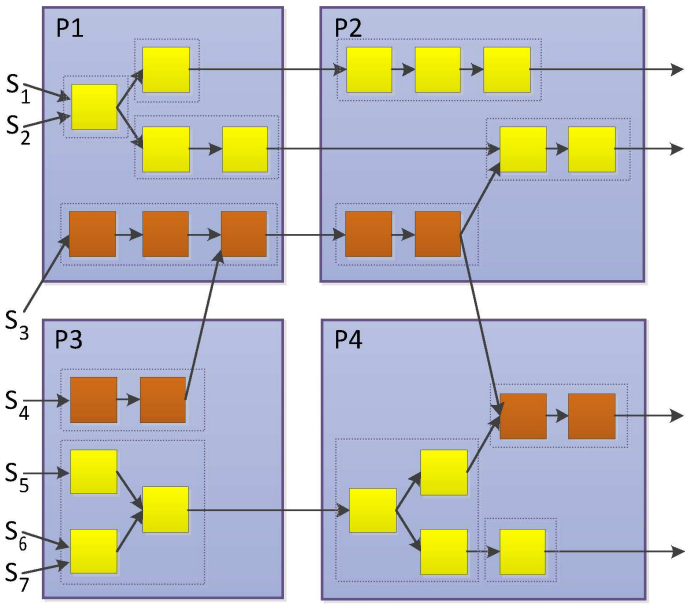
Asumimos también que el conjunto de tareas nos es dado y que cada tarea emplea una fracción relativamente pequeña de los recursos de un nodo. Si un operador en un DSMS utiliza una gran cantidad de recursos, por ejemplo un gran `Aggregate`, entonces el operador tendrá la posibilidad de particionarse en múltiples tareas más pequeñas [SHCF03].

5.1.2 Utilidad

La cantidad total de recursos empleados por las tareas constituyen la *carga* que experimentan los participantes. Para cada participante, la carga impuesta sobre sus recursos representa el *costo*. De hecho cuando la carga es baja, el participante puede manejar fácilmente todas sus tareas.

A medida que la carga se incrementa se vuelve más difícil para el participante proveer un buen servicio a sus clientes. Este resultado puede lograr que los clientes se sientan insatisfechos: los clientes pueden buscar otro proveedor o incluso demandar una compensación monetaria. Asumimos entonces que un participante puede cuantificar el costo de procesamiento para una carga determinada.

Más específicamente definiremos una *función de costo* para un participante i,



(b) Algunas tareas se migran hacia participantes menos cargados (P1 y P2).

Figura 5.2: Reasignación de carga en un DSMS de cuatro participantes

como

$$\forall i \in N, D_i : \{ConjuntoTareas_i \subseteq K\} \rightarrow \Re \tag{5.1}$$

Donde $ConjuntoTareas_i$ es un subconjunto de tareas en K corriendo en i. D_i es el costo total incurrido por i para correr su $ConjuntoTareas$. Este costo depende de la carga, $carga(ConjuntoTareas_i)$, impuesta por las tareas. Desde la perspectiva de la teoría de juegos, la función de costo puede verse como el *tipo* de cada participante, es la información privada que determina completamente la preferencia de un participante por las diferentes asignaciones de carga.

Cada participante monitorea su propia carga y computa su costo de procesamiento. A su vez cada uno podría tener una función de costo diferente. Sin embargo, asumiremos que este costo es una *función convexa*[1] y *monótonamente creciente*. De hecho, para muchas aplicaciones que procesan mensajes *e.g.*, los streams de tuplas, una función de costo útil es la demora de procesamiento por mensaje. Para la mayoría de las disciplinas que emplean *scheduling* este costo para la carga ofrecida es una función monótonamente creciente y convexa, pues hay dificultad creciente al ofrecer servicio rápido (de baja latencia) bajo una carga alta. La Figura 5.3 ilustra una función de costo de este tipo para el caso de un único recurso.

[1]Las funciones convexas también se conocen con el nombre de *cóncavas hacia arriba*, http://en.wikipedia.org/wiki/Convex_function.

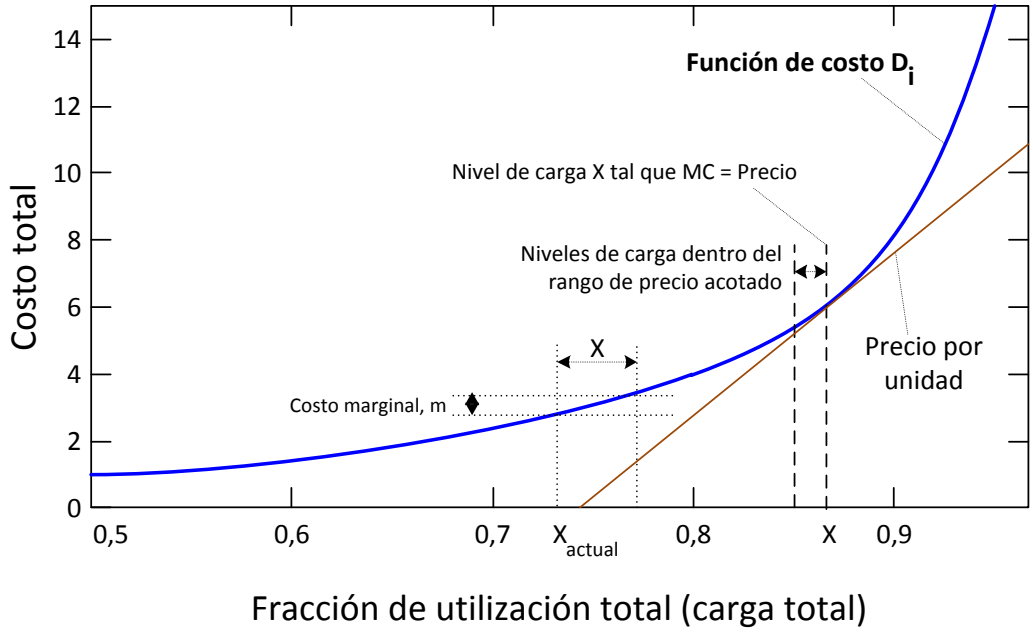


FIGURA 5.3: Precios y costos de procesamiento

Denotamos el *costo incremental* o *costo marginal* para el nodo i al correr la tarea u dado su $ConjuntoTareas_i$ como:

$$\forall i \in N, MC_i : \{(u, ConjuntoTareas_i) | ConjuntoTareas_i \subseteq K, u \in \{K - ConjuntoTareas_i\}\} \rightarrow \Re \quad (5.2)$$

La Figura 5.3 muestra el costo marginal m, causado por el agregado de la carga x cuando la carga actual es X_{act}. Asumiendo que el conjunto de tareas en $ConjuntoTareas_i$ impone una carga total X_{act} y u impone una carga x, entonces $MC(u, ConjuntoTareas_i) = m$. Si x es una unidad de carga, llamaremos m al *costo marginal* de la unidad. Más adelante discutiremos cómo los nodos computan el costo marginal para determinar cuándo es conveniente ofrecer o aceptar un conjunto de tareas.

En la introducción del capítulo mencionamos que los participantes de sistemas federados no son cooperativos sino que tratan de maximizar su utilidad, por eso asumiremos que son *egoístas*. Su utilidad $u_i(ConjuntoTareas_i)$ se computa como la diferencia entre el pago $p_i(ConjuntoTareas_i)$ que recibe el participante i por procesar tareas y el costo de procesamiento $D_i(ConjuntoTareas_i)$ en el que incurre:

$$u_i(ConjuntoTareas_i) = p_i(ConjuntoTareas_i) - D_i(ConjuntoTareas_i) \quad (5.3)$$

Asumimos que el cliente que envía una tarea a un participante debe pagarle. Más adelante analizaremos el pago por el procesamiento que debe efectuarse cuando los participantes migran carga de un lugar a otro.

También suponemos que cada participante tiene un nivel de carga máximo predefinido, T_i, que corresponde al máximo costo de procesamiento $D_i(T_i)$, por encima del cual el participante i se considera asimismo sobrecargado. Usualmente diremos que T_i es la capacidad del participante, sin embargo los participantes pueden seleccionar cualquier nivel de carga por debajo de su capacidad como su nivel de carga máximo deseado.

5.1.3 Correspondencia de elección social

En contraste con propuestas previas argumentamos que no es necesario obtener un balance de carga óptimo en un sistema federado. Cuando un participante se encuentra poco cargado puede proveer buen servicio a sus clientes; en ese caso mover carga[2] hacia otros nodos no mejorará demasiado la performance global del sistema, por el contrario agregará *overhead*. Sin embargo, si un participante está fuertemente cargado (sobrecargado) la performance puede degradarse significativamente.

Usualmente los participantes cuentan con recursos suficientes para manejar su propia carga, pero pueden surgir picos de carga donde la carga total excede significativamente la carga usual. Para solucionar el problema de los picos de carga, un participante puede actuar precavidamente para manejar estos picos esporádicos, o puede colaborar con otros participantes durante la sobrecarga.

El objetivo general es posibilitar que los participantes redistribuyan el exceso de carga, en particular el objetivo de MPA es asegurar que no haya participantes sobrecargados siempre que exista capacidad suficiente. Si el sistema completo está sobrecargado, el objetivo es utilizar tanta capacidad disponible como sea posible. Una asignación de carga será *aceptable* si satisface las dos propiedades que a continuación mencionaremos.

Resumiendo, el objetivo de MPA es implementar una correspondencia de elección social[3] cuyo resultado sea siempre asignaciones aceptables.

Definición 5.1 *Asignación aceptable*

Una *asignación aceptable* es una distribución de tareas donde (1) no hay ningún participante que se encuentre por sobre su umbral de capacidad, o (2) donde todos los participantes alcanzan (o sobrepasan) sus umbrales de capacidad si la carga total ofrecida excede la suma de los umbrales de capacidad. ■

[2]En este trabajo se utilizan los términos *mover carga* y *migrar carga* en forma intercambiable.

[3]Dados los tipos de agentes, *i.e.,* sus funciones de costo, la correspondencia de elección social selecciona un conjunto de alternativas de asignaciones de carga y pagos de participantes.

Formalmente, una asignación aceptable satisface:

$$D_i(ConjuntoTareas_i) \begin{cases} \leq D_i T_i : & \forall i \in N, \text{si el sistema federado tiene carga baja}, o \\ \geq D_i T_i : & \forall i \in N, \text{si el sistema federado tiene carga alta} \end{cases} \tag{5.4}$$

Por ejemplo, tomando como punto de partida el diagrama de consulta de la Figura 5.1 vemos que durante la mayor parte del tiempo cada uno de los cuatro participantes maneja su propia carga sin problemas. Cuando el *rate* de entrada aumenta significativamente en los streams de entrada $S3$ y $S4$ la carga total puede exceder la capacidad de los participantes $P3$ y $P4$ y por lo tanto preferiríamos que otros participantes manejasen parte del exceso de carga mientras dure la sobrecarga (Figura 5.2).

5.2 Descripción de MPA

A continuación presentaremos una descripción general de MPA. En las subsecciones siguientes discutiremos los detalles de cada componente de nuestro sistema.

El objetivo de nuestro mecanismo es implementar una solución global óptima a un problema de optimización descentralizada, donde cada agente tiene un parámetro de entrada al problema y prefiere ciertas soluciones por sobre otras.

Los agentes son los participantes y sus funciones de costo, capacidades y tareas son los parámetros de optimización. El objetivo global del sistema es lograr una asignación aceptable, mientras cada participante trata de optimizar su utilidad según su capacidad predefinida. El conjunto de tareas cambia con el tiempo y por ende el problema de la asignación constituye una optimización *online*. Dado que el sistema es una federación de participantes débilmente acoplados, una única entidad es incapaz de tomar el rol de optimizador central y como consecuencia el mecanismo debe ser necesariamente distribuido.

A continuación presentaremos el mecanismo de precio fijo, el cual asume que la carga es fija. Extenderemos el mecanismo con carga dinámica en la Sección 5.4 y a precios acotados en la Sección 5.6.

Para implementar un mecanismo de precio fijo debemos definir: (1) un conjunto S de estrategias disponibles a los participantes, *i.e.,* la secuencia de acciones que pueden seguir, y (2) un método que permita seleccionar el resultado a partir de un conjunto de estrategias elegidas por los participantes. El resultado es la asignación final de tareas a participantes. El método es una regla de resultado, $g : S^N \rightarrow O$, que mapea cada posible combinación de estrategias adoptadas por los N participantes

en un resultado O.

El mecanismo propuesto es indirecto: los participantes revelan sus costos y tareas indirectamente, es decir, ofreciendo y aceptando tareas en lugar de anunciando sus costos directamente a un optimizador central o a otros participantes. Adicionalmente los agentes pagan a otros por la carga que procesan. Nuestro mecanismo se basa en contratos de precio fijo:

Definición 5.2 *Contrato de precio fijo*
Un *contrato de precio fijo* $C_{i,j}$ entre los participantes i y j define un precio $PrecioFijo$ $(C_{i,j})$ por el cual el participante i debe pagar a j por cada unidad de recurso que i compre en tiempo de ejecución, *i.e.,* para cada unidad de carga movida desde i a j.
■

Los participantes establecen los contratos offline, en tiempo de ejecución pueden *transferir carga* sólo aquellos nodos que hayan establecido un contrato con el otro. A partir de su nivel de carga cada participante decide aceptar o no un conjunto de tareas, ctm (conjunto de tareas a migrar), que será transferido de un nodo a otro. El participante que ofrece la carga paga a su colega la suma de $PrecioFijo(C_{i,j}) * carga(ctm)$, éste pago es proporcional a la cantidad de recursos que la tarea requiere. Esto refleja que i compra recursos de j para ejecutar tareas específicas que necesitan estos recursos. Por lo tanto, los *partners* determinan un precio offline pero negocian en tiempo de ejecución la cantidad de recursos que un partner compra a otro.

Nuestra propuesta es entonces que los nodos participen de dos juegos en diferentes escalas de tiempo: uno offline donde se negocian contratos y otro en tiempo de ejecución donde se efectúan los movimientos de carga.

En el juego offline, los participantes establecen los contratos bajo las siguientes asunciones: (1) los participantes son idénticos, (2) la estrategia de un participante está compuesta por el número de contratos que elige establecer y por el precio que negocia para cada contrato. El resultado de este juego es un *grafo conexo* de contratos en el cual los nodos son los participantes y los vértices entre nodos representan la existencia de un contrato entre los correspondientes pares.

Las reglas propuestas para la negociación de contratos son las siguientes:

- Los dos participantes acuerdan en qué significa una unidad de procesamiento, ancho de banda, o cualquier otro recurso.
- Distintos pares de participantes pueden tener contratos especificando diferentes precios de unidad.

- Cada contrato se aplica en una única dirección.
- Existe a lo sumo un contrato por cada par de participantes en cada dirección.
- Los participantes pueden periódicamente renegociar, establecer o terminar contratos offline.

Denominaremos ccp al conjunto de contratos que mantiene un participante y C denotará el número máximo de contratos que tiene un participante. Los contratos pueden contener cláusulas adicionales como por ejemplo el conjunto de tareas que pueden migrarse, performance mínima requerida, seguridad, disponibilidad, etc. Estas cláusulas adicionales se discutirán en las Secciones 5.5 y 5.6.

En el juego en tiempo de ejecución, los participantes mueven carga a sus partners. En un ambiente de contratos de precio fijo, el conjunto de acciones disponibles a los participantes comprende las siguientes tres acciones:

1. Ofrecer carga al precio prenegociado.
2. Aceptar carga al precio prenegociado.
3. Ninguna de las dos acciones anteriores.

La estrategia de cada participante determina cuando efectúa cada una de las acciones anteriores [4]. No consideraremos el problema del ordenamiento de los contratos, es decir a cuál partner se le debe ofrecer primero la carga. Este orden usualmente se define offline a partir de las relaciones entre participantes.

El resultado global del sistema deseable es una asignación aceptable, donde la secuencia de movimientos de carga en tiempo de ejecución define la asignación de tareas final, *resultado del juego*.

Las reglas del juego online son:

1. Los participantes sólo pueden mover carga a los partners con los cuales establecieron un contrato y deben pagarle al otro el precio contratado.
2. Cada vez que un participante ofrezca un conjunto de tareas a un dado precio significa que la oferta es vinculante[5].

[4] Nuestro sistema funciona independientemente de la estrategia empleada por los agentes para seleccionar el conjunto de tareas que ofrecen o aceptan. Para simplificar el análisis excluiremos el problema de la selección de tareas del espacio de estrategias (*i.e.,* del conjunto de todas las posibles combinaciones de estrategias seleccionadas por los participantes)

[5] **Oferta vinculante:** la oferta está sujeta a una obligación de cumplimiento.

3. Si el partner acepta entonces el participante que inició el movimiento debe pagar el precio ofrecido. En la Sección 5.3 mostraremos que esta elección de partner ayuda a que los participantes puedan *sobrevender* sus recursos.

Continuaremos la discusión de los movimientos de carga en tiempo de ejecución en las Secciones 5.3 y 5.4.

Es posible que los participantes no deseen mover ciertas tareas hacia ciertos partners debido a cuestiones de confidencialidad de los datos procesados o porque las tareas por si mismas tienen suficiente valor intelectual. Por esta razón los contratos pueden especificar el conjunto (o tipo) de tareas que pueden migrarse a partir de las restricciones de selección de tareas en tiempo de ejecución. En los acuerdos online, los participantes pueden también evitar que sus partners (dado que MPA se basa en acuerdos entre pares) muevan sus operadores agregándoles restricciones a sus selecciones de tareas. Por simplicidad ignoraremos este punto en el resto de la sección.

Para asegurar que un partner pueda proveer los recursos necesarios para procesar una tarea, los contratos pueden también especificar un mínimo de latencia de procesamiento de mensajes o cualquier otra métrica de performance y como consecuencia el partner deberá cumplir con estas restricciones o pagar una penalidad monetaria. Este tipo de restricciones es usual cuando se emplean SLAs, por ejemplo para grid computing [SGMvM03], hosting de aplicaciones web [BSC01, FV02], y web services [GS05]. Para lograr que estas restricciones se cumplan emplearemos infraestructuras que utilizan verificación automática [BSC01, KL03, GS05, SGMvM03]. Para evitar que se rompan contratos cuando la carga aumenta, los participantes pueden priorizar las tareas en curso por sobre las que acaban de llegar.

En resumen, proponemos un conjunto de dos juegos: uno offline donde los participantes negocian contratos que especifican el precio que van a pagar a su partner por el procesamiento de carga y otro en tiempo de ejecución, donde dado un conjunto de contratos, cada participante ofrece y acepta carga pagando o recibiendo el precio contratado.

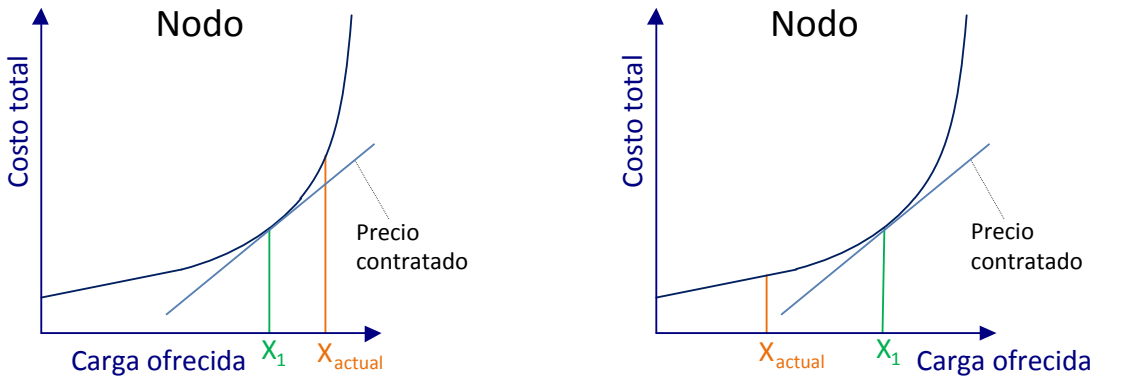
En las siguientes secciones presentaremos los componentes del *Mecanismo de Precio Acotado* y analizaremos sus propiedades. Mostraremos que MPA implementa asignación aceptable en un *equilibrio de Nash Bayesiano*[6], *i.e.,* cuando los participantes adoptan estrategias que optimizan sus utilidades esperadas dadas asunciones precisas acerca de la carga y contratos de los otros participantes.

[6]Información básica al respecto en http://en.wikipedia.org/wiki/Bayesian_game#Bayesian_Nash_equilibr

5.3 Migraciones en tiempo de ejecución

Un participante puede utilizar distintas estrategias (en tiempo de ejecución) para mejorar su utilidad a partir de un conjunto de contratos. Presentaremos a continuación el algoritmo empleado por cada nodo, mediante el cual un participante ofrece o acepta carga con el objetivo de mejorar su utilidad. En la Sección 5.8 mostraremos que bajo ciertas condiciones el algoritmo propuesto constituye la estrategia óptima en un equilibrio Nash Bayesiano.

La estrategia propuesta para contratos de precio fijo se basa en la simple observación de que si el costo marginal por unidad de carga de una tarea es mayor que el precio en un contrato, entonces procesar la tarea localmente es más caro que pagarle al partner por su procesamiento. Como resultado, ofrecer la tarea a un partner potencialmente mejora la utilidad. Inversamente, cuando el costo marginal por unidad de carga de una tarea está por debajo del precio especificado en un contrato, entonces aceptar la carga resulta en un pago mayor (versus incremento de costo) y puede mejorar la utilidad. La Figura 5.4 ilustra ambos casos. En este ejemplo X_{act} es el nivel de carga actual del nodo y X es el nivel del carga mediante el cual el costo marginal de la unidad iguala el precio contratado. Cuando $X_{act} > X$ el costo marginal de la unidad está por encima del precio contratado y por ende ofrecer carga potencialmente mejora la utilidad (Figura 5.4(a)), por el contrario, si $X_{act} < X$ entonces aceptar carga mejora la utilidad, Figura 5.4(b). Migrar carga empleando costos marginales o utilidades marginales es una elección natural empleada por varios sistemas tradicionales [KS89, San93]. Cuando la función de costo es convexa, migrar tareas de un participante con costo marginal alto a uno con menor costo marginal lleva a un gradiente descendiente: cada movimiento de carga decrece estrictamente el costo total de la asignación.



(a) Ofrecer carga puede mejorar la utilidad. (b) Aceptar carga puede mejorar la utilidad.

FIGURA 5.4: Decisiones de movimiento de carga basadas en los costos marginales

Dado un conjunto de contratos, cada participante ejecuta concurrentemente un algoritmo para manejar el exceso de carga (Algoritmo 5.1) y otro para tomar carga adicional (Algoritmo 5.2).

```
0  Procedimiento OfrecerCarga
1  while true do
2     ordenar(ccp según precio(ccp_j) ascendente)
3     foreach contrato C_j ∈ ccp do
4        conj_oferta ← ∅
5        foreach tarea u ∈ conj_tareas do
6           carga_total ← conj_tareas − conj_oferta − {u}
7           if MC(u, carga_total) > carga(u) * precio(C_j)
8              conj_oferta ← conj_oferta ∪ {u}
9           if conj_oferta ≠ ∅
10             oferta ← (precio(C_j), conj_oferta)
11             (resp, conj_acept) ← oferta_envio(j, oferta)
12             if resp = aceptar and conj_acept ≠ ∅
13                transferir(j, precio(C_j), conj_acept)
14                break contratos
15    wait Ω_1 unidades de tiempo
```

Algoritmo 5.1: Algoritmo para manejar el exceso de carga

```
0  Procedimiento AceptarCarga
1  while true do
2     ofertas ← ∅
3     for Ω_2 unidades de tiempo o while (movimiento = true)
4        foreach nueva oferta recibida, nueva_oferta
5           ofertas ← ofertas ∪ {nueva_oferta}
6     ordenar(ofertas según precio(ofertas_i) descendiente)
7     conj_potenciales ← ∅
8     foreach oferta o_i ∈ ofertas
9        conj_acept ← ∅
10       foreach tarea u ∈ conj_oferta(o_i)
11          carga_total ← conj_tareas ∪ conj_potenciales ∪ conj_acept
12          if MC(u, carga_total) < carga(u) * precio(o_i)
13             conj_acept ← ∪ {u}
14       if conj_acept = ∅
15          conj_potenciales ← conj_potenciales ∪ conj_acept
16          resp ← (aceptar, conj_acept)
17          movimiento ← true
18       else resp ← (rechazar, ∅)
19       responder(o_i, resp)
```

Algoritmo 5.2: Algoritmo para tomar carga adicional

Cuando un participante se encuentra sobrecargado debe deshacerse del exceso de carga; para ello selecciona un conjunto maximal de tareas a partir de su $\mathsf{conj\_tareas}_i$

tal que cueste más procesarlo localmente que lo que costaría si lo procesase alguno de sus partners y se lo ofrece a ese dado partner. Los participantes pueden emplear diversos algoritmos y políticas para seleccionar estas tareas.

En el Algoritmo 5.1 presentamos un algoritmo general que realiza "uniones" entre productores y consumidores, si el partner acepta todas las tareas ofrecidas o incluso un subconjunto de las mismas, éstas se transfieren y el participante que está bajando su nivel de carga debe pagarle a quien se hará cargo de sus tareas migradas.

Un participante sobrecargado podría considerar sus contratos en cualquier orden, sin embargo asumiremos que este orden se establece offline. Una posibilidad interesante es considerar primero los contratos de menor precio con la esperanza de pagar menos y mover más tareas. El procedimiento `OfrecerCarga` espera (`wait`) entre transferencias de carga para permitir estimaciones de carga locales, *e.g.,* mediante *exponential moving averages*[7], permite alcanzar el nuevo nivel de carga media. Si no es posible realizar transferencias, el nodo seguirá intentando liberarse de su exceso de carga periódicamente. Alternativamente puede pedirle a sus partners que lo notifiquen cuando sus cargas decrezcan lo suficiente como para aceptar nuevas tareas.

Un participante acepta carga (proveniente de otros partners) siempre y cuando las tareas a recibir sean menos costosas procesadas localmente que el pago ofrecido por el partner. Estas ofertas mejoran la utilidad a medida que aumenta el pago total por encima del costo total de procesamiento. Dado que potencialmente las ofertas pueden llegar al mismo tiempo, los participantes pueden acumular ofertas durante un período limitado de tiempo antes de examinarlas (aunque MPA no requiere esto). Más específicamente en el procedimiento `AceptarCarga` (Algoritmo 5.2) cada participante acumula continuamente ofertas de carga y periódicamente acepta subconjuntos de las tareas ofrecidas, examinando primero las ofertas de mayor precio por unidad de carga. Una alternativa a este comportamiento podría ser aceptar las ofertas que presenten el mayor aumento de utilidad esperado por unidad de carga. Aceptar una oferta conlleva a una migración de carga, pues las ofertas se envían a un partner por vez. Los participantes que aceptan una oferta de carga no pueden cancelar las transferencias y devolver las tareas, sin embargo si pueden utilizar sus propios contratos para volver a mover la carga (incluso potencialmente devolverla).

La Figura 5.5 ilustra tres escenarios de movimiento de carga. En una migración

[7]Técnica también conocida como *exponentially weighted moving averages*, donde se aplican factores de peso que se decrementan exponencialmente. Ver información básica al respecto en http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average.

de carga, A puede transferir a B todas las tareas cuyo costo marginal unitario sea mayor que el precio del contrato (escenarios 1 y 2). Se transferirán solamente aquellas tareas que presenten un costo marginal por unidad de carga en B que no exceda el precio contratado (escenario 3).

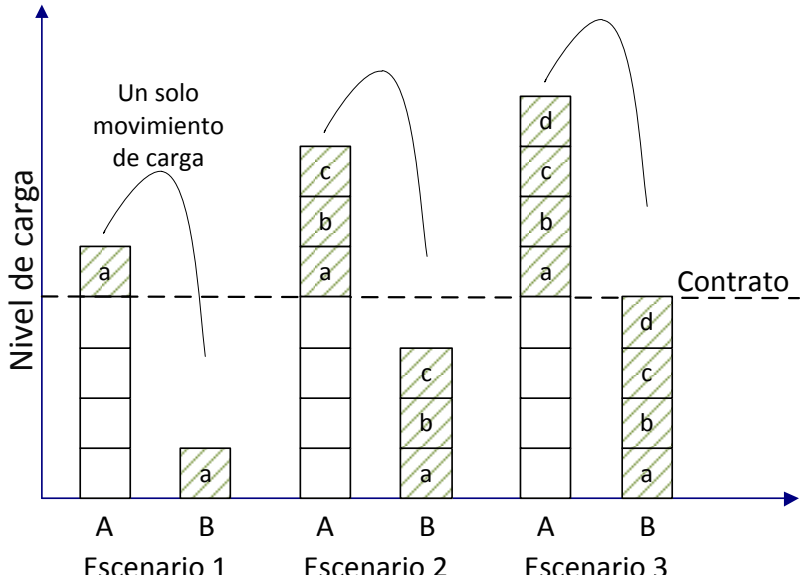


Figura 5.5: Tres escenarios de movimiento de carga para dos partners

Bajo un entorno de contratos de precios fijos pueden pensarse otros posibles protocolos, como por ejemplo ofrecer carga simultáneamente a todos los partners. Si se tiene un único participante sobrecargado este método converge más velozmente en el peor caso, pero al costo de un gran overhead en las comunicaciones dado que se realizan C (el número de contratos) ofertas por cada movimiento de carga. Ofrecer carga simultáneamente a muchos partners trae aparejado el problema de que estos partners no son capaces de saber cuales tareas recibirán de entre todas las que acepten, evitando entonces que puedan determinar que tantos movimientos rentables pueden efectuar sin sobrevender sus propios recursos. En contraste, en presencia de una carga estática, MPA permite que los participantes acepten varias ofertas simultáneamente sin riesgos de sobreventa, mientras al mismo tiempo evita el overhead de comunicación extra dado que los participantes envían las ofertas hacia un partner por vez.

En esta sección mostramos una estrategia para migrar carga en tiempo de ejecución asumiendo un sistema con carga estática. Esta estrategia se basa en dos protocolos simples para emplear contratos en tiempo de ejecución de forma tal de mejorar la utilidad. En la próxima sección extenderemos el algoritmo para que contemple carga dinámica.

5.4 Condiciones de carga dinámica

Las decisiones de movimiento de carga de los participantes se ven afectadas por las variaciones que se producen en la carga con el correr del tiempo y como consecuencia de ello es de nuestro interés extender los contratos de precio fijo para incluir, además de un precio, una unidad de duración de movimiento de carga. De esta forma los participantes pueden ahora tomar mejores decisiones respecto a ofrecer o aceptar carga.

A partir de una distribución de carga determinada los participantes migran su carga a sus partners mejorando sus utilidades individuales y produciendo asignaciones cada vez menos costosas. Un cambio repentino en la carga total del sistema modificará abruptamente la utilidad de uno o más participantes, quien o quienes podrán nuevamente mejorar sus utilidades moviendo carga. Por lo tanto un pico de carga se traduce en recomenzar la convergencia hacia una asignación aceptable a partir de una nueva distribución inicial de carga.

Existen dos problemas en nuestro planteo para manejar adecuadamente las variaciones de carga. En el mecanismo actual, una vez que la carga migra lo hace en forma definitiva y por ende sucesivas variaciones de carga pueden provocar situaciones indeseables, donde potencialmente todos los participantes se encuentran corriendo tareas de otros pares en lugar de las propias. Más aún, es posible que los participantes tengan ciertas expectativas acerca de la duración de su sobrecarga y no deseen mover carga si esperan que la sobrecarga tenga poca duración. Es posible reducir estos problemas modelando los aumentos y decrementos de carga como llegada y partida de tareas. Este modelo es de difícil implementación en la práctica debido a que requiere mecanismos adicionales (no siempre disponibles) para aislar y mover solamente el exceso de carga. Adicionalmente los operadores acumulan estado temporal que necesita ser devuelto luego de que la carga se regulariza.

En lugar de extender nuestra definición de contrato para solucionar estos problemas proponemos incluir una unidad de duración d para cada movimiento de carga:

Definición 5.3 *(revisada) Contrato de precio fijo*
Un *contrato de precio fijo* $C_{i,j}$ entre los participantes i y j define una unidad de duración d para cada movimiento de carga y un precio $PrecioFijo(C_{i,j})$ por el cual el participante i debe pagar a j por cada unidad de recurso que i compre en tiempo de ejecución, *i.e.,* para cada unidad de carga movida desde i a j. ■

La carga siempre se mueve durante un tiempo predefinido d, transcurrido el mismo cualquier participante puede solicitar que la carga sea devuelta. Obviamente

los partners pueden estar de acuerdo en que todavía es beneficioso para ambos dejar la carga donde se encuentra por otro período d de tiempo. Para limitar la frecuencia de ofertas de carga incluiremos la siguiente restricción: si un participante rechaza carga entonces su partner puede volver a ofrecerle carga únicamente transcurrido un período d de tiempo. Discutiremos esta restricción más en detalle en la Sección 5.8.

Si se tiene carga estática es beneficioso para un participante mover una tarea u hacia un partner j cuando el costo de procesamiento por unidad de carga excede el precio del contrato:

$$MC(u, carga_total) > carga(u) * precio(C_j). \tag{5.5}$$

Cuando la carga ofrecida (*i.e.,* el conjunto de tareas del sistema y la carga impuesta por cualquier tarea) cambia dinámicamente, tanto $carga$ como MC se vuelven funciones del tiempo. Por ejemplo, $carga(u, t)$ es la carga impuesta por la tarea u en el tiempo t, y $MC(u, carga_total, t)$ es el costo marginar de la tarea u en el tiempo t.

En el escenario de carga dinámica para que un participante pueda tomar la decisión de ofrecer o aceptar carga necesita comparar su costo marginal esperado para una tarea durante el próximo período de tiempo d y su pago esperado por esa tarea. Entonces un participante debería migrar carga solamente cuando este movimiento redundase en una mejora en la utilidad de carga esperada, por ejemplo el participante podría ofrecer carga cuando [8]:

$$\int_{t=a}^{t=a+d} E[MC(u, carga\_total, t)]\, dt > \int_{t=a}^{t=a+d} E[carga(u, t)] * precio(C_j)\, dt \tag{5.6}$$

Asimismo un participante debería aceptar carga durante el período de tiempo d cuando el costo marginal esperado para la carga está por debajo del pago esperado para la misma carga:

$$\int_{t=a}^{t=a+d} E[MC(u, carga\_total, t)]\, dt < \int_{t=a}^{t=a+d} E[carga(u, t)] * precio(C_j)\, dt \tag{5.7}$$

En cualquier caso, aunque la carga varíe el precio $precio(C_j)$ permanece fijo.

Los participantes pueden utilizar diferentes técnicas para estimar los niveles de cargas y costos marginales esperados. Si la unidad de período de tiempo d es pequeña en comparación con la frecuencia de variaciones de carga, entonces los participantes pueden, por ejemplo, asumir que el nivel de carga actual permanecerá fijo durante todo el período d.

[8] En las siguientes ecuaciones a significa $ahora$.

Migrar carga sólo durante un intervalo limitado de tiempo facilita la necesidad de tomar en cuenta el costo de la migración (el cual fue intencionalmente ignorado hasta este momento). Si la sobrecarga esperada para el período de tiempo d es mayor que el precio que un participante debería pagar a un partner más dos veces el costo de la migración, entonces es beneficioso efectuar el movimiento de carga.

La elección del valor apropiado para d constituye un problema interesante, pues debe ser lo suficientemente grande como para obtener ganancias del movimiento de carga a pesar del overhead del costo de la migración; de otra forma puede suceder que nunca valga la pena mover carga. Por otro lado d no debe ser demasiado grande porque la probabilidad de que la carga no persista durante todo el intervalo de tiempo aumente con d. En este trabajo asumimos que d tiene algún valor fijo obtenido mediante un estudio empírico del sistema particular.

5.5 Contratos offline de precio fijo

En las secciones previas presentamos estrategias para migración de carga asumiendo que cada participante tiene un conjunto de contratos de precio fijo. A continuación analizaremos el conjunto de contratos que un participante debe establecer para tratar de evitar sobrecargas y mejorar su utilidad.

Para simplificar el análisis asumimos un sistema con participantes idénticos con niveles de carga independientes pero que presentan las mismas distribuciones. Además asumimos que los participantes comparten el mismo umbral T de carga empleado para determinar cuándo el nodo está sobrecargado. En trabajos futuros estudiaremos mediante simulaciones valores heterogéneos de T.

Cuando un participante negocia un contrato para deshacerse de carga primero debe determinar su nivel de carga máxima T y el correspondiente costo marginal por unidad de carga. Este costo marginal constituye también el precio por unidad máximo que debe aceptar al negociar un contrato, si excede este valor el participante se arriesga a sobrecargarse y ser incapaz de deshacerse de carga. Un precio elevado es a su vez equivalente a un costo de procesamiento mayor que T, costo que el nodo debe evitar. Un precio menor que el máximo es aceptable, pero generalmente contratos de precios mayores incrementan la probabilidad de que el partner considere productivo aceptar carga. Por lo tanto, la estrategia óptima de un participante para minimizar sus posibilidades de encontrarse sobrecargado es negociar contratos a un precio justo por debajo de T. La Figura 5.3 ilustra para el caso de un único recurso y una función estrictamente convexa como un nivel de carga X se mapea a una unidad de precio. En general este precio es el gradiente de la función de costo evaluada en

X.

Un participante podría tratar de establecer tantos contratos como fuese posible, sin embargo mantener un contrato significa afrontar costos recurrentes debido a las negociaciones offline periódicas. Por ello deben establecerse contratos solamente cuando los beneficios del contrato sobrepasen los costos recurrentes de mantenimiento del mismo. Para calcular el número óptimo de contratos computamos el beneficio acumulativo de los contratos y lo comparamos con los costos de mantenimiento.

Designaremos como B a un participante comprador que quiere deshaciendo de carga mediante la compra de recursos a otros participantes, para B todos los otros participantes del sistema son potenciales vendedores de recursos. A continuación estudiaremos el número óptimo de contratos que B debería establecer.

Desde la perspectiva de B la carga de cada vendedor sigue alguna función de densidad de probabilidad [9], $p(x)$ en el rango $[k, T]$. El comprador no distingue entre un vendedor con una carga T y otro con una carga mayor a T debido a que ninguno de los dos presenta recursos adicionales (libres) para ofrecer. A la función de distribución acumulativa para un dado vendedor i la denotaremos con $F(x) = P(X_i < x)$. El comprador tiene una segunda función de densidad de probabilidad, $g(x)$, para representar sus picos de carga esperados cuando su carga local excede T [10]. Modelaremos los picos de carga y las distribuciones de carga normales mediante dos funciones separadas debido a que los vendedores también intercambian carga entre ellos, pudiendo entonces presentar distribuciones de carga más uniformes dentro de $[k, T]$ que las de un nodo aislado.

Analizaremos distintos tipos de funciones de densidad. Si consideramos una distribución uniforme sobre el intervalo $[k, T]$, cualquier valor dentro del intervalo ocurre con la misma probabilidad: $p_{uniform}(x) = \frac{1}{T-k}$ donde $k \leq x \leq T$ y 0 para cualquier otro caso. La correspondiente distribución acumulativa es:

$$F_{uniform}(x) = P_{uniform}(k \leq X \leq x) = \frac{x-k}{T-k} \quad \text{para } k \leq x \leq T \tag{5.8}$$

Sin embargo, frecuentemente sucede que un participante dentro de $[k, T]$ se encuentra la mayor parte del tiempo muy poco cargado o muy sobrecargado. En ciertos casos, especialmente frente a picos de carga, la distribución puede incluso ser de Larga Cola [11]. De entre todas las posibles distribuciones que pueden emplearse

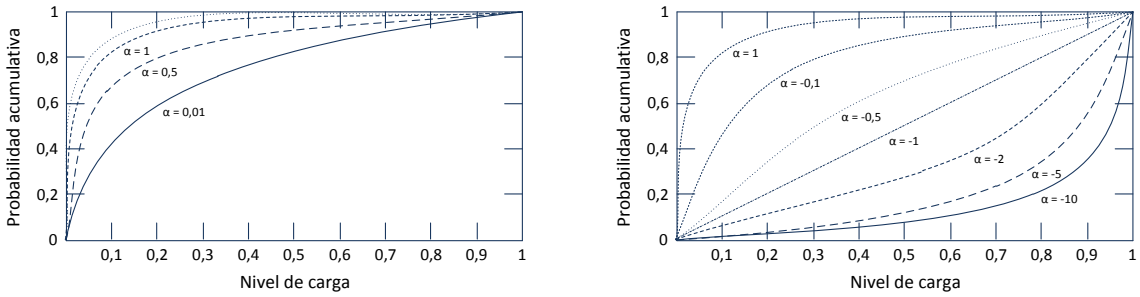
[9] En estadística, la función de densidad de probabilidad (FDP), representada comúnmente como $p(x)$, se utiliza con el propósito de conocer cómo se distribuyen las probabilidades de un suceso o evento, en relación al resultado del mismo.

[10] Es posible incluso que los picos de carga superen a T por varios órdenes de magnitud.

[11] La Larga Cola (*The Long Tail*) es el nombre coloquial para una bien conocida característica de las distribuciones estadísticas (Zipf, Ley de Power, distribuciones de Pareto o/y en general

para modelar estas situaciones de carga usualmente se emplea la función de densidad Pareto acotada: $p(x) = \frac{\alpha}{x^{\alpha+1}} \frac{k^\alpha}{1-(\frac{k}{T})^\alpha}$, para $k \leq x \leq T$ y donde $0 \leq \alpha \leq 2$ es el parámetro *shape*. La correspondiente distribución acumulativa es:

$$\begin{aligned} F_{pareto} &= P_{pareto}(k \leq X \leq x) \\ &= \int_{X=k}^{X=x} \alpha X^{-\alpha-1} \frac{k^\alpha}{1-\left(\frac{k}{T}\right)^\alpha} \, dX \\ &= \left(1 - \left(\frac{k}{x}\right)^\alpha\right) \left(\frac{1}{1-\left(\frac{k}{T}\right)^\alpha}\right) \end{aligned} \tag{5.9}$$



(a) Distribuciones de carga Pareto limitadas.

(b) Distribuciones de carga variando α.

FIGURA 5.6: Distribuciones de carga limitadas con $k = 0{,}01$, $T = 1{,}0$ y diferentes valores de *alpha*

La Figura 5.7(a) ilustra la distribución de carga para $k = 0{,}01$, $T = 1{,}0$ y diferentes valores de *alpha*. La media de la distribución Pareto limitada es:

$$E(x) = \left(\frac{k^\alpha}{1-\left(\frac{k}{T}\right)^\alpha}\right)\left(\frac{\alpha}{\alpha-1}\right)\left(\frac{1}{k^{\alpha-1}} - \frac{1}{T^{\alpha-1}}\right) \tag{5.10}$$

Emplearemos la distribución Pareto para modelar los picos de carga y asumiremos que es posible que los picos excedan el umbral T por varios órdenes de magnitud.

Para analizar distribuciones que no son de Larga Cola emplearemos la misma distribución acumulativa arriba descripta, variando α por un rango más amplio. La Figura 5.7(b) ilustra las distribuciones resultantes. Con un $\alpha = -1{,}0$ la distribución es uniforme, pero al disminuir α aumenta la cantidad de nodos que tienen una carga mayor que la de la distribución uniforme. Aumentar α incrementa la cantidad de nodos que cuentan con una carga más liviana respecto de la distribución uniforme.

distribuciones de Lévy). La característica es también conocida como *heavy tails*, *power-law tails*, o las *colas de Pareto*. En estas distribuciones una amplia frecuencia de población es seguida por una baja frecuencia o baja amplitud de la población que disminuye gradualmente. En muchos casos, los acontecimientos de baja frecuencia o escasa amplitud (la larga cola) pueden abarcar la mayor parte del gráfico.

En ausencia de contratos, si ocurre un pico de carga, el participante queda sobrecargado con una probabilidad de 1.

$$P_{sobrecarga}(0,T) = G(X > 0) = 1 \tag{5.11}$$

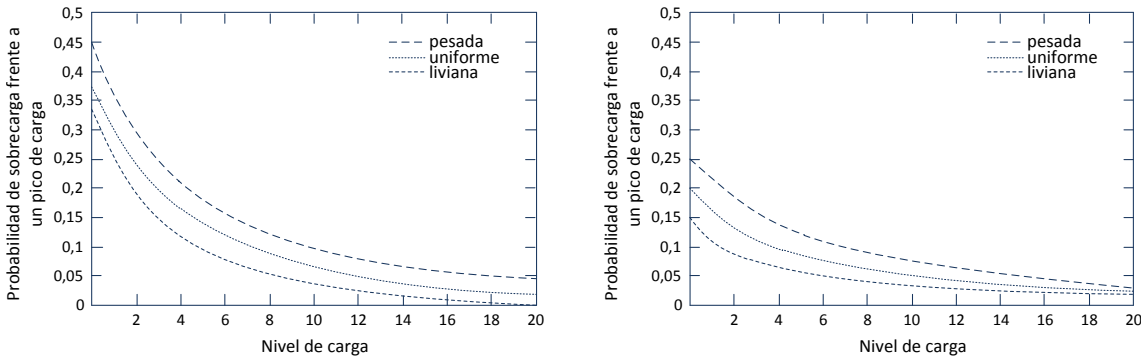
Con un contrato en el umbral T, un participante se encuentra sobrecargado únicamente cuando experimenta un pico de carga que no puede ser absorbido por su partner. Si asumimos que las tareas son de grano fino la probabilidad es

$$P_{sobrecarga}(1,T) \approx G(X > T) + \int_{x=0}^{T} G(X = x)P(X_1 > T - x)\, dx \tag{5.12}$$

Donde $G(X > T)$ es la probabilidad de ocurrencia de un pico de carga que exceda la capacidad total T del partner. La integral computa la probabilidad de que ocurra un pico dentro de la capacidad T del partner pero donde la carga del partner (X_1) es demasiado alta como para poder absorberlo (dado $P(X_1 > T - x)$).

Se puede generalizar esta relación para N contratos. Un participante está sobrecargado cuando experimenta un pico de carga mayor que la capacidad total de sus C partners, $G(X > CT)$, o cuando atraviesa un pequeño pico de carga pero la capacidad disponible total de sus partners es menor que el valor de ese pico:

$$P_{sobrecarga}(C,T) \approx G(X > CT) + \int_{x=0}^{CT} G(X = x)P(X_1 + X_2 + \ldots + X_C > CT - x)\, dx \tag{5.13}$$



(a) Picos que siguen una distribución de Pareto limitada con α=0,14 y rango [0,01, 10T].

(b) Picos que siguen una distribución de Pareto limitada con α=0,5 y rango [0,01, 100T].

Figura 5.7: Probabilidad de sobrecarga frente a un pico de carga

La Figura 5.7 muestra la probabilidad de sobrecarga en seis configuraciones concretas, donde cada subfigura muestra resultados para una distribución diferente para modelar los picos de carga, aunque ambas distribuciones tienen la misma media, T. Cada curva en cada gráfico muestra los resultados para una distribución de carga

diferente en los vendedores. Estas distribuciones presentan diferentes medias, pero en todos los casos la carga varía en el rango [0,01, 1,0]. La distribución "liviana" tiene una media de 0,37 ($\alpha = -0{,}5$), la "uniforme" tiene una media de 0,505 ($\alpha = -1{,}0$) y la "pesada" tiene una media de 0,67 ($\alpha = -2{,}0$). Se efectuaron corridas de una simulación de Monte-Carlo en GNU/Linux (Octave [12]) para computar $P_{sobrecarga}(C, T)$ para valores crecientes de C. En todas las configuraciones el aumento en la cantidad de contratos reduce las probabilidades de sobrecarga durante un pico. Los picos de carga pueden ser de varios órdenes de magnitud más que la capacidad y por ello la probabilidad siempre está por encima del cero, aún teniendo muchos contratos. Es interesante notar que contando con unos pocos contratos (3 a 5 en los ejemplos) se consiguen los mayores beneficios: cuando los picos ocurren distribuidos uniformemente (Figura 5.7(a)) contar con muchos contratos es útil, aunque por otra parte cada contrato adicional conlleva a una mejora cada vez más pequeña.

Los beneficios de un conjunto C de contratos, el cual denominaremos $Beneficio(C)$ no incluye únicamente los ahorros realizados a partir de la reducción de la frecuencia de sobrecarga sino también como una función de reducción de la magnitud esperada de la sobrecarga. La Figura 5.8 muestra dicha magnitud esperada para los mismos experimentos que la Figura 5.7. El beneficio de un conjunto de contratos se computa como:

$$Beneficio(C) = P_{pico}\, f(E[Sobrecarga(0)] - E[Sobrecarga(C)]) \tag{5.14}$$

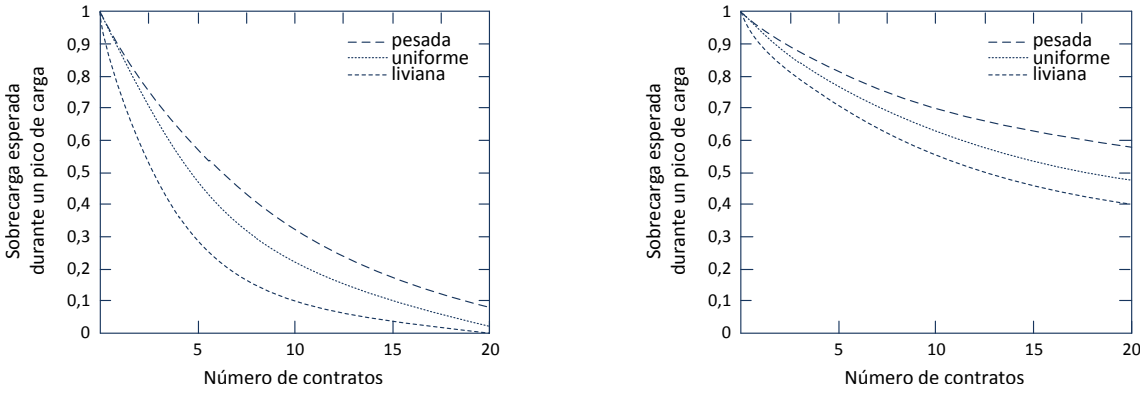


FIGURA 5.8: Magnitud de sobrecarga esperada cuando ocurre un pico de carga

Donde $E[Sobrecarga(0)]$ es la magnitud esperada de la sobrecarga cuando ocurre un pico de carga y el participante no tiene contratos y $E[Sobrecarga(C)]$ representa

[12] GNU Octave (http://www.gnu.org/software/octave) es un lenguaje de alto nivel cuyo propósito principal es el cómputo numérico y es (mayormente) compatible con MatLab.

la sobrecarga esperada con C contratos. La diferencia constituye el ahorro y f es función de los ahorros realizados al migrar el exceso de carga a un partner en lugar de incurrir en costos de procesamiento local. P_{pico} es la probabilidad de que ocurra un pico de carga. Es de interés establecer un contrato cuando el beneficio del contrato sobrepasa el costo de la negociación y el mantenimiento del mismo. La Figura 5.9 muestra el $Beneficio(C)$ de un número creciente de contratos para el conjunto de experimentos mostrado en la Figura 5.7, donde $P_{pico} = 0{,}05$, f es la función identidad y el costo total de un conjunto de contratos crece linealmente con el número de contratos del mismo. Una vez más cada contrato adicional es beneficioso cuando los picos de carga exceden la capacidad de procesamiento en órdenes de magnitud, el beneficio aumenta casi linealmente con el número de contratos. Si los contratos son excesivamente baratos entonces es beneficioso para un comprador establecer un gran número de ellos. Con picos de cargas más leves los primeros contratos son los que ofrecen los mayores beneficios, ya que a medida que el costo de mantenimiento de los contratos aumenta rápidamente se vuelve efectivo mantener sólo unos pocos contratos (menos de 10 en este experimento). Es claro también que si los contratos son demasiado caros puede ser beneficioso no establecer ningún contrato.

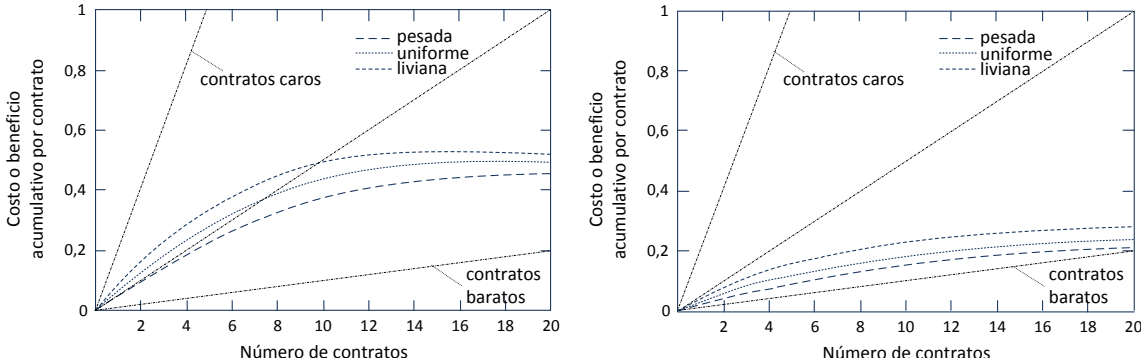


Figura 5.9: Ejemplo de costo y beneficio producto de un aumento en el número de contratos de precio fijo

El cómputo en la dirección opuesta de los contratos es diferente, pues cuando un participante actúa como vendedor cada contrato que establece provoca que eventualmente reciba carga adicional. Cada comprador experimenta sobrecargas con una probabilidad pequeña, P_{pico}, y debido a que el comprador no utiliza todos sus partners por cada pico de carga, un vendedor recibe carga adicional con una probabilidad $P_{carga\_adicional} < P_{pico}$. Como la probabilidad de carga adicional a partir de un solo contrato es pequeña el beneficio acumulativo de los contratos crece linealmente con su número. Si el beneficio generado por un contrato es mayor que el costo de mantenerlo un participante puede obtener ganancias estableciendo contratos como

vendedor. Sin embargo, si el número de contratos es demasiado grande, eventualmente el vendedor estará siempre sobrecargado y el beneficio de establecer contratos adicionales disminuye.

Un participante puede establecer contratos adicionales comprando recursos a un precio menor a T y usarlos no para absorber sus picos de carga sino para reducir sus costos totales de procesamiento. Similarmente un participante puede establecer contratos adicionales para vender recursos a un precio menor a T. En ambos caos los nodos maximizan los beneficios de sus contratos maximizando la ganancia por cada movimiento de carga y por la frecuencia de los mismos.

La Figura 5.10 muestra la cantidad esperada de recursos que un comprador va a adquirir y el ahorro esperado que logrará a partir de un conjunto de contratos con niveles de carga βT con $T = 1{,}0$ y $\beta \in \{0{,}25, 0{,}33, 0{,}5, 0{,}66, 0{,}75\}$. La figura muestra resultados para tres distribuciones de carga diferente:

Uniforme. $\alpha = -1{,}0$, rango $[0{,}01, 1{,}0]$.

Liviana. $\alpha = -0{,}5$, rango $[0{,}01, 1{,}0]$.

Pesada. $\alpha = -2{,}0$, rango $[0{,}01, 1{,}0]$.

Para el cómputo del ahorro utilizaremos a modo de ejemplo una función monótona creciente simple de costo: $\frac{\rho - 0{,}2}{1 - \rho + 0{,}2}$.

Empleando pocos contratos (hasta un máximo de 3), el precio medio (en el sentido de mediana estadística), *i.e.,* 0,5 para uniforme, 0,3 para liviana y 0,7 para pesada logra la mayor cantidad de recursos movidos. Con más contratos un precio ligeramente por debajo de la mediana resulta en más recursos movidos porque el participante se encuentra más frecuentemente en una posición de compra de recursos mientras haya al menos un vendedor con recursos extras disponibles. Los precios por debajo de la mediana también producen el mayor aumento en la utilidad para el comprador. Cuando los precios son demasiado bajos el beneficio es menor, pues menos son los recursos que pueden ser movidos.

Desde el punto de vista de un vendedor los gráficos se invierten, los precios ligeramente por encima de la mediana son los que brindan el mayor beneficio. Dado que ni el comprador ni el vendedor tienen algún incentivo para conceder más rápidamente durante la negociación y debido a que ambos se beneficiarán en mayor medida con un contrato al valor del precio medio puede esperarse que los contratos se establezcan a dicho precio. Con un precio que ronde la mediana bastan de 5 a 6 contratos para obtener un valor cercano al de máximo beneficio.

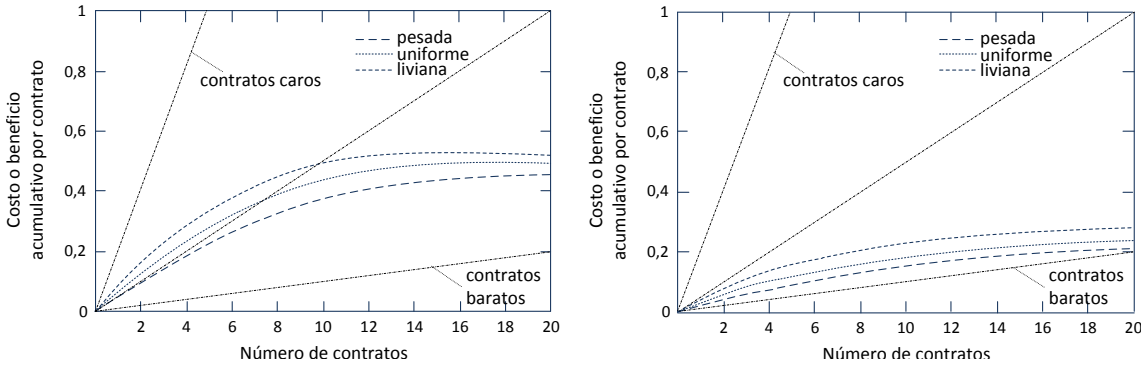


Figura 5.10: Beneficios para el comprador por contratos por debajo del umbral T

La Tabla 5.1 resume el número óptimo de contratos de precio fijo que un participante debería tratar de establecer. Para los compradores de recursos, los primeros contratos (aproximadamente cinco) son suficientes para obtener la mayor parte de los potenciales beneficios. Cada contrato adicional provee solamente un pequeño incremento en la utilidad. Sin embargo, si los picos de carga superan en órdenes de magnitud la capacidad de procesamiento mientras los precios son extremadamente baratos entonces el comprador debería establecer tantos contratos como le sea posible. Para los vendedores de recursos, si los contratos tienen precios bajos, es beneficioso establecer sólo unos pocos contratos. Para aquellos contratos donde el precio corresponde a la capacidad máxima, la utilidad aumenta casi linealmente con el número de los mismos, haciendo que sea redituable contar con muchos contratos.

Nivel de carga correspondiente al precio del contrato	**Dirección del contrato**	**Cantidad de contratos**
Carga máxima	Comprador	Pocos
Carga media	Comprador	Pocos
Carga máxima	Vendedor	Muchos
Carga media	Vendedor	Pocos

Tabla 5.1: Heurísticas para establecer contratos offline

5.6 Contratos de precio acotado

En las secciones anteriores presentamos los componentes básicos de MPA. Mostramos cómo establecer y utilizar contratos de precio fijo para migrar carga, posiblemente sólo durante una cantidad limitada de tiempo. Sin embargo, los contratos de precio fijo no siempre producen asignaciones aceptables.

En esta sección mostraremos que extendiendo los contratos de precio fijo para cubrir un pequeño rango de precios podemos garantizar que un sistema con nodos

uniformes converja a asignaciones aceptables en cualquier configuración de contratos y carga. Dado que ahora el precio final no es fijo los participantes deben negociar un precio final dentro del rango preestablecido. Presentaremos un protocolo simple de negociación y mostraremos que en la mayoría de los casos los participantes acuerdan un precio final aún sin negociar.

Los contratos de precio fijo no necesariamente llevan a asignaciones aceptables debido a que la carga no siempre se propaga por una secuencia de nodos. En la Sección 5.5 mostramos que no es beneficioso para un participante establecer contratos de forma tal que sus partners directos puedan siempre absorber su exceso de carga. Los partners siempre pueden utilizar sus propios contratos para mover carga hacia otros nodos (potencialmente a nodos a varios *hops* de distancia), sin embargo estas migraciones no siempre son posibles en todas las configuraciones del grafo de contratos. Una cadena de contratos idénticos es un ejemplo de configuración que evita que la carga se propague. En la Figura 5.12 podemos ver como un nodo livianamente cargado en medio de una cadena acepta nuevas tareas mientras el costo marginal se encuentre estrictamente por debajo del precio contratado. Eventualmente el nodo alcanza su capacidad máxima (definida por el precio contratado) y rechaza carga adicional. No ofrece carga a sus partners (que podrían tener recursos disponibles) debido a que su costo marginal unitario está todavía por debajo que cualquiera de sus precios de contratos. Por lo tanto, si todos los contratos son idénticos, una tarea sólo se puede propagar un hop de distancia desde su origen.

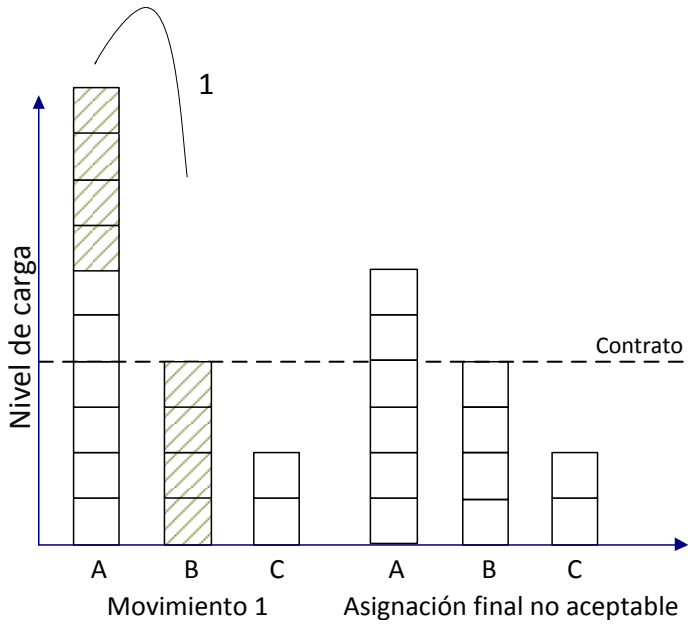


FIGURA 5.11: Contratos de precio fijo no siempre producen asignaciones aceptables

5.6.1 Rango de precio mínimo

Los participantes deben especificar un rango pequeño de precios en los contratos para poder lograr asignaciones aceptables para todas las configuraciones, $[PrecioFijo - \Delta; PrecioFijo]$. A partir de un rango de precios en el contrato los partners negocian el precio final para cada movimiento de carga en tiempo de ejecución. Al permitir esta variación de precios se posibilita la propagación de la carga por una cadena de nodos. De hecho, un participante puede efectuar ahora un reenvío (*forward*) de carga desde un partner sobrecargado hacia otro más ligeramente cargado aceptando tareas a un precio mayor y ofreciéndola a uno menor, *i.e.,* vendiendo recursos a un alto precio y comprándolos a uno bajo. Llamaremos *contratos de precio acotado* a aquellos contratos que especifican un rango de precios en lugar de un precio fijo.

Definición 5.4 *Contrato de precio acotado*
Un contrato de precio acotado $C_{i,j}$ entre los participantes i y j define una unidad de duración d para cada movimiento de carga y un rango de precio $[PrecioFijo - \Delta; PrecioFijo]$ que en tiempo de ejecución limita el precio que será pagado por el participante i al participante j por cada unidad de recurso que compra a j en tiempo de ejecución, *i.e.,* por cada unidad de carga migrada desde i hacia j. ■

Dado que una unidad de precio fijo coincide con el gradiente (o la derivada) de la curva de costo en algún nivel de carga, un rango de precio lo convierte en un intervalo de nivel de carga, como lo ilustra la Figura 5.3. El rango de precio es la diferencia en los gradientes de la curva de costo en los límites del intervalo.

Si el rango de precios es mayor, es más probable que el costo marginal por unidad de los nodos caiga dentro del rango dinámico. Las variaciones de carga dentro del rango de precios contratado crean mayores oportunidades de migración de carga y por ende un mayor rango de precios incrementa la volatilidad de los precios y el número de reasignaciones causadas por pequeñas variaciones de carga. Nuestro objetivo es entonces mantener el rango tan pequeño como sea posible y extenderlo solamente lo suficiente como para asegurar la convergencia hacia asignaciones aceptables.

A continuación derivaremos el rango mínimo de precios que asegura la convergencia hacia asignaciones aceptables y analizaremos una red de nodos homogéneos con contratos idénticos. Queda pendiente para trabajos futuros el análisis por medio de simulaciones de contratos heterogéneos. Para clarificar la exposición asumiremos en el análisis que todas las tareas son idénticas a la tarea migrable de menor tamaño, u, la cual impone la misma carga.

Definimos δ_k como la disminución en la unidad de costo marginal debido a la eliminación de k tareas del $conj\_tareas$ de un nodo:

$$\delta_k(conj\_tareas) = \frac{MC(u, conj\_tareas - u) - MC(u, conj\_tareas - (k+1)u)}{carga(u)} \quad (5.15)$$

δ_k es entonces aproximadamente la diferencia en el gradiente de la función de costo evaluada al nivel de carga actual incluyendo y excluyendo las k tareas. La Figura 5.12 ilustra el concepto de δ_k.

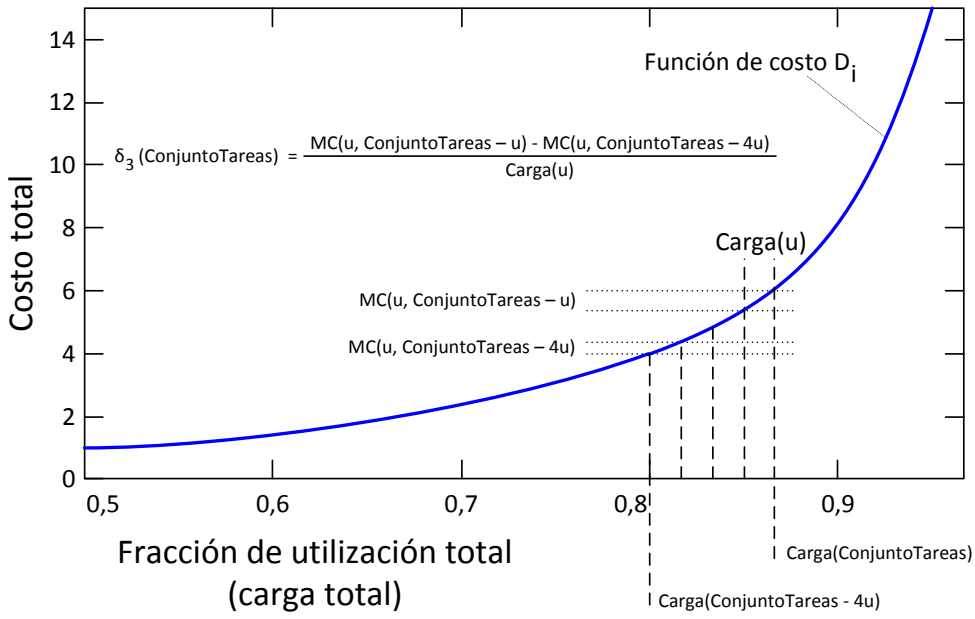


FIGURA 5.12: Ejemplo del cómputo de $\delta_k(conj\_tareas)$ con $k = 3$

Dado un contrato con un precio $PrecioFijo$ definimos $conj\_tareas^F$ como el conjunto maximal de tareas idénticas u que puede ser manejado por un nodo antes de que su costo marginal por unidad exceda el $PrecioFijo$ y dispare una migración de carga. Formalmente $conj\_tareas^F$ satisface $MC(u, conj\_tareas^F - u) \leq carga(u) * PrecioFijo$ y $MC(u, conj\_tareas^F) > carga(u) * PrecioFijo$.

Si todos los contratos del sistema especifican el mismo rango de precios $[PrecioFijo - \Delta, PrecioFijo]$ tal que $\Delta = \delta_1(conj\_tareas^F)$, cualquier tarea ahora puede viajar dos hops desde el nodo donde se originó. La Figura 5.13 muestra como un nodo levemente cargado acepta tareas a $PrecioFijo - \delta_1(conj\_tareas^F)$ hasta que su carga alcanza $conj\_tareas^F - u$. Luego el nodo alterna entre aceptar una tarea u a $PrecioFijo$ y ofrecer una tarea a $PrecioFijo - \delta_1(conj\_tareas^F)$. Similarmente para que viaje carga por una cadena de $M + 1$ nodos (o M transferencia) el rango de precio debe estar por encima de $\delta_{M-1}(conj\_tareas^F)$. El nodo$_j$ en esta cadena

alterna entre aceptar una tarea a $PrecioFijo - \delta_{j-1}(conj\_tareas^F)$ y ofrecerla a un precio de $PrecioFijo - \delta_j(conj\_tareas^F)$.

Lema 5.1 En una red de nodos, tareas y contratos homogéneos para asegurar la convergencia a asignaciones aceptables en un sistema no sobrecargado, el rango de precios en los contratos debe ser al menos
$[PrecioFijo - \delta_{M-1}(conj\_tareas^F), PrecioFijo]$,
donde M es el diámetro de la red de contratos y $conj\_tareas^F$ es el conjunto de tareas que satisface $MC(u, conj\_tareas^F - u) \leq carga(u) * PrecioFijo$ y
$MC(u, conj\_tareas^F) > carga(u) * PrecioFijo$. ■

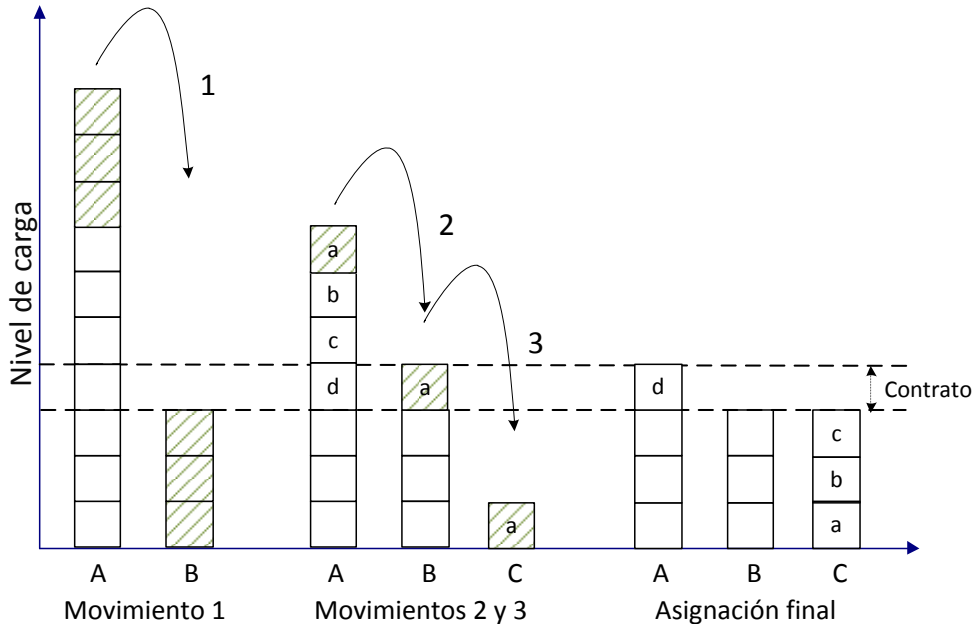


Figura 5.13: Movimientos de carga entre tres nodos empleando un rango pequeño de precios

Condiciones para el Lema: (1) todos los participantes presentan funciones de costo monótamente crecientes y convexas; (2) cualquier nodo puede correr cualquier tarea.

Consideraremos a un sistema livianamente cargado cuando la carga total es menor que la suma de todas las capacidades de los nodos en el límite de menor precio. Para un rango de precio $[PrecioFijo - \delta_{M-1}(conj\_tareas^F), PrecioFijo]$, la capacidad de un nodo al límite del menor precio es $conj\_tareas^F - (M-1)u$.

Demostración: probaremos este lema por contradicción. Supondremos que el nodo N_0 tiene su nivel de carga por encima de su capacidad en la asignación final y mostraremos que N_0 no puede existir en un sistema no sobrecargado con las propiedades mencionadas en el lema.

Por definición de sistema no sobrecargado, mientras N_0 esté por encima de su capacidad y por ende tenga una carga que supere la media, entonces existe al menos un nodo N_M en el sistema que tiene un nivel de carga $conj\_tareas^F - (M-1)u$ (por debajo de la media) que pueda aceptar carga. N_M se encuentra a lo sumo a M (diámetro de la red de contratos) hops de distancia de N_0. Dado que el rango de precio es $\delta_{M-1}(conj\_tareas^F)$ la carga puede propagarse M hops, desde N_0 al nodo más cercano que no se encuentre sobrecargado, N_M, hasta que la carga en N_M alcance $conj\_tareas^F - (M-1)$. Si en este punto la carga de N_0 todavía está por encima de su capacidad entonces debe existir algún otro nodo N_M con un nivel de carga por debajo de $conj\_tareas^F - (M-1)u$ y por lo tanto las migraciones no se detendrían hasta que el nivel de carga de N_0 caiga dentro de su capacidad. Dado que la convergencia sigue un gradiente descendente entonces eventualmente las migraciones de carga se detienen. Por lo tanto N_0 no puede existir en la asignación final. □

Consideraremos a un sistema sobrecargado cuando la carga total es mayor que la suma de las capacidades de los nodos en el límite de menor precio. Cuando el sistema está sobrecargado un rango de precio no lleva a una asignación aceptable donde $\forall i \in N, D_i(conj\_tareas_i) \geq D_i(conj\_tareas_i^F)$; de hecho debido a la dinámica de las migraciones de carga en cadena, en la asignación final algunos participantes pueden tener un costo marginal tan bajo como $PrecioFijo - \delta_M(conj\_tareas_i^F)$[13]. Si un sistema se encuentra sobrecargado el empleo de contratos con rango de precio logra asignaciones cuasiaceptables definidas como:

Definición 5.5 *Asignación cuasiaceptable*
Una asignación cuasiaceptable satisface
$\forall i \in N, D_i(conj\_tareas_i) > D_i(conj\_tareas_i^F - Mu)$. Esto quiere decir que *todos los participantes operan por encima o ligeramente por debajo de su máxima capacidad.*
■

En resumen, un pequeño rango de precios proporcional al diámetro de la red de contratos es suficiente para asegurar que los sistemas no sobrecargados siempre converjan hacia una asignación aceptable y que los sistemas sobrecargados lo hagan hacia asignaciones cuasi-aceptables. A continuación examinaremos cómo los participantes negocian en tiempo de ejecución el precio final.

[13] Si ampliamos los rangos no se mejora este límite.

5.6.2 Negociación del precio final

Mediante el uso de contratos de precio acotado los participantes deben negociar en tiempo de ejecución el precio final dentro del rango contratado. La negociación se efectúa automáticamente con *agentes* que representan a los participantes. En esta sección propondremos un protocolo de negociación simple y eficiente.

El protocolo de negociación se basa en tres observaciones:

a Lograr un acuerdo constituye un mecanismo de *racionalidad individual*[14], *i.e.,* ambos participantes están interesados en lograr un acuerdo porque un movimiento de carga a una unidad de precio dentro de sus respectivos costos marginales incrementa las utilidades de ambos.

b El rango de precios es pequeño, lo cual limita la ganancia máxima que puede obtener un participante al negociar comparado con aceptar el peor precio dentro del rango.

c En la Sección 5.5 mostramos que los participantes mejoran sus utilidades esperadas estableciendo múltiples contratos al mismo precio, creando competencia entre los compradores y los vendedores de recursos. La competencia por contratos que corresponden a la capacidad de carga predefinida T es mayor entre los vendedores que entre los compradores de recursos, pues estos últimos raramente se encuentran sobrecargados. Para los contratos de menores precios la competencia es la misma para ambos tipos de participantes.

A partir de estas tres observaciones proponemos un protocolo de negociación simple que asegura eficiencia, usualmente lograda en un solo paso. El protocolo favorece a los compradores de recursos mediante el incremento de la competencia entre los vendedores de recursos con el objetivo de mejorar la eficiencia (duración) de la negociación.

Asumiendo un rango de precios $[p_L = PrecioFijo - \Delta, p_H = PrecioFijo]$, donde $0 \leq p_L \leq p_H < 1$ y $p_H - p_L \ll 1$, la negociación se efectúa de la siguiente forma:

1. Cuando un comprador desea obtener recursos le ofrece un conjunto de tareas a un vendedor a un precio de $p_1 = p_L$, el menor precio dentro del rango contratado.

2. Si el vendedor acepta la carga ofrecida (o un subconjunto de la carga) entonces la negociación finaliza (la carga migra y el comprador paga p_1).

[14] La propiedad de *racionalidad individual* significa que participar conlleva a una mayor utilidad esperada comparada con la que se obtendría en el caso de no participar.

3. Si el vendedor no acepta (rechaza) el menor precio puede responder con una contraoferta $p_2 \in [p_1, p_H]$ para un subconjunto de las tareas ofrecidas o puede terminar la negociación. El vendedor debe responder pronto (antes de que expire el timeout preestablecido) o el comprador concluirá que el vendedor rechazó la oferta.

4. La contraoferta del vendedor también es vinculante. Si la migración al precio p_2 es beneficiosa para el comprador entonces éste puede mover el subconjunto de la carga aceptada y pagar p_2. En caso contrario el comprador rechaza la contraoferta y la negociación termina. Es claro que el comprador debe responder rápido para evitar que el vendedor asuma que el comprador rechazó su contraoferta.

5. El comprador debe esperar al menos d unidades de tiempo luego de la oferta inicial para tratar de ofrecer carga a $p_1 = p_L$ nuevamente. En este punto comienza un nuevo proceso de negociación.

Todas las ofertas y contraofertas son vinculantes durante un período de tiempo limitado. Las contraofertas hechas por parte de los vendedores deben ser vinculantes durante un tiempo lo suficientemente largo como para posibilitar que el comprador ofrezca carga a todos su partners antes de que la primer contraoferta expire. Si una oferta de un comprador es vinculante durante un tiempo t y posee C contratos, entonces las contraofertas del vendedor deben ser vinculantes durante un tiempo Ct. Con estas restricciones una vez que el comprador recibe una contraoferta entonces puede ofrecer carga a otro partner con el que no ha intentado negociar, o si todos los partners están sobrecargados puede migrar carga hacia el partner que respondió con la contraoferta de menor precio pagando el precio propuesto. Dado que hay múltiples vendedores, que las contraofertas son vinculantes y que a su vez estos no conocen los precios de las contraofertas del resto de los vendedores, entonces la negociación es equivalente a una *subasta en reversa de primer precio (a sobre cerrado) first-price reverse auction (with sealed-bids)*[15]. Se trata de una subasta en reversa porque está conducida por el comprador en lugar del vendedor.

A continuación analizaremos las estrategias disponibles tanto para el comprador como para el vendedor, comenzando por el caso del comprador. Para un dado precio propuesto en un contrato, si las distribuciones de carga en los nodos vendedores son

[15]En una subasta a sobre cerrado o en reversa de primer precio, cada comprador puede realizar solamente una oferta que se formula al mismo tiempo que las de todos los demás, desconociendo qué han ofertado los otros. El recurso se adjudica a la oferta más alta siendo el precio pagado el precio de la oferta. Es más conocida en nuestro medio como *subasta a sobre cerrado*.

iguales o no son conocidas, entonces el comprador puede ofrecer carga a un vendedor elegido aleatoriamente (o podría utilizar un orden predefinido). Si el vendedor rechaza el precio más bajo entonces el comprador debe enviar la misma oferta a otro vendedor, pues es posible que otro acepte al menor precio (p_1). Sin embargo, si todos los vendedores rechazan p_1, la estrategia óptima para el comprador es mover la carga al partner con la contraoferta más baja, pagando el menor precio posible por la carga. En caso de empate el comprador puede elegir al vendedor aleatoriamente o a partir de relaciones offline preestablecidas. La mejor estrategia para el comprador es entonces actuar como subastador en una subasta en reversa.

A continuación examinaremos las estrategias disponibles para el vendedor. Un vendedor puede aceptar el menor precio p_1 o responder con un precio mayor p_2, esperando en este último caso obtener una mayor utilidad. Dado que cada participante establece C contratos entonces hay una probabilidad $1 - p_1^{C-1}$ de que otro vendedor tenga un costo marginal por debajo de p_1 y pueda aceptar p_1 directamente. Esta probabilidad es alta cuando p_1 está cerca de 1 y/o con C grande. Adicionalmente, dado que el rango de precio es pequeño las ganancias potenciales que brinda una contraoferta también son pequeñas y por lo tanto vender recursos a p_1 aumenta la utilidad, *i.e.,* aceptar p_1 directamente constituye frecuentemente la estrategia óptima para el vendedor (discutiremos este escenario en la Sección 5.8, Propiedad 5.2).

Sin embargo, si el vendedor está sobrecargado y su costo marginal es cercano o por encima de p_1, el vendedor maximiza su utilidad esperada haciendo un contraoferta con un precio mayor. Si existe al menos otro vendedor capaz de aceptar p_1 entonces el valor p_2 de la contraoferta carece de importancia, pero si todos los demás vendedores también están sobrecargados entonces todos ellos también pueden proponer contraofertas (o incluso pueden rechazar la oferta). En este caso un p_2 alto aumenta las posibilidades de utilidad pero reduce las posibilidades de obtener la carga debido a lo competencia y porque el comprador puede no ser capaz de aceptar un precio tan alto. El vendedor entonces ofrece p_2 como contraoferta maximizando:

$$E[Utilidad_{Vendedor}] = (p_2 - S)P(p_2) \tag{5.16}$$

Donde S es la valuación del vendedor sobre las tareas ofrecidas, es decir el costo marginal promedio por unidad, $p_2 - S$ es la utilidad del vendedor dada su valuación S y $P(p_2)$ es la probabilidad de ganar la carga con una contraoferta p_2.

A modo de ejemplo asumamos un rango de precios $[0, 1]$ y una distribución uniforme de ofertas que compiten en este rango. El vendedor debe seleccionar su

contraoferta p_2 de forma tal de maximizar su utilidad esperada:

$$E[Utilidad_{Vendedor}] = (p_2 - S)(1 - p_2)^{C-1}(1 - p_2) \tag{5.17}$$

Donde $p_2 - S$ es la utilidad del vendedor, $(1 - p_2)^{C-1}$ es la probabilidad de que ninguno de los $C - 1$ vendedores (competencia) proponga una oferta por debajo de p_2. A su vez, el comprador también puede tener una valuación dentro del rango, se empleará $(1 - p_2)$ para modelar una probabilidad linealmente decreciente de que el comprador acepte el precio. La Tabla 5.2 muestra ejemplos de contraofertas asumiendo entre 1 y 7 vendedores competidores. Con 5 competidores nunca es útil sobremarcar el precio por más de un 15 % del rango total del precio, mostrando que los vendedores producirán contraoferta con poco más de sus valuaciones verdaderas. Es claro que en la práctica un vendedor racional seguiría la estrategia de sobremarcar y produciría contraofertas más altas que su valuación.

Valuación del vendedor, S	0	0.25	0.5	0.75
1 competidor	0.33 (33 %)	0.50 (25 %)	0.67 (17 %)	0.83 (8 %)
3 competidores	0.20 (20 %)	0.40 (15 %)	0.60 (10 %)	0.80 (5 %)
5 competidores	0.14 (14 %)	0.36 (11 %)	0.57 (7 %)	0.79 (4 %)
7 competidores	0.11 (11 %)	0.33 (8 %)	0.55 (5 %)	0.78 (3 %)

TABLA 5.2: Ejemplo de contraofertas (y los correspondientes sobreprecios expresados como porcentaje sobre el rango del precio total) de vendedores con diferentes valuaciones para recursos y diferente número de competidores

Un resultado conocido proveniente de la teoría de subastas [Vic61b] asevera que si un conjunto C de participantes de *riesgo neutral*[16] ofertan por un único ítem y si además sus valuaciones están uniformemente distribuidas en el rango $[0, 1]$ entonces una estrategia que sigue un *equilibrio de Nash*, (es decir la mejor estrategia para cada agente asumiendo que el resto de ellos sigue esta estrategia) es ofertar $\alpha B = \frac{C-1}{C}B$, donde B es la valuación del ítem dada por el cliente.

En una subasta en reversa son los vendedores los que ofertan en lugar de los compradores. Esta situación puede también verse como una subasta común donde los vendedores ofertan el descuento que le hacen al comprador. Si el vendedor oferta su valuación verdadera $S \in [0, 1]$, el descuento al comprador es $1 - S$, *i.e.,* el ahorro comparado con el máximo precio 1. Las ofertas de los vendedores bajo un equilibrio de Nash pueden expresarse como $1 - \frac{C-1}{C}(1 - S)$, en nuestro caso las contraofertas

[16]Un participante de *riesgo neutral* se encuentra entre uno de miedo al riesgo (*risk-averse*) y uno buscador de riesgo (*risk-seeking* o *risk-lover*). Un participante neutral es indiferente al riesgo, *i.e.,* el riesgo no afecta sus decisiones. Por ejemplo apostaría $50 si tiene 0.5 chances de ganar $100 y una chance igual de ganar $0.

deben encontrarse dentro del rango contratado $[p_L, p_H]$. Dada una valuación $S \in [p_L, p_H]$ y asumiendo que una fracción β de los vendedores tiene un costo marginal unitario que cae dentro del rango contratado, entonces la contraoferta debe ser:

$$p_2 = p_H - \frac{\beta C - 1}{\beta C}(p_H - S) \tag{5.18}$$

La igualdad anterior no considera el hecho de que el comprador no pueda ser capaz de aceptar un precio mayor, sin embargo podríamos modelarlo considerando al comprador como un vendedor adicional. El resultado anterior asume que el comprador rompe empates aleatoriamente.

En resumen, asumiendo que el comprador tiene un conjunto de C contratos equivalentes, la estrategia óptima para un vendedor cuyo costo marginal promedio por unidad es S para las tareas ofrecidas que caen dentro del rango de precios contratado es aceptar el menor precio p_1. Cuando el costo marginal promedio por unidad cae dentro del rango contratado, la estrategia óptima del vendedor es ofrecer una contraoferta con un precio p_2, valor ligeramente por encima de S; en caso contrario el vendedor debe rechazar la oferta. En la Sección 5.8 presentaremos la demostración de esta propiedad y posibles valores de β.

El análisis de la estrategia anterior asume que el comprador ofrece una única tarea a los vendedores. Sin embargo, en MPA el comprador ofrece grupos de tareas y mientras el costo marginal unitario del vendedor se encuentre por debajo (p_L), el vendedor puede aceptar el conjunto completo o un subconjunto de tareas al menor precio. Del mismo modo, si su contraoferta es igual al mayor precio del rango el vendedor acepta todas las tareas que mejoran su utilidad. Sin embargo, dentro del rango de precios, un participante puede elegir entre proponer una contraoferta por un precio menor por una única tarea o un mayor precio por un grupo de ellas[17].

En la mayoría de los casos, aun cuando las contraofertas de los participantes se encuentran por encima de sus valuaciones reales se produce una migración de carga y tanto la utilidad del comprador como la del vendedor mejoran.

Una migración de carga potencial falla solamente cuando la valuación del vendedor B se encuentra dentro del rango $[S, p_2]$, donde p_2 es el precio de la contraoferta. Sin embargo, esta situación rara vez ocurre, pues p_2 es apenas mayor que S. Adicionalmente esta falla no es permanente, pues luego de un período de tiempo d el comprador puede reintentar migrar carga al precio mínimo y a partir de la no concreción del movimiento de carga anterior, el vendedor puede actualizar la estimación

[17] Para simplificar nuestra propuesta asumimos que dentro del rango de precios las contraofertas se efectúan por una única tarea.

de la carga del vendedor y así contraofertar con un precio $p'_2 < p_2$. Entonces, mediante el empleo de este protocolo si un movimiento de carga es posible entonces eventualmente ocurre.

El protocolo de negociación presentado es sólo uno de los posibles y por ello MPA trabaja de manera independiente del protocolo empleado en la negociación del precio final.

5.7 Aplicación de MPA a sistemas DSMS federados

Al principio de este capítulo mencionamos que MPA puede aplicarse directamente a un stream de nodos de procesamiento, donde las tareas presentan la propiedad de que los streams de salida de algunas tareas sirven como streams de entrada de otras.

Estas relaciones entre tareas afectan los costos marginales, pues para la misma tarea adicional un participante que ya se encuentra corriendo tareas relacionadas incurre en un menor incremento en el costo de procesamiento que un participante que solo está ejecutando tareas no relacionadas. Por lo tanto, los participantes pueden bajar sus costos de procesamiento (e incluso mejorar la performance global) agrupando tareas relacionadas, prefiriendo ofrecer tareas más débilmente acopladas durante períodos de sobrecarga.

Las relaciones entre las tareas tienen un segundo impacto sobre el sistema: un movimiento de carga de un participante puede afectar el costo de procesamiento de participantes upstream debido a que estos últimos pueden tener que enviar sus streams de salida a diferentes nodos, causando posiblemente que un participante upstream tenga efectuar un movimiento de carga. Sin embargo estas dependencias no causan oscilaciones, pues no hay ciclos en los diagramas de consultas y las dependencias son movimientos downstream unidireccionales que afectan solamente a participantes upstream.

5.8 Propiedades de MPA

En esta sección se presentan las propiedades de MPA, las cuales se resumen en la Tabla 5.3. Para cada propiedad daremos una descripción intuitiva, plantearemos la propiedad y por último la probaremos (bajo ciertas condiciones) o en su defecto expondremos argumentos sólidos.

La propiedad más relevante de MPA es que posibilita que participantes autónomos

Propiedad	Descripción (resumida)
Propiedad 5.1	El mecanismo de precio acotado presenta racionalidad individual.
Teorema 5.1	La propuesta presentada constituye un equilibrio de Nash Bayesiano.
Propiedad 5.2	Usualmente la estrategia óptima para el vendedor es negociar el precio final sólo cuando la valuación cae dentro del rango de precios y contraofertar con un precio apenas superior a la valuación.
Teorema 5.2	Un pequeño rango de precios es suficiente para lograr asignaciones aceptables para sistemas no sobrecargados y cuasi-aceptables en presencia de sistemas sobrecargados.
Propiedad 5.3	El mecanismo tiene baja complejidad: converge rápidamente.
Propiedad 5.4	El mecanismo impone un bajo overhead en las comunicaciones.

Tabla 5.3: Resumen de las propiedades del mecanismo de precio acotado

manejen sus picos de carga colaborativamente mediante el empleo de contratos prenegociados entre pares. Los contratos permiten que los participantes desarrollen o maximicen relaciones de preferencia de larga duración, las cuales gracias a precios acotados prenegociados proveen predictibilidad y escalabilidad en tiempo de ejecución. Las propiedades de MPA distinguen a nuestro protocolo de otras propuestas que emplean mercados y subastas donde los precios y los colaboradores generalmente cambian con mayor frecuencia. Sostenemos que conceptos como privacidad, predictibilidad, estabilidad y confianza en las relaciones establecidas offline hacen de MPA un mejor protocolo en la práctica. En las siguientes secciones veremos de qué forma MPA redistribuye la carga entre los participantes y cuánto overhead se genera en tiempo de ejecución.

Nuestro protocolo provee suficiente incentivo a los agentes para participar en el mecanismo propuesto (*Propiedad 5.1*) y también es beneficioso para el participante seguir la implementación propuesta en las secciones previas (*Teorema 5.1 y Propiedad 5.2*). Dado que los participantes siguen dicha implementación, entonces demostraremos que es suficiente un pequeño rango de precios para asegurar que un sistema uniforme siempre converja hacia una asignación aceptable (*Teorema 5.2*). MPA es también *indirecto* (los participantes revelan sus costos sólo indirectamente cuando aceptan u ofrecen carga a sus partners), *algorítmicamente tratable* (converge a una asignación de carga final en tiempo polinomial, como veremos en la *Propiedad 5.3* y con un overhead polinomial en el peor caso de comunicación de carga que se necesita migrar (*Propiedad 5.4*)) y *distribuido* (no existe un optimizador central).

5.8.1 Asunciones adicionales

Para facilitar el análisis introduciremos tres asunciones. La primera es que los participantes presentan riesgo neutral, están dispuestos a pagar (o incurrir en un costo) para migrar una carga hasta un monto que depende de la ganancia esperada para esa migración. MPA generaliza a los participantes, aún si ellos presentan diferentes actitudes frente al riesgo (pues los participantes utilizan diferentes umbrales para decidir cuando involucrarse en una migración de carga). Los participantes con miedo al riesgo (*risk-averse*) se involucran sólo si las ganancias esperadas exceden los costos por un predeterminado margen (proporcional a los costos esperados), en cambio, los buscadores de riesgo (*risk-seeking* (o también conocidos como *risk-lover*) son capaces de arriesgarse y mover carga aún si las ganancias esperadas menos los costos son negativas. La actitud frente al riesgo también afecta las contraofertas que un participante propone cuando negocia el precio final dentro de un rango contratado. Para simplificar las propiedades ignoraremos estos ajustes de riesgos.

Nuestra segunda asunción es que los participantes evalúan los costos y ganancias esperados sin considerar migraciones de carga futuras. Contando con esta simplificadora asunción un participante nunca acepta una migración de carga que se espera baje su utilidad con la esperanza de que a futuro otra migración le de una mayor ganancia. Esta asunción nos permite reducir y simplificar significativamente el espacio de estrategias de los participantes.

La tercera y última asunción es que cada participante puede estimar correctamente sus condiciones de carga esperadas para el intervalo de tiempo actual d_i y que las condiciones de carga actuales coinciden con las esperadas. Los participantes emplean estas estimaciones para tomar las decisiones sobre los movimientos de carga. Sin embargo, también asumimos que las distribuciones de carga de los participantes son independientes (sin memoria) entre estos intervalos de tiempo, *i.e.,* la carga en el intervalo de tiempo d_{i+1} no depende de la carga presente en el intervalo d_i. Esta asunción nos permite evitar la consideración de estrategias en las que los participantes traten de anticipar sus condiciones de carga futuras o las de sus partners. En la práctica esta propiedad puede mantenerse durante intervalos de tiempo suficientemente largos.

5.8.2 Propiedades

Los participantes no están forzados a seguir nuestro protocolo y por ello parece razonable comenzar mostrando los beneficios de participar en el mecanismo de precio acotado. Intuitivamente la participación es beneficiosa porque cada participante

tiene la opción en tiempo de ejecución de aceptar solamente aquellos movimientos que espera mejoren su utilidad. Más específicamente nuestro mecanismo presenta la siguiente propiedad

Propiedad 5.1 El mecanismo de precio acotado presenta *racionalidad individual*: la utilidad esperada por la participación es mayor que la utilidad de la no participación. ■

Precondición: para cada participante la carga actual para el intervalo de tiempo actual d_i es igual a la carga esperada para ese intervalo de tiempo.

Demostración: Si los participantes eligen formar parte del mecanismo, entonces luego de la negociación sus estrategias en tiempo de ejecución deben restringirse a aceptar y ofrecer carga a un precio que se encuentra dentro del rango contratado.

En el caso de carga estática, los participantes mueven carga hacia un partner o aceptan carga de un partner solamente si al hacerlo incrementan su utilidad (*i.e.,* cuando el costo de procesamiento marginal es estrictamente superior o estrictamente inferior que el precio contratado), entonces es evidente que la participación mejora la utilidad.

Bajo una carga dinámica los participantes migran carga solamente durante un período de tiempo d (*i.e.,* para el intervalo de tiempo actual d_i), luego de dicho tiempo el participante puede solicitar el retorno de la carga, revirtiendo el estado a una asignación que hubiese existido de no haber participado en el mecanismo. Asumiendo que los participantes pueden estimar correctamente su nivel de carga esperado para el intervalo actual d_i (*i.e.,* el nivel de carga actual coincide con el nivel esperado), moverán carga sólo si hacerlo mejora su utilidad esperada para ese período de tiempo, resultando en una utilidad esperada mayor que la que hubiese obtenido en el caso de no haber participado.

Por lo tanto en ambos casos, carga estática y dinámica, los participantes se benefician con la utilización de nuestro mecanismo. □

A continuación exploraremos las estrategias que los participantes deben adoptar para maximizar sus utilidades. Mostraremos que un participante que sigue la implementación propuesta en la Sección 5.3 logra la *estrategia de mejor respuesta*[18] dado el conocimiento de las distribuciones de carga sin memoria de los otros participantes y asumiendo que los participantes están generalmente poco cargados. A continuación consideraremos la negociación en tiempo de ejecución para precio fijo.

[18]La *estrategia de mejor respuesta* es la estrategia que maximiza la utilidad esperada del participante dadas las estrategias (esperadas) adoptadas por el resto de los participantes.

Bajo negociación de precio fijo, MPA constituye una estrategia simple y natural: un participante ofrece carga a sus partners cuando su costo marginal se encuentra por encima del precio contratado y acepta ofertas de carga cuando su costo marginal se encuentra por debajo de ese precio. Intuitivamente esta estrategia es óptima para el caso de un comprador cuando el último trata de deshacerse de carga (en primera instancia) mediante sus contratos más baratos. Esta estrategia es también óptima para un vendedor cuando los participantes están normalmente poco cargados, pues rechazar una oferta de carga es equivalente a perder una oportunidad para mejorar su utilidad durante el período de tiempo actual d_i sin una probabilidad favorable de que se presente una mejor oportunidad durante ese mismo período de tiempo. A continuación formalizaremos el razonamiento anterior para convertirlo en un argumento que aún considerando nuestra simplificación efectúe asunciones razonables acerca de los participantes y sus distribuciones de carga.

En un entorno de precio fijo con dos únicos participantes, la estrategia dominante es simplemente ofrecer o aceptar carga cuando hacerlo implica mejorar la utilidad esperada. Esta estrategia no sólo lleva a la mayor utilidad esperada sino que también es independiente de lo que el resto de los participantes hace.

Para el caso de múltiples participantes y contratos, aún aunque los participantes puedan únicamente ofrecer o aceptar carga al precio contratado, el espacio de estrategias se vuelve más amplio. Los participantes pueden, por ejemplo, tratar de optimizar su utilidad aceptando mucha carga con la esperanza de pasarla a un precio menor. Los participantes pueden asimismo posponer la acción de aceptar u ofrecer carga con la esperanza de lograr una migración más beneficiosa en el futuro. La primer estrategia viola nuestra asunción de que los participantes nunca aceptan migraciones de carga que bajan su utilidad. Mostraremos que la segunda estrategia no mejora la utilidad cuando las distribuciones de carga son independientes entre intervalos de tiempo y los participantes se encuentran usualmente poco cargados. Más específicamente demostraremos el siguiente teorema:

Teorema 5.1 Supongamos que las distribuciones de carga son independientes (sin memoria) y el ordenamiento predefinido de los contratos se hace de forma incremental a partir de los precios de los contratos. Bajo el esquema de precio fijo, ofrecer carga por medio de cualquier contrato que mejore la utilidad y aceptar todas las ofertas que mejoren la utilidad constituyen una metodología que cumple con el *equilibrio de Nash Bayesiano*, *i.e.,* es la *estrategia de mejor respuesta* contra las estrategias esperadas adoptadas por el resto de los participantes. ■

Precondición: para cada participante la carga actual para el intervalo de tiempo actual d_i es igual a la carga esperada para ese intervalo de tiempo. Los participantes deben estar generalmente poco cargados, produciendo ofertas de carga con una probabilidad lo suficientemente pequeña como para asegurar que las ganancias esperadas a partir de una carga esperada sean reducidas. Estas ganancias esperadas deben ser menores a las ganancias que potencialmente se generarían a partir de la migración de una tarea individual de carga a efectuarse en el intervalo d_i mediante cualquier otro contrato.

Demostración: Probaremos la propiedad anterior mostrando que cualquier otra estrategia es incapaz de producir una mayor utilidad esperada. Existen dos estrategias adicionales posibles: efectuar migraciones de carga que potencialmente puedan decrementar la utilidad o rechazar migraciones de carga que potencialmente puedan incrementar la utilidad.

Anteriormente asumimos que un participante nunca lleva a cabo movimientos de carga que disminuyan su utilidad esperada, por lo tanto la única estrategia posible es demorar el ofrecimiento y la aceptación de carga con la esperanza de lograr en el futuro una migración más beneficiosa. Mostraremos que demorar una migración de carga no mejora la utilidad esperada ni para el comprador ni para el vendedor. Comenzaremos por analizar la estrategia del comprador.

Es claro que ofrecer carga cuando al hacerlo se mejora la utilidad esperada constituye una mejor estrategia que procesarla localmente. Asumiendo que un comprador ordena sus contratos en orden creciente en cuanto a precios de contratos entonces la estrategia óptima del comprador es siempre tratar de migrar carga empleando el primer contrato beneficioso. Sin embargo, dado que los precios son fijos, si el partner rechaza la carga el comprador tiene dos opciones: probar con el próximo y potencialmente más caro contrato o esperar que la carga disminuya en el primer partner y reintentar. Como MPA requiere que los participantes reintenten una oferta luego de al menos d unidades de tiempo, el comprador tiene la posibilidad de migrar carga en el intervalo actual de tiempo d_i empleando un contrato potencialmente más caro o continuar procesando carga hasta el próximo intervalo de tiempo d_{i+1}. Si el precio de contrato más caro se encuentra por debajo de la unidad de costo marginal, entonces utilizarlo incrementa la utilidad comparado con el procesamiento de la carga localmente. Por lo tanto, la mejor estrategia para el comprador es intentar todos sus contratos en vez de demorar una migración de carga.

A continuación examinaremos las estrategias del vendedor. Cuando un vendedor recibe una oferta de carga que mejora su utilidad puede aceptarla para el intervalo actual de tiempo d_i o rechazarla con la esperanza de que arribe una oferta de un

contrato más caro durante el mismo período de tiempo. Más precisamente, la oferta debería ser rechazada si el incremento en la utilidad esperada para una oferta probable es mayor que el incremento en la utilidad esperada para una oferta dada. El vendedor podría anticipar una oferta de carga si conociera las condiciones de carga de sus partners, pero dado que las distribuciones de carga no tienen memoria, no es posible que el vendedor pueda deducir estas condiciones.

MPA no permite que un comprador efectúe más de una oferta durante un dado intervalo d_i y por ende el vendedor no podría ser capaz de observar las condiciones de carga actuales sin antes haber recibido una oferta de su partner, por lo tanto el vendedor no puede anticipar las condiciones de carga de sus partners. Dado que asumimos (en las precondiciones del teorema) que los participantes se encuentran en general poco cargados, las ganancias potenciales de las ofertas de carga esperadas están por debajo de las ganancias potenciales de las migraciones de carga concretas, y entonces la mejor estrategia para el vendedor es aceptar la oferta de carga.

Un vendedor podría potencialmente tener un partner que le ofreciese carga en forma determinística independientemente de su nivel de carga y por lo tanto, en este caso en particular, sería posible que el vendedor pudiese anticipar las migraciones de carga y rechazar ofertas beneficiosas. Sin embargo, debido a que un partner que siguiese esta estrategia no estaría cumpliendo con una estrategia de mejor respuesta, entonces el vendedor no puede asumir la existencia de este partner. Por definición de equilibrio de Nash Bayesiano, la estrategia del vendedor debe ser una *mejor respuesta* a las estrategias esperadas de los otros nodos.

Por lo tanto, siguiendo nuestra propuesta de aceptar y ofrecer carga cada vez que ello implique mejora de la utilidad se logra una estrategia de *mejor respuesta* tanto para los compradores como para los vendedores bajo las asunciones de que las distribuciones de carga de todos los participantes son independientes y sin memoria y que la probabilidad de que algún participante compre recursos es baja. □

En el caso de los contratos de precio fijo, los participantes deben negociar el precio final en tiempo de ejecución. En la Sección 5.6 propusimos un protocolo de negociación donde el vendedor tiene la opción de aceptar el menor precio dentro del rango o contraofertar un precio mayor. Presentamos diversas condiciones que determinan cuándo aceptar el precio menor o cuándo contraofertar con un precio mayor para obtener un mayor incremento potencial en la utilidad. A continuación enunciamos la siguiente propiedad:

Propiedad 5.2 Supongamos que la distribución de los costos marginales por unidad de los participantes es independiente y uniforme dentro del rango $[0, 1]$, y

que cada participante tiene C contratos con un rango de precio $[p_L, p_H]$, donde $0 < p_L < p_H < 1$ y $p_H - p_L \ll 1$. Bajo contrato de precio fijo generalmente la solución óptima para un comprador es aceptar el menor precio cuando el costo marginal unitario para la tarea ofrecida se encuentra por debajo del rango de precio y contraofertar con un precio apenas por encima de su valuación cuando su costo marginal está dentro del rango de precio. Cuando el costo marginal excede el rango el vendedor debe rechazar la oferta. ■

Para esta propiedad sólo delinearemos el argumento que la justifica dado que el valor exacto del factor por el cual el vendedor debería incrementar su valuación cuando su costo marginal cae dentro del rango contratado depende de la carga total del sistema, es decir depende de las distribuciones de carga reales.

En primera instancia examinaremos el caso en el que la valuación S (promedio del costo marginal por unidad para la tarea ofrecida) del vendedor se encuentra por debajo del rango de precio y mostraremos que la mejor estrategia es aceptar el menor precio.

En MPA un vendedor no conoce al resto de sus competidores con certeza pero sabe que es sensato esperar que el comprador tenga un conjunto de C contratos. El vendedor maximiza su utilidad aceptando el menor precio en lugar de contraofertar con un precio mayor p_2 cuando el beneficio cierto, $p_L - S$, es mayor que el beneficio esperado, $(p_2 - S)P(p_2)$. Una cota superior para $P(p_2)$ es la probabilidad $(1-p_L)C-1$ de que todos los demás participantes tengan un costo marginal unitario por encima de p_1 y no puedan aceptar dicho menor precio p_1. Incluso con esta aproximación aceptar el menor precio es beneficioso si:

$$\begin{array}{rcl} p_L - S & > & (p_2 - S)(1 - p_L)^{C-1} \\ S & < & \frac{p_L - p_2(1-p_L)^{C-1}}{1-(1-p_L)^{C-1}} \end{array}$$

Dado que p_2 puede a lo sumo alcanzar a p_H:

$$S < p_L \left(\frac{1 - \frac{p_L}{p_H}(1 - p_L)^{C-1}}{1 - (1 - p_L)^{C-1}} \right)$$

Por ejemplo si el rango de precio es $[0{,}75, 0{,}8]$ y $C = 5$ el vendedor debería aceptar el precio mientras $S < 0{,}99997 p_L$. Por lo tanto, cuando el rango de precio es pequeño o cuando existe un número elevado de contratos, el vendedor debe aceptar el menor precio ofrecido mientras su valuación S se encuentre por debajo de p_L.

A continuación examinaremos la estrategia del vendedor cuando $S \in [p_L, p_H]$. En la Sección 5.6.2 vimos que en una subasta de primer precio a sobre cerrado donde compiten C compradores por un bien (recurso) y sus valuaciones se encuentran

uniformemente distribuidas dentro del rango $[0, 1]$, la estrategia de ofertar una fracción $\frac{C-1}{C}$ de la valuación cumple con el Equilibrio de Nash [Vic61b, Tro99]. En una subasta en reversa con un rango de precio, el factor aplicado al descuento ofrecido por el vendedor se traduce en la siguiente contraoferta: $p_2 = p_H - \frac{\beta C-1}{C}(p_H - S)$, donde $\beta C - 1$ es el número de vendedores que compiten contraofertando dentro del rango de precios. Por lo tanto, para determinar la estrategia del vendedor debemos computar el número de vendedores que compiten. Debido a que contraofertar puede resultar en la obtención de la carga sólo si el resto de los vendedores que compiten tienen un costo marginal por encima de p_L, el vendedor debe siempre considerar que éste es el caso al realizar una contraoferta. Esta situación es más factible cuando la carga total del sistema es alta. Si todos los competidores presentan un costo marginal por encima de p_L, sólo aquellos cuyo costo marginal también se encuentre por debajo de p_H serán los que realmente compitan por la carga. Dado que el rango de precios $[p_L, p_H]$ es pequeño, si la carga total ofrecida es alta entonces muchos participantes tendrán una carga superior a p_H. Sin embargo, cada participante posee un conjunto de C contratos que utiliza para redistribuir cualquier carga por encima de p_H incrementando el número de participantes con un costo marginal por unidad por debajo de p_H. Por lo tanto, si el costo marginal del vendedor está dentro del rango de precios entonces debe contraofertar con un precio basado en la asunción de que β es cercano a 1.

En resumen, para el caso del vendedor casi siempre es óptimo aceptar el precio ofrecido cuando su valuación se encuentra por debajo de p_L. Para el caso de una valuación dentro de $[p_L, p_H]$ lo ideal es contraofertar con un precio ligeramente por encima de la valuación que maximiza la utilidad esperada. Cuando la valuación del vendedor se encuentra por encima de p_H el vendedor debe entonces rechazar la oferta.

Creemos que la negociación aquí propuesta es eficiente, *i.e.,* los participantes no emplean ni mucho tiempo ni muchos recursos en la negociación. Usualmente los vendedores poco cargados aceptan directamente el menor precio en una negociación de un único paso. En el raro caso en el que todos los servers estén sobrecargado el comprador directamente acepta la menor contraoferta. Si ocurre el escenario aún más raro en el que todos los vendedores y el comprador tienen costos marginales dentro del rango y dichos costos son similares, los participantes pueden no ser capaces de migrar carga y por lo tanto deberán esperar un período d de tiempo antes de reintentar nuevamente. Si el vendedor pierde una oferta y vuelve la misma oferta es probable que el comprador no pueda aceptarla y tenga que esperar antes de reintentar. Por lo tanto, el vendedor debe contraofertar con un valor menor para mejorar

sus probabilidades de un trato (movimiento de carga) mutuamente beneficioso. Mediante el reajuste de las contraofertas, los participantes pueden rápidamente acordar un precio final que sea beneficioso para ambas partes.

Mostramos entonces que es beneficioso para los participantes formar parte del protocolo MPA y que su mejor estrategia (bajo algunas asunciones de simplificación) es seguir la implementación propuesta. A continuación discutiremos las propiedades algorítmicas de MPA. Para simplificar el análisis asumiremos que el sistema está compuesto por nodos y contratos homogéneos. Dejaremos los sistemas heterogéneos para trabajos futuros.

Antes de examinar el tiempo de convergencia y la sobrecarga en las comunicaciones consideraremos las condiciones necesarias para garantizar la convergencia a asignaciones aceptables, el objetivo principal de MPA.

En MPA una transferencia de carga ocurre solamente si el costo marginal del nodo que ofrece la carga se encuentra estrictamente por encima del costo marginal del nodo que está aceptando la carga. Dado que las funciones de costo son convexas, las asignaciones sucesivas estrictamente decrecen la suma de todos los costos y por lo tanto los movimientos eventualmente terminan bajo una carga constante. Si todos los participantes pudiesen comunicarse unos con otros entonces la asignación final sería siempre aceptable y Pareto optimal, *i.e.,* ningún agente podría mejorar su utilidad sin que otro agente decrezca la propia. Sin embargo, en MPA debido a que los participantes establecen sólo unos pocos contratos e intercambian carga únicamente con sus partners directos esta propiedad no se mantiene. Por el contrario, dada una carga determinada, MPA limita la máxima diferencia en los niveles de carga que pueden existir entre los participantes una vez que el sistema converge a su asignación final. Esta propiedad y el cómputo del rango de precio mínimo originan el siguiente teorema:

Teorema 5.2 Si los nodos, los contratos y las tareas son homogéneas y a su vez los contratos se establecen de acuerdo al Lema 5.1, la asignación final bajo una carga estática constituye una asignación aceptable para sistemas poco cargados y una asignación cuasiaceptable para sistemas sobrecargados. ■

Precondición: Todos los participantes tienen funciones de costo que son monótonamente crecientes y convexas. Además cualquier nodo es capaz de correr cualquier tarea.

Por conveniencia a continuación repetiremos el Lema 5.1:

Lema 5.1 En una red de nodos, tareas y contratos homogéneos para asegurar la convergencia a asignaciones aceptables en un sistema no sobrecargado, el rango de

precios en los contratos debe ser al menos
$[PrecioFijo - \delta_{M-1}(conj\_tareas^F), PrecioFijo]$,
donde M es el diámetro de la red de contratos y $conj\_tareas^F$ es el conjunto de tareas que satisface $MC(u, conj\_tareas^F - u) \leq carga(u) * PrecioFijo$ y
$MC(u, conj\_tareas^F) > carga(u) * PrecioFijo$. ■

Demostración: Examinaremos este teorema por separado bajo un sistema sobrecargado y bajo un sistema poco cargado. Para un sistema sobrecargado mostraremos que si al menos un nodo permanece sobrecargado en la asignación final, entonces todos los nodos operarán justo o por encima de su límite menor de contrato y el sistema se encontrará en una asignación cuasiaceptable. Para un sistema poco cargado veremos que es imposible que un nodo presente un costo marginal por unidad mayor que el rango contratado en la asignación de carga final, lo que hace que la asignación sea aceptable.

A continuación examinaremos las condiciones necesarias para la convergencia bajo una carga estática. En MPA las funciones de costo son monótonamente crecientes y convexas y debido a que la carga se mueve únicamente en la dirección decreciente de los costos marginales la convergencia sigue un gradiente descendiente. La convergencia se detiene solamente cuando se vuelve imposible mover más tareas entre cualquier par de partners. Supongamos que un nodo N_i tiene un nivel de carga $conj\_tareas_i$. N_i no puede migrar más carga a ningún partner si y solo si $conj\_tareas_i$ se encuentra por debajo del rango contratado, la carga de cada partner N_{i+1} está por encima de $conj\_tareas_i - u$, o la carga de cada partner está por encima del rango contratado.

Empleando las condiciones necesarias para que la convergencia se detenga, mostraremos que en la asignación final en un sistema sobrecargado todos los nodos presentan un nivel de carga que se encuentra igual o por encima de $conj\_tareas^F - (M-1)u$, asumiendo el contrato y las condiciones del sistema que pide el Lema 5.1.

Supongamos que el nodo N_0 existe en la asignación final y tiene un costo marginal por unidad mayor que $PrecioFijo$. Por definición de $conj\_tareas^F$, la carga en N_0 es al menos $conj\_tareas^F + u$ (una tarea por encima de su capacidad), debido a que el sistema alcanzó su asignación final no se pueden realizar nuevas migraciones de carga y ninguna de las tareas en exceso pertenecientes a N_0 puede migrarse hacia alguno de sus partners. En este caso para que no puedan ocurrir migraciones debe darse que cada partner N_1 de N_0 tenga un nivel de carga de al menos $conj\_tareas^F$. Si el nivel de carga de cualquier N_1 estuviese al menos una tarea por debajo, por definición de $conj\_tareas^F$, el partner podría aceptar al menos una tarea de N_0 y la asignación no sería final. Similarmente cualquier partner N_2 de N_1 debe tener un

nivel de carga de al menos $conj\_tareas^F - u$. De otra forma todavía sería posible mover carga desde N_1 hacia N_2. Por inducción en la longitud del camino, cualquier nodo N_i, i saltos (*hops*) de distancia de N_0 debe tener un nivel de carga de al menos $conj\_tareas^F - (i-1)u$, mientras la carga en N_{i-1} esté dentro del rango de precios. Como el diámetro de la red de contratos es M, cada nodo del sistema está a lo sumo a M saltos de N_0. El nivel de carga en este nodo N_M es al menos $conj\_tareas^F - (M-1)u$, pues la carga en sus partners directos, un salto más cerca de N_0 es al menos $conj\_tareas^F - (M-2)u$ y esa carga mayor se encuentra todavía dentro del rango de precios. Por lo tanto, todos los nodos del sistema tienen un nivel de carga superior a $conj\_tareas^F - (M-1)u$ y por definición el sistema logró una asignación cuasiaceptable.

Ahora consideraremos el caso de un sistema poco cargado y mostraremos por contradicción que ningún nodo puede tener una carga final por encima de su capacidad. El argumento para los sistemas sobrecargados demuestra que si al menos existe un nodo cuya carga supera su capacidad entonces todos los nodos operan a la misma o con una carga que supera su capacidad en el límite bajo de su rango de precio contratado. Por definición, una distribución de carga de este tipo indica que el sistema está sobrecargado. Por contradicción, un nodo N_0 con un nivel de carga por encima de su capacidad no puede existir en un sistema poco cargado. □

De acuerdo al teorema y lema anteriores es suficiente un pequeño rango de precios para lograr una asignación aceptable bajo un sistema poco cargado y una asignación cuasiaceptable para sistemas sobrecargados. Para una red de diámetro M, el *ancho* del rango de precios es solamente $\delta_{M-1}(conj\_tareas^F)$.

Examinaremos a continuación la complejidad computacional, *i.e.,* el tiempo de convergencia y el overhead sobre las comunicaciones en el mecanismo de precio acotado. En general, podemos decir que los contratos negociados offline, especialmente los de precio fijo hacen que la negociación en tiempo de ejecución sea mucho más simple. Estos contratos traen usualmente aparejados bajos niveles de complejidad de cómputo en la mayoría de las configuraciones y los overheads en las comunicaciones son significativamente menores a aquellos provenientes de subastas. Primero examinaremos la complejidad computacional a partir de la siguiente propiedad:

Propiedad 5.3 Supongamos que cada participante i tiene C contratos a un precio correspondiente a sus capacidades predefinidas T_i y que el exceso total de carga fija es de K tareas. MPA tiene un mejor caso de tiempo de convergencia de $O(1)$ y un peor caso de tiempo de convergencia de $O(KC)$. ■

En esta propiedad, K es la suma de los excesos de carga de todos los participantes y por ello deben reasignarse un total de K tareas.

El costo de migrar carga es proporcional al número de tareas migradas y al tamaño de su estado, sin embargo no consideraremos este último factor en el análisis de complejidad por cuestiones de simplicidad. Un nodo selecciona el conjunto de tareas a ofrecer o a aceptar y computa su costo marginal a lo sumo una vez por cada oferta de carga que produce o recibe. Si bien los cómputos del costo marginal podrían incrementarse con el número de tareas ofrecidas, asumiremos que este incremento es despreciable[19]. En el mejor caso los participantes sobrecargados simultáneamente contactan diferentes nodos poco cargados, quienes aceptan todo el exceso de carga al precio ofrecido. Todas las ofertas y sus respuestas proceden en paralelo para las K tareas, resultando en un tiempo de convergencia de $O(1)$. La Figura 5.14(a) ilustra la configuración del mejor caso. Las subastas son computacionalmente más complejas y durante la convergencia es probable que cada nodo esté participando en múltiples subastas al mismo tiempo, corriendo el riesgo de sobrevender o subutilizar sus recursos. La sobreventa (*overbooking*) no puede ocurrir en MPA cuando las condiciones de carga actuales coinciden con las esperadas.

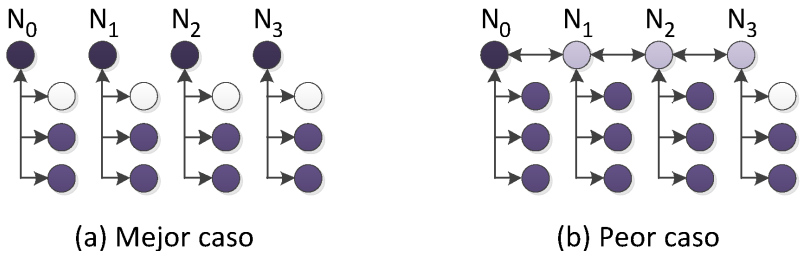


FIGURA 5.14: Ejemplos de mejor y peor caso de distribuciones de carga

En el peor caso, los contratos de precio fijo no permiten obtener una asignación aceptable, sin embargo, los contratos de precio acotado logran siempre una asignación aceptable incluso en el peor caso, pues la convergencia siempre sigue un gradiente descendiente y por lo tanto la carga se podrá propagar por una larga cadena de nodos. La Figura 5.14(b) ilustra esta configuración de peor caso. En este ejemplo la totalidad del exceso de carga se ubica en un único nodo; del resto, uno presenta una capacidad libre significativa y los demás se encuentran justo en su nivel de capacidad. Si los nodos contraofertan, entonces cada movimiento de carga reasigna exactamente una tarea y en cada iteración la mitad de los nodos de la cadena ofrecen una tarea y la otra mitad acepta una tarea. Debido a que deben migrarse un total de K tareas, el tiempo de convergencia es entonces $2K$ independientemente

[19]Dependiendo de las aplicaciones este incremento no siempre es despreciable.

de la longitud de la cadena. Adicionalmente, dado que cada nodo tiene C contratos debe contactar sucesivamente a los C partners cada vez que negocia un precio final dentro del rango. De esta discusión se desprende que el tiempo de convergencia en el peor caso es entonces de $O(KC)$.

Con las subastas, el tiempo de convergencia en el peor caso es algo menor debido a que los nodos se comunican con los C partners en paralelo y potencialmente los nodos pueden además mover más tareas en cada iteración. Sin embargo, la longitud de la cadena acota el número de tareas que se migran juntas. El tamaño de estos grupos también se decrementa a medida que el sistema se acerca a una distribución de carga pareja.

Por lo tanto concluimos que el mecanismo de precio acotado presenta una menor complejidad que para el caso de subastas en el mejor caso y no necesariamente se comporta peor en el peor caso. Sin embargo, tanto el mejor como el peor caso constituyen ejemplos extremos y bajo el mecanismo de contratos con precio acotado generalmente ni la carga suele ser absorbida por vecinos directos ni debe propagarse por una única cadena. Por el contrario, la carga se propaga por caminos paralelos creados por la red de contratos y por ello puede esperarse que los tiempos de convergencia decrezcan rápidamente con cada contrato adicional que establezca el nodo. En trabajos futuros verificaremos empíricamente esta hipótesis.

A continuación analizaremos el overhead de nuestro mecanismo con respecto a las comunicaciones y lo compararemos con el incurrido por subastas. Para ello discutiremos la siguiente propiedad:

Propiedad 5.4 Supongamos que cada participante i tiene C contratos a un precio correspondiente a sus capacidades predefinidas T_i y que el exceso total de carga fija es de K tareas. MPA tiene un mejor caso de overhead en las comunicaciones de $O(1)$ y un peor caso de overhead en las comunicaciones de $O(MKC)$, donde M es el diámetro de la red de contratos. ■

Con nuestro mecanismo la mayoría de las migraciones de carga requieren solamente tres mensajes: una oferta, una respuesta y un movimiento de la carga. Los mensajes de migración de carga imponen el mayor overhead producido durante la migración de una tarea (migración que incluye su correspondiente estado), pero dado que este análisis sólo puede efectuarse mediante experimentación sobre una variedad de aplicaciones ignoraremos este overhead en este punto. Los mensajes de oferta y respuesta son pequeños: sólo contienen listas de tareas, estadísticas acerca de la utilización de recursos y precios. Asumiremos que el overhead por mensaje

es constante. Por lo tanto el overhead de comunicación es en el mejor escenario de $O(1)$.

En contraste, bajo un esquema de subastas el mejor caso para el overhead en la comunicación por movimiento es de $O(C)$. Bajo subastas dado que el sistema converge a distribuciones de carga uniforme el overhead total puede superar $O(C)$. Si los participantes deben mover K tareas por una cadena de M nodos el overhead en el peor caso en MPA puede llegar a $O(MKC)$, pero puede decrecer hasta $O(M(K+C))$ si los nodos sobrecargados le indican a sus partners que dejen de ofertar luego de que falló la primer oferta. Con las subastas los nodos deben siempre enviar ofertas a sus C partners pero pueden potencialmente migrar más tareas por vez por una cadena.

Entonces, si comparamos MPA con las subastas podemos decir que en el mejor caso el primero reduce significativamente tanto el overhead en las comunicaciones como la complejidad computacional. En el peor caso ambas técnicas pueden alcanzar elevados niveles de complejidad y overhead, pudiendo incluso MPA ser potencialmente peor. Sin embargo, una de las ventajas más importantes de MPA es el menor número total de migraciones de carga. La carga sólo se mueve cuando la carga de un participante excede el precio de un contrato mientras existen partners que tienen capacidad libre, o cuando la carga de un participante cae dentro del precio de un contrato mientras algunos de sus participantes están sobrecargados.

5.9 Resumen

En este capítulo presentamos MPA y sus propiedades. La idea básica detrás de MPA es que los participantes establezcan contratos bilaterales offline y que dichos contratos especifiquen un pequeño rango de precios. En tiempo de ejecución los participantes deben pagar a sus pares el precio pactado por procesar su carga. Los contratos también pueden especificar la duración de una migración de carga, la cual determina la cantidad mínima de tiempo que debe transcurrir para que el participante pueda cancelar un movimiento de carga previo. Mostramos que la mejor estrategia que debe seguir un participante es la que aquí se propone: ofrecer y aceptar carga cuando hacerlo mejora su utilidad, y que un pequeño rango de precios asegura que un sistema homogéneo siempre converja hacia una asignación aceptable o al menos a una cuasiaceptable.

Capítulo 6

Evaluación

Índice

6.1. Implementación del simulador 175
6.1.1. Topologías simuladas 176
6.2. Convergencia hacia asignaciones aceptables 178
6.3. Velocidad de convergencia 181
6.4. Estabilidad bajo variaciones de carga 183
6.5. Implementación 185
6.5.1. Experimentos con el prototipo 185
6.5.2. Limitaciones y extensiones 188
6.6. Resumen 189

A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it.

— *Albert Einstein*

En este trabajo fueron presentados formalmente propiedades y resultados teóricos para demostrar la validez de la propuesta. Sin embargo la naturaleza del tema requiere evaluación experimental para completar el análisis.

Mediante la simulación de redes aleatorias de contratos y asignaciones de carga aleatorias podemos verificar que MPA funciona bien en entornos heterogéneos. En esta evaluación trabajaremos con cargas en los nodo provenientes de la misma distribución y nodos con diferentes capacidades para establecer contratos a diferentes precios.

Este estudio nos permite comprobar que es suficiente contar con un pequeño número de contratos para asegurar que la asignación final sea cuasiaceptable tanto para nodos con carga liviana como para sistemas heterogéneos con nodos sobrecargados. Es interesante notar que aún en el caso en que no se obtenga una asignación aceptable comprobamos que los contratos de precio fijo suelen converger hacia buenas distribuciones de carga. Más aún, un pequeño número de contratos de precio fijo por nodo es suficiente para que los mismos puedan reasignar la mayor parte de su exceso de carga, y de esta forma utilizar la mayor parte de la capacidad disponible en el sistema. Adicionalmente, los contratos de precio fijo convergen en forma rápida hacia una distribución de carga final. En el caso del empleo de contratos de precio acotado (rango de precios) la convergencia puede tomar un tiempo mayor, no obstante en las configuraciones simuladas el 95 % de los beneficios ocurren aproximadamente dentro del 15 % inicial del tiempo de convergencia. Además mostraremos que, en ambientes con carga dinámica, MPA se comporta en forma adecuada ante variaciones de carga que podrían ocurrir en una aplicación de monitoreo de una red real.

6.1 Implementación del simulador

Para la simulación empleamos SimPy `http://simpy.sourceforge.net`, un simulador opensource de eventos discretos basado en Python. SimPy nos permite modelar cada nodo como un proceso autónomo, encargándose adicionalmente del scheduling de estos procesos simulados.

Cada nodo implementa los algoritmos propuestos para manejar el exceso de carga (Algoritmo 5.1) y para tomar carga adicional (Algoritmo 5.2). Los nodos son forzados a dormir durante un período de tiempo $D = 1$ segundos simulados luego de cada movimiento de carga exitoso o en el caso en el que no haya movimiento de carga posible. En la práctica es necesario insertar un tiempo de demora entre movimientos de carga para lograr estabilizar la carga antes de computar las nuevas condiciones de

carga. Dado que migrar carga implica transferir estado (un estado inherentemente complejo para modelar), asumiremos en nuestro simulador que el movimiento de carga resultante es inmediato. Esta asunción no debería afectar nuestros resultados, pues en la práctica la demora para intercambiar mensajes y mover la carga es significativamente menor que el período de tiempo D.

Para los contratos con rango de precio es necesario simular las negociaciones de precio siguiendo el algoritmo propuesto. Por lo tanto, cuando un nodo recibe una oferta de carga puede aceptar la totalidad de la misma o un subconjunto de ella, rechazar la oferta, o responder con una contra-oferta. A menos que se indique lo contrario, la contra-oferta es igual a la valuación para una tarea. En la Sección 6.3 discutiremos contra-ofertas más altas. Dado que las contra-ofertas son vinculantes, cuando un nodo envía una contra-oferta, temporalmente reserva recursos para la carga esperada. Cuando se recibe una contra-oferta, el comprador envía la oferta original a otro partner. Si todos los partners envían una contra-oferta, entonces el comprador migra carga hacia el nodo con la mejor (menor valor) contra-oferta y se comunica con los restantes partners para "liberarlos" de sus correspondientes contra-ofertas. Para la implementación de este protocolo necesitamos que los compradores esperen durante un período de tiempo corto, por ejemplo $d = 0{,}025D$ luego de cada contra-oferta, lo cual hace que para 10 contratos el tiempo total de negociación sea de a lo sumo (peor caso) un cuarto de segundo. Estos pequeños retardos d nos permiten simular mejor el *interleaving* de las negociaciones de precio, donde múltiples nodos tratan de migrar carga a los mismos partners al mismo tiempo. Si no existen migraciones de carga posibles, el nodo debe esperar un tiempo D antes de reintentar ofrecer carga al mismo precio mínimo.

6.1.1 Topologías simuladas

Se simuló un sistema de 1000 participantes conectados mediante contratos bilaterales y se varió el número mínimo de contratos por nodo, conformando de esta forma varias topologías de contratos al azar. Para crear una topología primero cada nodo establece un contrato bilateral con un nodo con el cual presenta una conexión previa. Los nodos que cuentan con muy pocos contratos seleccionan partners al azar. Mediante este algoritmo se mantiene una baja diferencia entre el mínimo y el máximo número de contratos. Por ejemplo, ningún nodo tiene más de 18 contratos cuando el número mínimo de los mismos es de 10 contratos por nodo.

Se simularon tanto sistemas uniformes como heterogéneos, donde en los primeros todos los nodos presentan la misma capacidad. Se configuró está capacidad en 100

tareas (o 100 %). Para lograr heterogeneidad distribuimos las capacidades de los nodos uniformemente en el rango [80, 120]. Si bien esta elección es arbitraria permite que las diferencias entre las capacidades de los nodos alcance el 50 %. La granularidad de movimiento de carga es de una tarea (o 1 % de la capacidad total). En el simulador abstraemos las funciones de costo y los costos marginales expresando los precios de los contratos directamente en función de los niveles de carga. Si el precio de un contrato es igual a 100, entonces ocurre un movimiento de carga tan pronto como la carga del comprador está justo o por encima de 101 y la carga del vendedor está justo o por debajo de 99.

Se estudiaron y compararon las propiedades de convergencia de las cuatro variantes de nuestro mecanismo, obteniéndose los siguientes resultados:

Variante uniforme fija. Todos los nodos tienen la misma capacidad y los mismos contratos de precio fijo. Se asignan precios de contratos iguales a las capacidades de los nodos.

Variante uniforme con rango. Todos los nodos tienen la misma capacidad nuevamente, pero se extienden los contratos de precio fijo para cubrir un rango de 5 tareas, *i.e.,* los contratos cubren el rango [95, 100]. Utilizamos rangos de 5 tareas porque corresponden al diámetro de la segunda topología más pequeña que simulamos.

Variante heterogénea fija. Las capacidades de los nodos y los precios de los contratos varían dentro del rango [80, 100]. Cuando dos nodos con diferentes capacidades establecen un contrato bilateral, emplean un precio igual a la menor de sus capacidades. Por ejemplo si un nodo con capacidad 83 establece un contrato con un nodo cuya capacidad es 110, entonces el precio del contrato es 83.

Variante heterogénea con rango. Las capacidades de los nodos y los precios de los contratos varían en el rango [80, 100]. Cuando dos nodos con diferentes capacidades establecen un contrato bilateral, para determinar el límite mayor del rango de precio utilizan la menor de sus capacidades. El ancho del rango es siempre de 5 tareas. Por ejemplo si nodos con capacidades de 83 y 110 establecen un contrato, el mismo cubrirá el rango [78, 83].

6.2 Convergencia hacia asignaciones aceptables

Se estudiaron las propiedades de asignación de carga para sistemas heterogéneos y se compararon los resultados con los obtenidos en un sistema formado por nodos y contratos uniformes. Encontramos que incluso un pequeño número de contratos de precio fijo permiten que un sistema potencialmente heterogéneo converja hacia asignaciones cuasiaceptables para todos los niveles de carga simulados. Los rangos pequeños de precios colaboran para que el sistema alcance buenas distribuciones de carga bajo un entorno de pocos contratos por nodo.

Simulamos cada una de las cuatro variantes de MPA para cada una de las diez topologías bajo cuatro condiciones de carga diferentes: (1) carga baja, donde las cargas corresponden al 50 % de la capacidad del sistema, (2) carga alta o pesada, donde las cargas corresponden al 75 % de la capacidad del sistema, (3) baja sobrecarga, donde las cargas corresponden al 125 % de la capacidad del sistema, y por último (4) alta sobrecarga, donde las cargas corresponden al 150 % de la capacidad del sistema. Para crear cada una de las condiciones de carga, establecemos la carga de cada nodo a partir de la misma familia de distribuciones que empleamos al principio de este capítulo.

Una vez distribuidas las asignaciones de carga, comienzan los movimientos de carga entre los nodos. Una vez que las migraciones culminan examinamos que tan lejos se encuentra la asignación final de una asignación aceptable. La Figura 6.1 muestra los resultados. Para las dos configuraciones con baja carga, la figura muestra que la fracción de todas las tareas permanece por encima de la capacidad de algunos nodos. Estas tareas todavía deberían reasignarse, sin embargo el sistema fue incapaz de efectuar estas migraciones. Bajo las configuraciones de sobrecarga, la figura muestra que una fracción del total de capacidad todavía permanece sin utilizar. Esta capacidad debería utilizarse, pero una vez más el sistema mostró ser incapaz de lograrlo. En todos los gráficos, la columna con 0 número mínimo de contratos muestra las condiciones iniciales antes de que ocurran los movimientos de carga. Por ejemplo, para la configuración de baja carga, la asignación inicial es tal que el 30 % del total de tareas debe moverse para que el sistema converja hacia una asignación aceptable. En el escenario de baja sobrecarga, alrededor del 26 % del total de la capacidad disponible no es utilizada inicialmente.

El resultado de estas simulaciones muestra que cuando aumenta el número de contratos mejora la calidad de la asignación final bajo todas las variantes y todas las configuraciones de carga: los nodos reasignan una cantidad mayor de exceso de carga o explotan una mayor porción de la capacidad disponible. La mejora es

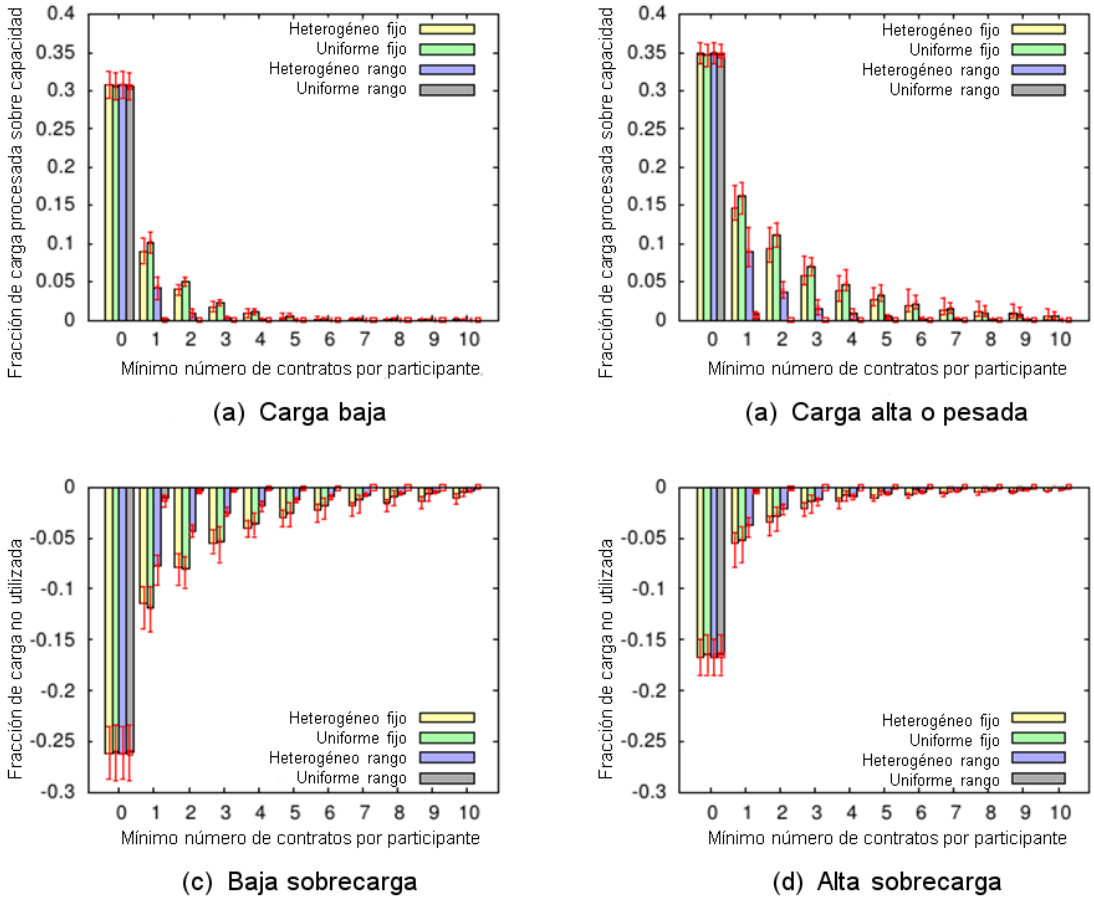


Figura 6.1: Convergencia hacia asignaciones aceptables para diferentes cargas y número de contratos

especialmente significativa para los primeros dos a cuatro contratos establecidos por los nodos. Con aproximadamente 10 contratos por nodo, el sistema converge hacia asignaciones cuasiaceptables para todas las configuraciones simuladas, *i.e.,* todos los tipos de contratos y todos los niveles de carga.

Cuando consideramos los contratos de precio acotado uniforme, el sistema alcanza una asignación aceptable contando con sólo dos contratos por nodo para ambas configuraciones de baja carga (Figura 6.1(a) y (b)). Este resultado es muy significativo, pues nuestro análisis teórico presentado en el capítulo previo predecía que son necesarios a lo sumo 6 contratos por nodo. Con 6 contratos el diámetro de la topología es igual al rango de precio, el cual establece una condición necesaria para asegurar una asignación aceptable. Sin embargo, empleando topologías aleatorias, con un rango de precios igual a sólo la mitad del diámetro, el sistema llega a una asignación aceptable para las diez topologías simuladas.

Para los sistemas sobrecargados (Figura 6.1(c) y (d)), con rangos uniformes de

precios de contratos, MPA sólo puede asegurar asignaciones cuasiaceptables, *i.e.,* algunos nodos puede operar ligeramente por debajo de sus capacidades. Los resultados muestran que con sólo dos contratos por nodo, menos del 0,5 % de la capacidad total permanece sin utilizar. Por lo tanto, con sólo dos contratos por nodo nuestro sistema es capaz de explotar el 99,5 % de la capacidad total del sistema. Es por ello interesante destacar que con unos pocos contratos adicionales el sistema alcanza una asignaciones aceptable. Para sistemas sobrecargados fueron suficiente 8 contratos para asegurar asignaciones aceptables bajo todas las topologías simuladas. Para sobrecargas pequeñas, se necesita un mínimo de 10 contratos en algunas configuraciones para comenzar a obtener una asignación aceptable.

Es necesario también destacar que empleando contratos de precio fijo también se obtienen buenas distribuciones de carga. En un sistema con baja carga, con sólo dos contratos por nodo el sistema reasigna la mayor parte (95 %) de la carga total. Son suficientes sólo 5 contratos por nodo para mantener este valor por encima del 99 % para todas las configuraciones simuladas. Este resultado muestra que los contratos de precio fijo presentan un nivel de eficiencia alto para manejar picos de carga en sistemas que en general presentan baja carga. Incluso cuando la carga se vuelve alta, los contratos de precio fijo presentan un buen comportamiento, pues bajo una carga total de 75 % y sólo 8 contratos por nodo, todas las configuraciones logran reasignar prácticamente todo el exceso de carga (menos el 2 %) y para sistemas sobrecargados con tan sólo 4 contratos por nodo el sistema siempre es capaz de explotar más del 95 % de la capacidad disponible. Los resultados son aún mejores para sistemas fuertemente sobrecargados y con 5 contratos por nodo, en los cuales el sistema puede explotar más del 99 % de su capacidad total. Es por ello que podemos concluir que los contratos de precio fijo son también efectivos bajo escenarios sobrecargados.

Nuestro protocolo funciona adecuadamente también en sistemas heterogéneos, los cuales son más frecuentes de hallar en la práctica. Como muestra la Figura X, aunque un sistema heterogéneo no siempre obtenga asignaciones aceptables o requiera un mayor número de contratos para lograrlo, unos pocos contratos son suficientes para arribar a asignaciones cuasiaceptables.

Más específicamente, los contratos de precio fijo en sistemas heterogéneos llevan a distribuciones de carga ligeramente mejores que las obtenidas en sistemas uniformes con contratos de precio fijo en sistemas con baja carga y bajo número de contratos. Contando con hasta con 7 contratos por nodo el sistema heterogéneo simulado reasigna hasta 2 % más tareas que en la variante uniforme para ambas configuraciones de baja carga. De hecho, los contratos heterogéneos hacen que sea posible que la carga se mueva más lejos de su punto de origen siguiendo una cadena

de contratos a un precio descendiente, permitiendo de esta forma que el sistema obtenga mejores distribuciones de carga. Sin embargo, la heterogeneidad da origen a ineficiencia, pues los movimientos de carga entre partners se limitan a la menor de sus dos capacidades. Esta ineficiencia se vuelve aparente al aumentar el números de contratos, pues los precios uniformes comienzan a obtener mejores resultados que los precios heterogéneos. La ineficiencia debido a la heterogeneidad hace que sea difícil para nuestro protocolo explotar toda la capacidad disponible en un sistema sobrecargado. Los contratos de precio fijo logran un mejor resultado que los heterogéneos empleando un número marcadamente menor de contratos en un sistema sobrecargado. Sin embargo, si consideramos todos los escenarios, la performance de las configuraciones uniformes y heterogéneas es comparable.

La performance de un sistema heterogéneo con rango de contratos está regida por factores similares: con sólo dos contratos por nodo el sistema reasigna todo menos el 5 % de la carga bajo condiciones de baja carga y utiliza más del 95 % de las capacidades disponibles bajo una configuración de sobrecarga. Sin embargo, se requieren al menos 6 contratos por nodo para que el sistema pueda garantizar la convergencia hacia una asignación estable en un ambiente de poca carga. El sistema logra sólo asignaciones cuasiaceptables en escenarios con altos niveles de carga o sobrecarga.

6.3 Velocidad de convergencia

Se estudió el tiempo que tarda el sistema para converger bajo sistemas uniformes y heterogéneos frente a diferentes condiciones de carga. Como era de esperarse bajo sistemas uniformes con contratos de precio fijo todos los movimientos de carga ocurren casi simultáneamente tan pronto como comienza la simulación, obteniendo así una convergencia extremadamente rápida. La convergencia se vuelve más lenta con los contratos de precio acotado debido a que toma tiempo que la carga se propague a través de cadenas de nodos. Sin embargo, la mayoría de las tareas se reasigna rápidamente, *i.e.,* tan pronto como comienza la simulación. Los movimientos de carga subsiguientes proveen en forma decreciente menores beneficios. En los sistemas uniformes la velocidad de convergencia también aumenta rápidamente junto con el número de contratos por nodo, pues la carga puede propagarse en paralelo en múltiples direcciones. Con los contratos de precio acotado encontramos que si cada nodo posee más de dos contratos, las contra-ofertas por encima de las valuaciones no impactan negativamente sobre la velocidad de convergencia.

La Figura ? muestra ejemplos concretos de convergencia. Estos ejemplos corresponden a una configuración con un nivel de carga del 75 % y un mínimo de tres contratos por nodo. Elegimos una configuración con pocos contratos para ilustrar los efectos de mover carga a través de una cadena de nodos. Solamente los contratos de precio acotado bajo sistemas uniformes obtienen asignaciones aceptables. Los resultados muestran el número total de tareas migradas, el número total de movimientos de carga, y el número total de tareas remanentes que deben reasignarse. Los valores exactos no son importantes, pues dependen de la configuración del sistema; por ello nos enfocaremos en las tendencias de los resultados. Las simulaciones se detienen luego de 10 segundos contados a partir del último movimiento de carga.

Para todas las variantes en las configuraciones, la mayoría de los movimientos de carga ocurre durante los primeros segundos de la simulación y cada uno de estos movimientos tempranos reasigna un gran número de tareas: *i.e.,* el número de tareas migradas es significativamente mayor que el número total de migraciones de carga, indicando que cada movimiento involucra múltiples tareas. Las migraciones de carga tempranas también proveen los mayores beneficios en la convergencia global, como se puede observar en cómo decrece el total de tareas que todavía necesita reasignación. Esta convergencia rápida puede explicarse parcialmente por la habilidad de MPA para balancear la carga entre cualquier par de nodos mediante una sola interacción y en parte por su habilidad para balancear carga simultáneamente en múltiples ubicaciones de la red.

Bajo escenarios uniformes y de contratos de precio fijo, la convergencia se detiene rápidamente, pues la carga sólo se puede mover un *hop* desde su origen. Todos los movimientos de carga ocurren entonces durante los primeros pocos segundos. Los sistemas heterogéneos de precio fijo permiten algunos movimientos adicionales debido a la propagación de carga por cadenas de contratos con precios decrecientes. Sin embargo, las migraciones de carga nuevamente se detienen rápido.

En la configuración uniforme bajo contratos de precio acotado, la convergencia ocurre en dos etapas. La primera toma lugar durante los primeros pocos segundos, donde los nodos sobrecargados mueven grandes grupos de tareas hacia sus partners directos (*i.e.,* ocurren muchos movimientos, cada uno involucrando a muchas tareas). El número de tareas que necesita reasignación decrece abruptamente. En la segunda etapa, los nodos lentamente empujan el exceso de carga remanente siguiendo cadenas de nodos. Los gráficos muestran una larga cola de migraciones de carga que provee pequeñas mejoras incrementales al número total de tareas a reasignar. El número de tareas migradas en cada unidad de tiempo es igual al número de migraciones de carga, indicando que cada movimiento reasigna una única tarea. Este

comportamiento es consistente con la propagación de carga a través de cadenas de nodos.

La heterogeneidad crea cadenas de contratos con precios decrecientes y por lo tanto reduce el número de tareas que se migra tempranamente debido a que los precios contratados son menores que los del entorno uniforme. Sin embargo, los movimientos de carga subsiguientes continúan con la migración de grupos de tareas en lugar de tareas individuales. Este fenómeno ocurre debido a que las diferencias en los precios contratados son mayores en los ambientes heterogéneos. En el caso uniforme los nodos pueden modificar precios sólo dentro del pequeño rango contratado.

Nuestras simulaciones muestran que cuando se negocia el precio final dentro del rango contratado, los nodos estuvieron siempre contra-ofertando con su valuación verdadera para cada tarea adicional. A continuación se examinan los efectos de las contra-ofertas por encima de las valuaciones. Para ello modificamos el simulador de la siguiente forma: cuando un nodo recibe una oferta mientras su nivel de carga se encuentra dentro del rango contratado, éste contra-oferta con un precio igual a su valuación más un número x de tareas. Si una contra-oferta no resulta en una migración de carga, entonces el nodo contra-oferta en su segundo intento con su valuación exacta. Si bien las contra-ofertas por encima de las valuaciones pueden entonces afectar el tiempo de convergencia, presentan las mismas propiedades en términos de distribución de carga final. A excepción de contra-ofertas iguales al máximo precio dentro del rango, todas las contra-ofertas se aplican a una única tarea. Dado que las contra-ofertas son siempre vinculantes, frente a la ocurrencia de la misma, el nodo reserva recursos para las tareas potenciales. Es de destacar que en nuestras simulaciones, a excepción de configuraciones con uno o dos contratos por nodo, contra-ofertar con un precio por encima de la valuación ni daña ni ayuda en la velocidad de convergencia. De hecho, incluso si las contra-ofertas son altas, los nodos siguen migrando generalmente una única tarea a la vez. Una contra-oferta demasiado alta provoca que una migración falle solamente cuando la valuación del comprador se encuentra a una o dos tareas de la del vendedor, la cual ocurre poco frecuentemente si existen múltiples vendedores.

6.4 Estabilidad bajo variaciones de carga

A continuación examinaremos de qué manera MPA gestiona condiciones de carga cambiantes. Para simular carga variable, agregamos dos procesos para cada nodo simulado: uno para aumentar periódicamente la carga y otro simétrico encargado de decrementarla. Los intervalos de tiempo entre el agregado y la remoción de carga

siguen distribuciones exponenciales. Para simular variaciones de carga crecientes decrementamos la media de las distribuciones. Debido a que la versión actual de nuestro simulador puede manejar sólo 1000 procesos, decrementamos la cantidad de nodos simulados a 330 y empleamos una topología donde cada nodo tiene al menos 10 contratos debido a que este alto número de contratos permite mayores números de migración de carga. La topología simulada presenta un diámetro de cuatro.

Dejamos que la simulación converja hacia una asignación aceptable, lo cual ocurre rápidamente al contar con 10 contratos por nodo. En el tiempo $t = 50$ segundos, comenzamos a simular pequeñas variaciones de carga utilizando una media de 50 segundos: cada nodo recibe una nueva tarea cada 50 segundos en promedio. Debido a que hay 330 nodos en el sistema, hay un promedio de 6,6 nuevas tareas por segundo en el sistema. Empleamos la misma distribución para remover carga y por lo tanto en promedio 6,5 tareas por segundo abandonan el sistema. La carga total en el sistema permanece entonces aproximadamente constante. En el tiempo $t = 300$ aumentamos la variación de carga reduciendo la media a 10 segundos. Finalmente, en el tiempo $t = 600$ reducimos aún más la media, llevándola a 1 segundo.

Sería una propiedad muy mala de nuestro protocolo si para variaciones de carga pequeñas se produjesen excesivas reasignaciones. Sin embargo, nuestras simulaciones muestran que el sistema gestiona en forma adecuada las variaciones de carga, pues para todas las condiciones de carga el sistema absorbe la mayor parte de las variaciones sin producir reasignaciones. Si bien esperaríamos que empleando contratos de precio fijo ocurriesen menos movimientos de carga, es interesante notar que tanto bajo contratos de precio fijo como para contratos de precio acotado, obtenemos un número similar de reasignaciones de carga. Dado que el rango de precios es pequeño, la probabilidad de que un nodo caiga exactamente dentro del rango es también pequeña, provocando de esta forma el resultado antes enunciado.

Las migraciones de carga ocurren solamente durante un período de tiempo D. Una vez transcurrido dicho período, cualquiera de los dos partners puede cancelar la migración si ya no le es redituable. Migrar carga durante un período limitado de tiempo conlleva a mayor inestabilidad. Corrimos simulaciones para medir el efecto de migrar carga durante un período de tiempo acotado y como era de esperarse, el tiempo mínimo de duración de dicho período origina un aumento en el número de reasignaciones. De todas formas, MPA logró absorber la mayor parte de las variaciones de carga sin provocar reasignaciones.

En esta sección exploramos mediante simulaciones algunas de las propiedades de MPA. Mostramos que bajo topologías aleatoriamente generadas, sólo se necesitan

unos pocos contratos de precio acotado por nodo para que el sistema puede asegurar la convergencia hacia asignaciones aceptables, independientemente del nivel de carga global. Además mostramos que tanto los contratos de precio fijo como los contratos heterogéneos producen asignaciones cuasiaceptables, empleando para ello sólo unos pocos contratos por nodo. Adicionalmente vimos que la convergencia hacia distribuciones de carga finales ocurre rápidamente, especialmente para los contratos de precio fija y que en todos los casos la mayor parte de los beneficios de la convergencia ocurre durante los primeros segundos. Por último mostramos que MPA maneja adecuadamente las condiciones de carga variantes, enmascarando la mayor parte de las variaciones de carga sin producir movimientos de carga.

6.5 Implementación

El Gestor de Carga necesita tres tipos de información definida offline: un conjunto de contratos con otros nodo, una función de costo para computar los costos marginales para el procesamiento de las tareas localmente, y una partición en fragmentos del diagrama de consulta, lo cual define las unidades de los movimientos de carga.

Cada nodo utiliza la misma función de costo total:

$$\frac{\rho_{cpu}}{1-\rho_{cpu}} + \frac{\rho_{ab}}{1-\rho_{ab}} \tag{6.1}$$

Donde ρ_{cpu} es la utilización total de CPU y ρ_{ab} es la utilización actual de ancho de banda. Esta función de costo es el número total de tuplas que están siendo procesadas o que están esperando procesamiento, empleando el modelo de cola M/M/1. Elegimos esta función debido a su simplicidad para el cómputo de los niveles de carga y para los costos marginales, utilizando únicamente información de granularidad gruesa para la utilización de CPU y ancho de banda.

Nuestra implementación presenta varias limitaciones: (1) se utilizan estadísticas crudas para el cómputo de las condiciones de carga locales, (2) una vez que se migra carga, ésta migra para siempre, (3) los Gestores de Carga no registran dónde está la carga ejecutándose y (4) se ignora el costo de la migración de carga (ignorando la transferencia de estado para lograr movimientos más veloces).

6.5.1 Experimentos con el prototipo

En esta sección demostramos el funcionamiento de nuestro protocolo en la práctica, corriendo una consulta de monitoreo de red sobe datos reales. La consulta es la misma que se mostró en la Figura 1.3 pero sin el operador *Join* final (Figura 6.2).

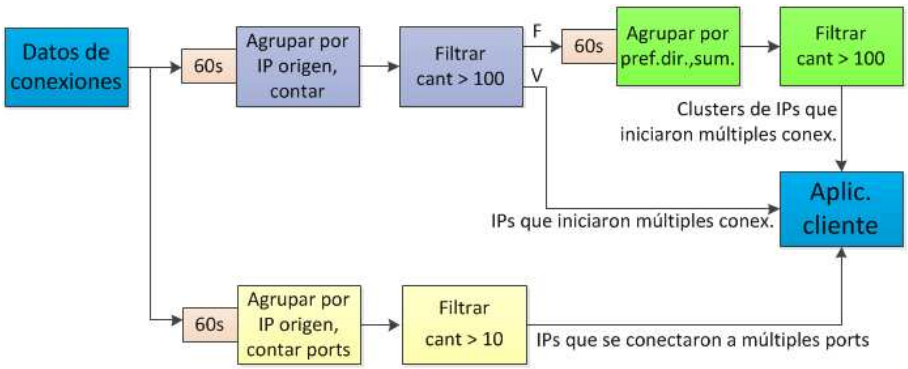


FIGURA 6.2: Digrama de consulta de monitoreo de red

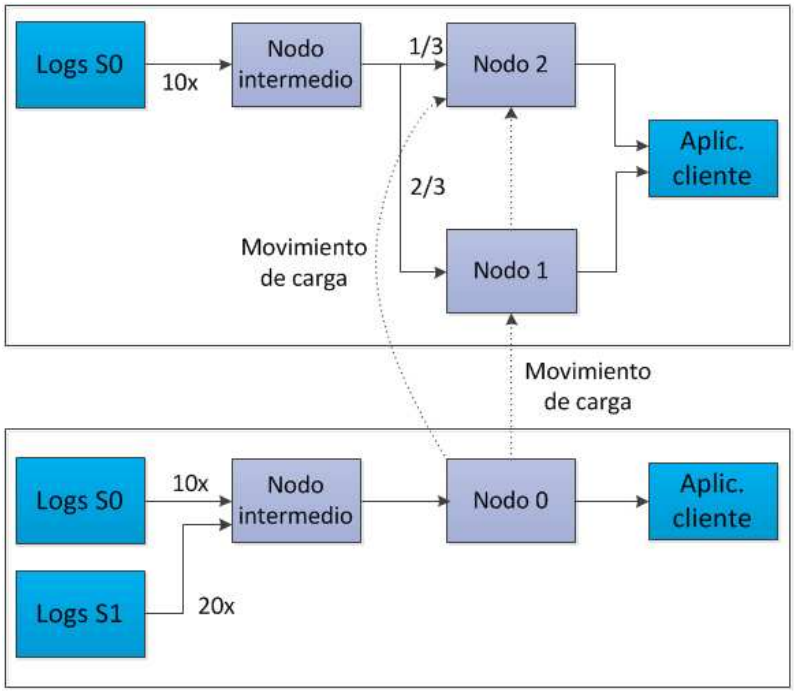


FIGURA 6.3: Ambiente experimental para monitoreo de red

Para reducir la posible granularidad de los movimientos de carga partimos los logs de una de las trazas (S_0) en cuatro y la otra (S_1) en tres para que puedan procesarse en paralelo. La Figura 6.3 muestra la configuración de nuestro *testbed*. El Nodo 0 inicialmente procesa todas las particiones de ambas trazas reales (S_0 y S_1). Los Nodos 1 y 2 procesan 2/3 y 1/3 de las trazas de S_0 respectivamente. Todos los nodos Federación poseen contratos de precio fijo con todos los nodos restantes y están configurados para tomar u ofrecer carga cada 10 segundos.

La Figura 6.4 muestra los resultados obtenidos. Inicialmente la carga en cada nodo es aproximadamente constante. Al llegar aproximadamente a 350 segundos, la carga de la traza S_1 empieza a incrementarse y provoca que el Nodo 0 envíe carga al Nodo 1 dos veces. Luego del segundo movimiento, la carga aumenta ligeramente

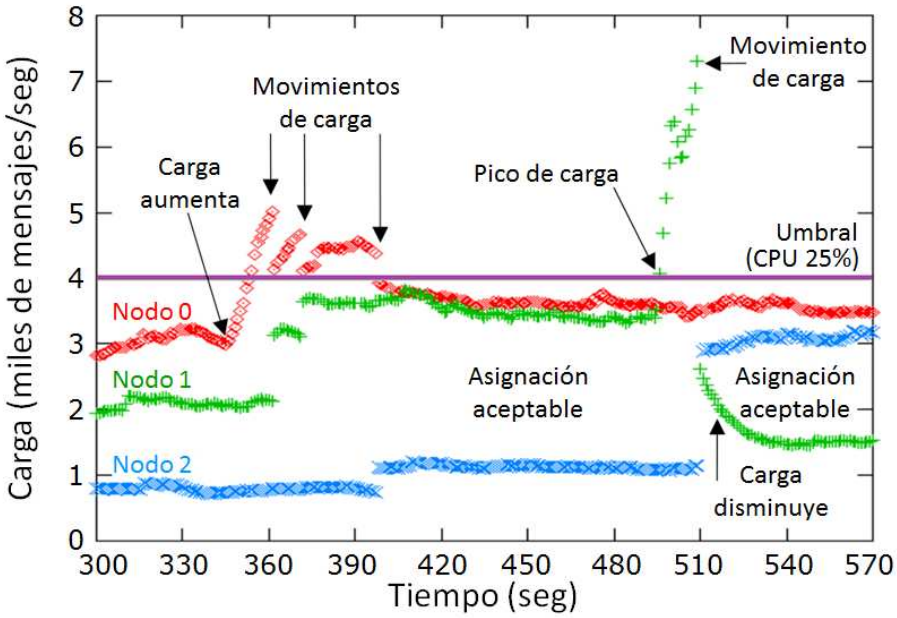


Figura 6.4: Carga en tres nodos Federación corriendo consulta de monitoreo de red

pero el Nodo 1 rechaza carga adicional provocando que el Nodo 0 mueva algunos operadores hacia el Nodo 2. La asignación de carga resultante no es uniforme pero si es aceptable. Cerca de los 500 segundos, el Nodo 1 experimente un pico de carga, causado por un aumento en la carga de la traza S_0. El pico es suficientemente largo como para provocar un movimiento de carga desde el Nodo 1 hacia el nodo 2, logrando que todos los nodos vuelvan a operar dentro de sus capacidades nuevamente. Es interesante notar que luego de la migración, la carga en el Nodo 1 decrece pero no provoca la ocurrencia de nuevas reasignaciones debido a que la asignación permanece aceptable.

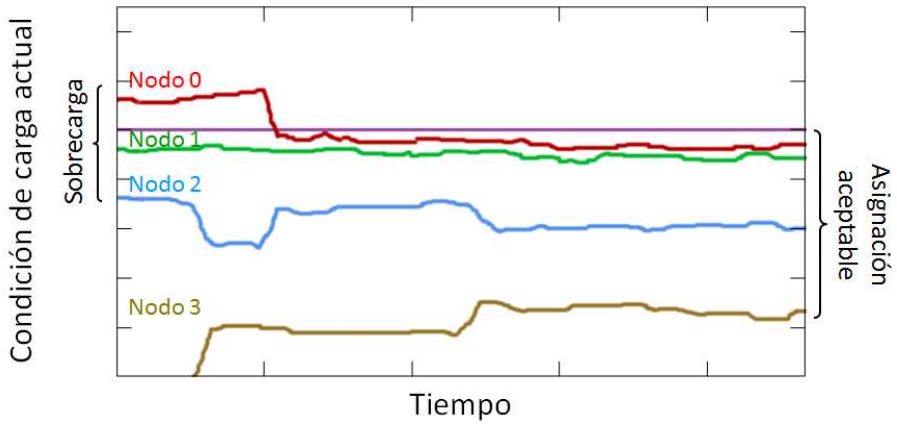


Figura 6.5: Movimientos de carga para la aplicación de monitoreo de red

La Figura 6.5 muestra las condiciones de carga y los movimientos de carga en

cada instante de tiempo. Las condiciones iniciales de este experimento fueron las mismas que para el caso previo: tres nodos con idénticos contratos con el resto. La Figura 6.5 muestra un escenario similar al de las primeras migraciones de la figura anterior. Luego sobrecargamos el sistema de forma tal que opere en una asignación cuasiaceptable. El Nodo 0 opera a su capacidad, el Nodo 1 está sobrecargado y el Nodo 2 opera justo por debajo de su precio contratado. Agregamos entonces un cuarto Nodo, al que llamaremos Nodo 3 en el sistema, el cual tiene un contrato sólo con el Nodo 2 pero está a un precio menor que el resto de los tres contratos (Figura 6.6). La figura muestra como el Nodo 2 migra carga hacia el Nodo 3, liberando capacidad para aceptar el exceso de carga proveniente del Nodo 1. Entonces el Nodo 2 migra este exceso de carga hacia el Nodo 3 porque es más barato pagarle al Nodo 3 que procesar la carga localmente. El sistema, una vez más, alcanza una asignación aceptables. Este segundo experimento muestra que nuestro protocolo es capaz de escalar incrementalmente: los nodos pueden agregarse al sistema en tiempo de ejecución.

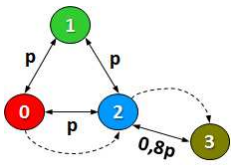


FIGURA 6.6: Contrato nuevo Nodo 3

6.5.2 Limitaciones y extensiones

A continuación discutiremos las limitaciones de nuestro protocolo y las posibles extensiones que permitan solucionar las mismas. Para los experimentos con el simulador de MPA se trabajó únicamente con granularidad pequeña, *i.e.,* nodos de procesamiento individuales; sería interesante también trabajar con granularidad de grupos de nodos. A continuación describiremos de qué forma puede emplearse MPA en sistemas federados.

En un sistema federado cada participante cuenta con su catálogo global, nodos de procesamiento, clientes y sus orígenes de datos. Obviamente dado que los participantes son entidades autónomas, no permiten acceso irrestricto a sus sistemas. Como consecuencia, en lugar de comunicarse en forma directa con los catálogos globales o los nodos de procesamiento de sus sistemas pares, emplean túneles seguros (*e.g.,* implementados mediante TLS) para toda comunicación.

La forma de colaboración más simple de colaboración es que un participante le provea a otro un stream de datos. Dado que el catálogo global de un participante

contiene únicamente información sobre los streams que existen localmente, para que un stream cruce límites administrativos debe estar definido en ambos dominios. Una vez definido en ambos dominios, un stream puede ser parte de dos dominios administrativos y comunicarse empleando el correspondiente túnel.

También debería ser posible la colaboración en ambientes más complejos, donde temporalmente un participante efectúa procesamiento perteneciente originalmente a un partner. Para migrar carga entre dominios administrativos los participantes utilizan definiciones remotas en lugar de la instanciación remota simple empleada en el seno de un participante. Una definición remota especifica de qué forma un operador en un dominio se mapea a otro operador en un dominio diferente. Este mapeo debe ser conocido por el proceso que controla el túnel. De manera offline se realizan las definiciones remotas; y en tiempo de ejecución, cuando un camino de operadores en un diagrama de consultas necesita ser migrado hacia otro dominio, todos los requerimientos por parte del proceso que controla el túnel instancia los correspondientes operadores en su dominio. Las definiciones remotas permiten que los participantes puedan mover carga con un overhead relativamente bajo, comparado con las técnicas usuales de migración de procesos.

La técnica arriba descripta facilita la colaboración, y por ello creemos que es interesante explorar esta línea en trabajos futuros.

6.6 Resumen

En este capítulo presentamos nuestra evaluación del protocolo MPA. Mostramos mediante simulación que logra buenas distribuciones de carga en cualquier entorno, inclusive bajo entornos heterogéneos. En una red de 995 nodos y aproximadamente 10 contratos por nodo, el sistema obtiene asignaciones cuasiaceptables para todas las configuraciones simuladas: sobrecarga, baja carga, contratos de precio fijo y contratos heterogéneos. En un sistema uniforme, un rango de precio igual a la mitad del diámetro de la red (y la mitad del valor teórico mínimo) es suficiente para lograr una asignación aceptable en un sistema con baja carga y una asignación cuasiaceptables para un sistema sobrecargado. Si bien no siempre consiguen asignaciones aceptables en todas las configuraciones, los contratos de precio fijo también obtienen buenas distribuciones de carga. Bajo topologías generadas al azar y con solamente cinco contratos por nodo es suficiente para que un sistema de 995 nodos utilice más del 95 % de la capacidad disponible cuando el sistema está sobrecargado, y que para el caso de baja carga pueda asegurar que más del 95 % de las tareas estén debidamente asignadas. Los contratos de precio fijo y la heterogeneidad provocan que

algunos recursos permanezcan ociosos, volviendo más difícil la obtención de asignaciones aceptables. No obstante, incluso bajo estos escenarios, MPA siempre consigue estar cerca (a un muy bajo porcentaje de distancia) de una asignación aceptable. Además mostramos en este capítulo que la convergencia hacia asignaciones aceptables o cuasiaceptables ocurre rápidamente, especialmente si se están utilizando contratos de precio fijo. También mostramos que MPA logra enmascarar la mayor parte de las variaciones de carga, incluso frente a carga que varía significativamente, (*e.g.,* en promedio una nueva tarea agregada y otra eliminada por unidad de tiempo por nodo), los movimientos de carga reasignan un número de tareas por debajo del 6 % de la variación en la carga ofrecida. Por último vimos que MPA se comporta adecuadamente utilizando datos de entrada reales.

Capítulo 7

Conclusiones y Resultados Obtenidos

Índice

7.1. Procesamiento de streams tolerante a fallas 192
7.2. Gestión de carga en sistemas federados 195
7.3. Trabajos Futuros 196

If I have ever made any valuable discoveries, it has been due more to patient attention, than to any other talent.

— *Isaac Newton*

En este trabajo nos enfocamos especialmente en dos problemas: la tolerancia a fallas en un DSMS distribuido y la gestión de carga en un sistema federado. A continuación resumiremos nuestras contribuciones y delinearemos algunos tópicos de interés que abren un conjunto de posibles líneas de investigación.

7.1 Procesamiento de streams tolerante a fallas

En un DSMS distribuido, especialmente en uno donde los orígenes de los datos y los nodos de procesamiento se encuentran diseminados en una WAN, pueden ocurrir diferentes tipos de fallas: los nodos de procesamiento (software o hardware) pueden fallar, la comunicación entre nodos puede verse interrumpida y las fallas de red pueden incluso provocar que el sistema quede particionado.

Es fácil asumir que dichos tipos de fallas pueden causar problemas en el procesamiento de streams, afectando la correctitud de los resultados (streams de salida).

Más aún, incluso podrían causar que el sistema dejase de producir resultados. Los esquemas previos de tolerancia a fallas aplicados al procesamiento de streams o bien no intentan solucionar las fallas de red [HBR+05] o estrictamente favorecen la consistencia por sobre la disponibilidad, requiriendo que exista al menos una copia completamente conectada con el diagrama de consulta en cada instante de tiempo para poder continuar con el procesamiento [BHS09]. Estos esquemas no cubren las necesidades de todas las aplicaciones de procesamiento de streams, pues muchas de ellas pueden progresar incluso empleando resultados aproximados; más aún, podrían incluso dar más importancia a la disponibilidad que a la consistencia. Ejemplos de aplicaciones de este tipo incluyen el monitoreo de redes, detección de intrusos a nivel de red (NIDS), monitoreo de entornos empleando redes de sensores e incluso algunas aplicaciones financieras. Sin embargo, estas aplicaciones pueden obtener aún mayor provecho del sistema subyacente si pueden determinar si los resultados más recientes son precisos o no, y claro está si pueden eventualmente obtener los valores finales correctos. Creemos que es importante para un esquema de recuperación de fallas poder alcanzar estos requerimientos de manera flexible para poder soportar la diversidad de tipos de aplicaciones en un único framework.

Contribución principal: presentamos el protocolo *Disponibilidad y Consistencia Ajustable a las Aplicaciones* (DCAA), un sistema de tolerancia a fallas basado en replicación aplicable al procesamiento de streams que persigue los objetivos arriba mencionados. DCAA permite que aplicaciones con diferentes requerimientos de disponibilidad y consistencia puedan especificar el máximo nivel de latencia de procesamiento que pueden tolerar, y que puedan procesar cualquier tupla disponible dentro del límite de tiempo preestablecido. Si asumimos que todas las tuplas necesarias pueden bufferearse, entonces DCAA garantiza consistencia eventual, *i.e.,* todas las réplicas eventualmente procesan las mismas tuplas de entrada en el mismo orden y las aplicaciones cliente eventualmente ven los streams de salida en forma completa y correcta.

Resumen del enfoque: la principal característica de DCAA es que permite que cada nodo (y cada una de sus réplicas) maneje su propia disponibilidad y consistencia mediante el monitoreo de sus streams de entrada, suspendiendo o demorando el procesamiento de las tuplas de entrada al producirse una falla tanto como sea necesario, y corrigiendo resultados anteriores luego de solucionada la misma.

DCAA utiliza un modelo de streams de datos que distingue entre tuplas estables y tentativas, donde estas últimas resultan del procesamiento parcial de las

entradas y que dados sus potenciales errores, luego pueden ser corregidas. Para asegurar la consistencia en tiempo de ejecución, nuestro protocolo utiliza un operador de serialización de datos denominado `SUnion`. Una vez solucionada la falla se debe recuperar la consistencia; por ello se investigaron dos técnicas principales: checkpoint/redo y undo/redo.

Principales resultados: nuestro análisis permite concluir que DCAA permite manejar tanto fallas únicas como múltiples, incluso soporta fallas concurrentes. El protocolo permite asegurar consistencia eventual mientras mantiene, siempre y cuando sea posible, el nivel de disponibilidad requerido bajo entornos de nodo único o distribuidos.

Descubrimos que reconciliar el estado de un DSMS empleando checkpoint/redo logra una reconciliación más veloz y menos costosa que utilizando undo/redo. Más aún, para evitar el overhead causado por los checkpoints periódicos y circunscribir la recuperación únicamente a caminos afectados por las fallas, el protocolo posee operadores que efectúan el checkpoint y reinicializan los correspondientes estados en respuesta a la primer tupla tentativa o tupla `undo` que procesan.

Cuando los buffers están acotados, nuestro protocolo se vuelve particularmente adecuado para los diagramas de consulta capaces de converger (*i.e.,* un grafo dirigido, libre de bucles y tipo workflow compuesto por operadores y streams interconectados cuyo estado siempre converge hacia el mismo estado luego de procesar suficiente cantidad de tuplas). Los diagramas de consulta capaces de converger hacen posible que DCAA pueda asegurar que el sistema eventualmente converja hacia un estado anterior consistente y que las tuplas tentativas más recientes sea corregidas, independientemente de la duración de la falla.

Como parte de DCAA investigamos técnicas que apuntaron a reducir el número de tuplas tentativas producidas por un DSMS sin comprometer el requerimiento de disponibilidad preestablecido, asegurando al mismo tiempo consistencia eventual. La mejor estrategia que encontramos consiste en que el primer operador `SUnion` que detecta una falla se bloquee hasta alcanzar la latencia de procesamiento incremental máxima. Si la falla persiste los `SUnion`s deben procesar las nuevas tuplas sin demoras, evitando incurrir en mayores retrasos. Con este enfoque es posible ocultar fallas durante la latencia de procesamiento incremental especificada sin introducir inconsistencia, independientemente del tamaño del DSMS. Para mantener el requerimiento de disponibilidad frente a

fallas prolongadas demostramos que es necesario que los nodos procesen tuplas tentativas durante los estados de falla y de estabilización. Adicionalmente mostramos de qué forma DCAA le posibilita al nodo este accionar.

En resumen, mostramos que es posible la construcción de un sistema de tolerancia a fallas aplicable al procesamiento distribuido de streams y que permite al mismo tiempo, adecuarse a fallas tanto de nodos como de red y soportar aplicaciones con diferentes requerimientos de disponibilidad y consistencia.

7.2 Gestión de carga en sistemas federados

En un sistema distribuido federado, los participantes individuales administran sólo un subconjunto de recursos. Los DSMS federados constituyen sistemas de este tipo, sin embargo también existen otros sistemas federados, como por ejemplo los sistemas *grid* [BAG00, BSG+01, FK98, TTL05], *peer-to-peer* [CFV03, DKK+01, LFSC03, NWD03, RD01, SMK+01, VCCS03] y sistemas basados en web services [W3C02, KL03]. En estos entornos, los participantes individuales pueden necesitar una cantidad inusual de recursos para hacer frente a picos de operación. Alternativamente los participantes pueden colaborar para gestionar su exceso de carga. El desafío en este tipo de colaboraciones radica en que los participantes autónomos tratan de maximizar sus propias utilidades en lugar de las del sistema como un todo.

Contribución principal: los enfoques previos basados en economías de cómputo [BAG00, BSG+01, SAL+96, WHH+92] mostraron no lograr suficiente aceptación, mientras que los contratos bilaterales suelen regular las colaboraciones entre pares autónomos [FSP00, Cor11, KL03, Rac11, GS05, Ver99, WS04, Wus02]. Nuestra propuesta se basa en un enfoque basado en contratos para gestión de carga en entornos federados, a la cual denominamos *Mecanismo de Precio Acotado* (MPA). Nuestro protocolo emplea contratos privados entre pares, en los cuales los participantes negocian dichos contratos en forma offline para regular los movimientos de carga en tiempo de ejecución. A diferencia de trabajos previos en la materia con participantes egoístas (*selfish*), el objetivo de MPA no es lograr un balance de carga óptimo sino asegurar que los participantes operen dentro de sus capacidades preestablecidas.

Resumen del enfoque: en MPA los participantes interactúan unos con otros en dos diferentes escalas de tiempo. Los participantes negocian offline contratos

privados entre pares. Cada contrato especifica un pequeño rango de precios que se usa para negociar el pago a un partner por el procesamiento de carga en tiempo de ejecución. Además un contrato especifica la duración de la unidad de movimiento de carga. En tiempo de ejecución los participantes emplean sus contratos para migrar carga hacia sus partners, y al hacerlo mejorar su utilidad. Los participantes negocian dinámicamente el precio final dentro del rango contratado y la cantidad de carga que deben migrar.

Principales resultados: nuestro análisis muestra que cuando los contratos especifican precios fuertemente acotados para las cargas y los participantes utilizan sus contratos de forma tal de maximizar su utilidad, la asignación de carga del sistema siempre converge hacia una asignación aceptable (o al menos cuasiaceptable). Aún si se trata de un sistema grande y heterogéneo es suficiente la existencia de un número pequeño de contratos por participante para asegurar una asignación cuasiaceptable, independientemente de la carga de los sistemas (funciona tanto para sistemas sobrecargados como no sobrecargados). Es interesante notar que incluso cuando dichos contratos no derivan en una asignación aceptable, generalmente bajo contratos de precio fijo se obtienen buenas distribuciones de carga, generándose escenarios donde la mayor parte de los excesos de carga se reasignan, logrando la utilización de la mayor parte de las capacidades de procesamiento.

Además mostramos que generalmente los participantes acuerdan un precio final sin negociar, y que la mayor parte de las convergencias hacia una distribución de carga final ocurre rápidamente, especialmente si se emplean contratos de precio fijo.

Finalmente y gracias a que los precios son casi fijos, MPA puede ocultar efectivamente una gran parte de las variaciones de carga, pues dichas variaciones suelen frecuentemente ocurrir fuera de los límites de los contratos.

En resumen, los contratos permiten que los participantes construyan o maximicen relaciones de preferencia de larga duración, las cuales gracias a precios acotados prenegociados proveen predictibilidad y escalabilidad en tiempo de ejecución. Además, facilitan la parametrización de servicios y la discriminación de precios mientras siguen llevando a buenas distribuciones de carga, logrando que nuestro enfoque sea entonces más práctico y liviano que lo presentado en propuestas previas. Avisoramos entonces así será también en la práctica.

7.3 Trabajos Futuros

Nuestro estudio de tolerancia a falla con DCAA y de gestión de carga con MPA abre nuevas posibilidades para trabajos futuros. En los dos capítulos previos se presentaron varios de dichos temas abiertos; y a continuación se resumen los más importantes.

Una de las principales oportunidades de trabajo futuro sobre DCAA es su falta de información de precisión. En el protocolo actual las tuplas solamente se etiquetan como tentativas o estables. Sería interesante mejorar los operadores agregándoles la capacidad de leer información de precisión a partir de sus tuplas de entrada y computar la misma información para sus tuplas de salida, especialmente cuando no se encuentra disponible alguno de sus streams de entrada. La información de precisión también puede ayudar a determinar cuándo las fallas afectan sólo a subconjuntos de resultados de salida. En lugar de etiquetar todas las salidas como tentativas, algunas de ellas podrían tener un alto grado de precisión mientras que otras podrían ser poco precisas.

Otra oportunidad de mejora sobre el sistema actual puede ser el agregado de restricciones de integridad sobre los streams. Dado que diferentes tipos de fallas afectan a diversas partes del sistema, en algunos casos puede no tener sentido correlacionar o mezclar ciertos streams tentativos. Sería interesante que las aplicaciones pudiesen definir restricciones de integridad describiendo cuándo están dadas las condiciones para considerar todavía útil la información transportada por los streams y cuándo pueden entonces combinarse. El sistema podría utilizar estas restricciones para asegurar la calidad de los datos de salida e incluso para lograr un algoritmo más efectivo para el cambio de vecino upstream.

Un tercer problema interesante podría ser extender DCAA para soportar una mayor variedad de tipos de fallas, como por ejemplo las fallas Bizantinas, donde un nodo (posiblemente malicioso) produzca resultados erróneos, o la falla (tipo crash) simultánea de todas las réplicas de uno o más nodos de procesamiento.

Uno de los principales problemas abiertos relacionados con MPA es su foco en reaccionar ante la ocurrencia de nodos sobrecargados, en lugar de hacer posible que los participantes puedan reservar recursos anticipadamente y así suavizar los “picos” de carga. Migrar la carga existente maximiza la utilización de recursos por parte de los participantes, pero es claro que este modelo no es viable para cualquier tipo de aplicación. Algunas aplicaciones, como por ejemplo la ejecución de una simulación larga en una plataforma de computación distribuida compartida, desperdiciaría una gran cantidad de recursos si los clientes tuviesen que iniciar estas aplicaciones en

orden para verificar la existencia de suficientes recursos disponibles. Si extendiésemos los contratos para permitir alguna forma de reserva de recursos podríamos extender entonces el alcance de nuestro protocolo de gestión de carga.

Un segundo tema de interés relacionado con la gestión de carga es explorar la posibilidad de que los participantes utilicen diferentes tipos de funciones de costo, incluso no necesariamente monótonamente crecientes y cóncavas hacia arriba. Para algunos tipos de recursos las funciones con pasos o las funciones cóncavas pueden mejorar el modelo de procesamiento de costos. Más aún, dado que la carga total varía con el tiempo, los participantes pueden también adquirir o deshacerse de los recursos que estén provocando variaciones en sus funciones de costo en diferentes instantes de tiempo. Es por ello desafiante estudiar cómo nuestro enfoque basado en contratos podría extenderse para diferentes tipos de modelos de costo.

Finalmente, en este trabajo investigamos los problemas de tolerancia a fallas y la gestión de carga en forma aislada. Sin embargo estos dos problemas están fuertemente ligados. De hecho, una vez que una falla se soluciona el participante debe procesar datos anteriores para asegurar consistencia eventual. Las tareas adicionales de reconciliación de estado pueden aumentar significativamente la carga del participante, y por ello creemos que podría ser beneficioso en algunos casos enviar un subconjunto de dichas tareas hacia un par considerado confiable. Dado que el procesamiento continua desde un checkpoint, migrar una tarea de reconciliación de estado podría ser aún menos costosa que migrar una tarea que se encuentra en ejecución. Un problema interesante es cómo integrar la existencia de este tipo de réplicas temporales en el protocolo DCAA y cómo extender los contratos de MPA para soportar estos nuevos tipos de tareas cuyos requerimientos de recursos son de alguna manera flexibles, al mismo tiempo que los requerimientos de disponibilidad se vuelven más estrictos. En general, en un sistema federado, los participantes pueden utilizar a sus pares no sólo para hacer frente al exceso de carga sino también como medida de tolerancia a fallas y replicación. Esta posibilidad de trabajo futuro suena muy prometedora, al expandir el estudio de contratos capaces de promocionar este tipo de colaboraciones.

> *I haven't a clue as to how my story will end. But that's all right. When you set out on a journey and night covers the road, you don't conclude the road has vanished. And how else could we discover the stars?*
>
> — *Anonymous*

Bibliografía

[61b02] The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[ABB+04] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava y J. Widom. STREAM: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, Marzo 2004.

[ABKM01] David Andersen, Hari Balakrishnan, Frans Kaashoek y Robert Morris. Resilient overlay networks. Reporte Técnico, 2001.

[ABW02] Arvind Arasu, Shivnath Babu y Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical Report 2002-57, Stanford InfoLab, 2002. A short version of this technical report appears in the proceedings of the 9th International Conference on Data Base Programming Languages (DBPL 2003). The most recent version this technical report is available on this publications server as technical report number 2003-67, titled The CQL Continuous Query Language: Semantic Foundations and Query Execution, at http://dbpubs.stanford.edu/pub/2003-67.

[ABW06] Arvind Arasu, Shivnath Babu y Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15:121–142, Junio 2006.

[Ac04] Yanif Ahmad y Uğur Çetintemel. Network-aware query processing for stream-based applications. En *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, páginas 456–467. VLDB Endowment, 2004.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker y Richard Tibbetts. Linear road: a stream data management benchmark. En *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, páginas 480–491. VLDB Endowment, 2004.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno y Vijay Machiraju. *Web Services: Concepts, Architecture and Applications.* Springer Verlag, 2004.

[ACM04] Panayotis Antoniadis, Costas Courcoubetis y Robin Mason. Comparing economic incentives in peer-to-peer networks. *Comput. Netw.*, 46:133–146, Septiembre 2004.

[AD88] R Axelrod y D Dion. The further evolution of cooperation. *Science*, (242):138–5, 1988.

[AGK+95] G. Alonso, Alonso Gunthor, M. Kamath, D. Agrawal, A. El Abbadi y C. Mohan. Exotica/FMDC: Handling disconnected clients in a workflow management system. páginas 99–110, 1995.

[AJ06] Christina Aperjis y Ramesh Johari. A peer-to-peer system as an exchange economy. En *Proceeding from the 2006 workshop on Game theory for communications and networks*, GameNets '06, New York, NY, USA, 2006. ACM.

[AL81] T. Anderson y P. A. Lee. Fault tolerance : principles and practice, 1981.

[AM97] Gustavo Alonso y C. Mohan. WFMS: The next generation of distributed processing tools. En *Advanced Transaction Models and Architectures*, páginas 35–62. 1997.

[AM03] Arvind Arasu y Gurmeet Manku. Approximate counts and quantiles over sliding windows. Technical Report 2003-72, Stanford InfoLab, Diciembre 2003.

[APB09] M. Allman, V. Paxson y E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), Septiembre 2009.

[ARM97] G. Alonso, B. Reinwald y C. Mohan. Distributed data management in workflow environments. En *Proceedings of the 7th International Workshop on Research Issues in Data Engineering (RIDE '97) High Performance Database Management for Large-Scale Applications*, RIDE '97, páginas 82–90, Washington, DC, USA, 1997. IEEE Computer Society.

[ASSC02a] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam y E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[ASSC02b] Lan F. Akyildiz, Welljan Su, Yogesh Sankarasubramaniam y Erdal Cayirci. A survey on sensor networks. 2002.

[BACdK02] Houssein Ben-Ameur, Brahim Chaib-draa y Peter Kropf. Multi-item auctions for automatic negotiation. CIRANO Working Papers 2002s-68, CIRANO, Julio 2002.

[BAG] Rajkumar Buyya, David Abramson y Jonathan Giddy. The evaluation of grid resource trading middleware services.

[BAG00] Rajkumar Buyya, David Abramson y Jonathan Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. *High-Performance Computing in the Asia-Pacific Region, International Conference on*, 1:283, 2000.

[Bar83] B Barber. The logic and limits of trust, 1983.

[BBC+04a] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts y Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13:2004, 2004.

[BBC+04b] Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eduardo F. Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts y Stanley B. Zdonik. Retrospective on aurora. *VLDB J.*, 13(4):370–383, 2004.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani y Jennifer Widom. Models and issues in data stream systems. En Popa [Pop02], páginas 1–16.

[BBDM03] Brian Babcock, Shivnath Babu, Mayur Datar y Rajeev Motwani. Chain: Operator scheduling for memory minimization in data stream systems. En *SIGMOD Conference*, páginas 253–264, 2003.

[BCH+09] Mark Burgin, Masud H. Chowdhury, Chan H. Ham, Simone A. Ludwig, Weilian Su y Sumanth Yenduri, editores. *CSIE 2009, 2009 WRI World Congress on Computer Science and Information Engineering, March 31 - April 2, 2009, Los Angeles, California, USA, 7 Volumes*. IEEE Computer Society, 2009.

[BCM+99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom y Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. *Distributed Computing Systems, International Conference on*, 0:0262, 1999.

[BDM04] Brian Babcock, Mayur Datar y Rajeev Motwani. Load shedding for aggregation queries over data streams. En *In IEEE ICDE Conference*, páginas 350–361, 2004.

[BDR04] James Bailey, Guozhu Dong y Kotagiri Ramamohanarao. On the decidability of the termination problem of active database systems. *Theor. Comput. Sci.*, 311:389–437, Enero 2004.

[Ben05] John Bent. *Data-Driven Batch Scheduling*. PhD thesis, University of Wisconsin, Madison, Mayo 2005.

[BGH86] Joel Bartlett, Jim Gray y Bob Horst. Fault tolerance in tandem computer systems, 1986.

[BGKSK09] Itzik Ben-Gan, Lubor Kollar, Dejan Sarka y Steve Kass. *Inside Microsoft SQL Server 2008: T-SQL Querying*. Microsoft Press, 1st edition, 2009.

[BGR+90] G. Beutler, W. Gurtner, M. Rothacher, U. Wild y E. Frei. Relative static positioning with the global positioning system: basic technical considerations. En N. Bock, Y.; Leppard, editor, *Global Positioning System: An Overview, IAG Symposia*, volume 102, páginas 1–23. Springer Verlag, Enero 1990.

[BHS09] Magdalena Balazinska, Jeong-Hyon Hwang y Mehul A. Shah. Fault-tolerance and high availability in data stream management systems. En Liu y Özsu [LÖ09], páginas 1109–1115.

[BKR98] Jonathan Bredin, David Kotz y Daniela Rus. Market-based resource control for mobile agents. En *Proceedings of the second international conference on Autonomous agents*, AGENTS '98, páginas 197–204, New York, NY, USA, 1998. ACM.

[BlDGs03] Miguel L. Bote-lorenzo, Yannis A. Dimitriadis y Eduardo Gómez-sánchez. Grid characteristics and uses: A grid definition. En *Across Grids 2003, LNCS 2970*, páginas 291–298, 2003.

[BMB+00] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel y Mark Spiteri. Generic support for distributed applications. *Computer*, 33:68–76, 2000.

[Bre01] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5:46–55, 2001.

[BSC01] P. Bhoj, S. Singhal y S. Chutani. SLA management in federated environments. *Comput. Netw.*, 35:5–24, Enero 2001.

[BSG+01] Rajkumar Buyya, Heinz Stockinger, Jonathan Giddy, David Abramson, Author One, Author Two y U. Thirdone. Economic models for management of resources in peer-to-peer and grid computing. Computational economics, EconWPA, 2001.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing y Stanley B. Zdonik. Scalable distributed stream processing. En *CIDR*, 2003.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss y Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. En *CIDR*, 2003.

[CcR+03] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack y Mike Stonebraker. Operator scheduling in a data stream manager. En *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, páginas 838–849. VLDB Endowment, 2003.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian y Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. En Chen et al. [CNB00b], páginas 379–390.

[CF02] Sirish Chandrasekaran y Michael J. Franklin. Streaming queries over streaming data. En *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, páginas 203–214. VLDB Endowment, 2002.

[CF04] Sirish Chandrasekaran y Michael Franklin. Remembrance of streams past: overload-sensitive management of archived streams. En *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, páginas 348–359. VLDB Endowment, 2004.

[CFV03] Brent Chun, Yun Fu y Amin Vahdat. Bootstrapping a distributed computational economy with peer-to-peer bartering. En *WORKSHOP ON ECONOMICS OF PEER-TO-PEER SYSTEMS*, 2003.

[CG98] Luca Cardelli y Andrew D. Gordon. Mobile ambients. En *In Proceedings of POPL'98*. ACM Press, 1998.

[CGM10] Badrish Chandramouli, Jonathan Goldstein y David Maier. High-performance dynamic pattern matching over disordered streams. *PVLDB*, 3(1):220–231, 2010.

[Chu01] BrentÑee Chun. *Market-based cluster resource management*. PhD thesis, 2001. Chair-Culler, David E.

[CJK+04] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck y Divesh Srivastava. Holistic UDAFs at streaming speeds. En *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, páginas 35–46, New York, NY, USA, 2004. ACM.

[CJS03] Chuck Cranor, Theodore Johnson y Oliver Spataschek. Gigascope: a stream database for network applications. En *In SIGMOD*, páginas 647–651, 2003.

[CJSS03] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck y Vladislav Shkapenyuk. The gigascope stream database. *IEEE Data Eng. Bull.*, 26(1):27–32, 2003.

[CL99] Miguel Castro y Barbara Liskov. Practical byzantine fault tolerance. En *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, páginas 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[CL02] Miguel Castro y Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Noviembre 2002.

[Cla71] E H Clarke. Multipart pricing of public goods. *Public Choice*, páginas 17–33, 1971.

[CLL78] A. Costes, C. Landrault y J.-C. Laprie. Reliability and availability models for maintained systems featuring hardware failures and design faults. *IEEE Transactions on Computers*, C-27:548–560, 1978. Special Issue on Fault-Tolerant Computing.

[CM01] Enzo Cialini y John Macdonald. Creating hot snapshots and standby databases with IBM DB2 universal database(TM) V7.2 and EMC timefinder(TM). *DB2 Information Management White Papers*, Septiembre 2001.

[CMM97] Anthony Chavez, Alexandros Moukas y Pattie Maes. Challenger: a multi-agent system for distributed resource allocation. En *Proceedings of the first international conference on Autonomous agents*, AGENTS '97, páginas 323–331, New York, NY, USA, 1997. ACM.

[CNB00a] Weidong Chen, Jeffrey F. Naughton y Philip A. Bernstein, editores. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 2000.

[CNB00b] Weidong Chen, Jeffrey F. Naughton y Philip A. Bernstein, editores. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 2000.

[Cor02] Oracle Corporation. Oracle data guard: Disaster recovery for sun oracle database machine. http://www.datarecoverycenter.info/oracle-data-guard-disaster-recovery-for Marzo 2002.

[Cor11] IBM Corporation. Web service level agreements (WSLA) project. http://www.research.ibm.com/wsla/, Enero 2011.

[Cri93] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34:56–78, 1993.

[CRW00] Antonio Carzaniga, David S. Rosenblum y Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. En *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, páginas 219–227, New York, NY, USA, 2000. ACM.

[CT96] Tushar Deepak Chandra y Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, Marzo 1996.

[CT04] Ludmila Cherkasova y Wenting Tang. Providing resource allocation and performance isolation in a shared streaming-media hosting service. En *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, páginas 1213–1218, New York, NY, USA, 2004. ACM.

[CZ09] Mitch Cherniack y Stanley B. Zdonik. Stream-oriented query languages and operators. En *Encyclopedia of Database Systems*, páginas 2848–2854. 2009.

[DBL95] *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*. ACM Press, 1995.

[DBL03] *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003), 12-15 May 2003, Tokyo, Japan*. IEEE Computer Society, 2003.

[DBL04a] *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA*. IEEE Computer Society, 2004.

[DBL04b] *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: The Hague, Netherlands, 10-13 October 2004*. IEEE, 2004.

[DBL05] *5th International Symposium on Cluster Computing and the Grid (CCGrid 2005), 9-12 May, 2005, Cardiff, UK*. IEEE Computer Society, 2005.

[DBL06] *Grid and Cooperative Computing - GCC 2006, 5th International Conference, Changsha, Hunan, China, 21-23 October 2006, Proceedings*. IEEE Computer Society, 2006.

[DBL07a] *19th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007), 24-27 October 2007, Gramado, RS, Brazil*. IEEE Computer Society, 2007.

[DBL07b] *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey*. IEEE, 2007.

[DBL07c] *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*. IEEE Computer Society, 2007.

[DGR03] Abhinandan Das, Johannes Gehrke y Mirek Riedewald. Approximate join processing over data streams. En *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, páginas 40–51, New York, NY, USA, 2003. ACM.

[DH04] Amol Deshpande y Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. En *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, páginas 948–959. VLDB Endowment, 2004.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Morris y Ion Stoica. Wide-area cooperative storage with CFS. En *SOSP*, páginas 202–215, 2001.

[DWL+06] Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha y Young-Kuk Kim, editores. *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006.

[EA06] Javier Echaiz y Jorge Ardenghi. Extending an ssi cluster for resource discovery in grid computing. En *GCC* [DBL06], páginas 287–293.

[EA09] Javier Echaiz y Jorge Ardenghi. On indirect reputation-based grid resource management. En Burgin et al. [BCH+09], páginas 476–480.

[EAS07] Javier Echaiz, Jorge Ardenghi y Guillermo Ricardo Simari. A novel algorithm for indirect reputation-based grid resource management. En *SBAC-PAD* [DBL07a], páginas 151–158.

[EAWJ02] E.Ñ. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang y David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, Septiembre 2002.

[Ech08] Javier Echaiz. Automatización de la Detección de Intrusos a partir te Políticas de Seguridad, Código 24/ZN14, 2008.

[EFGK03a] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui y Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, Junio 2003.

[EFGK03b] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui y Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, Junio 2003.

[ELZ86] Derek L. Eager, Edward D. Lazowska y John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12:662–675, Mayo 1986.

[ERS+95] Mei Hsu (Ed.), Andreas Reuter, Friedemann Schwenkreis, P. Lang, S. Rausch-schott, W. Retschitzegger, C. Mohan, G. Alonso, R. Gunthor, M. Kamath, Arvola Chan, Kieran Harty, Abraham Bernstein, Abraham Bernstein, Chrysanthos Dellarocas, Chrysanthos Dellarocas, Thomas W. Malone, Thomas W. Malone, John Quimby y John Quimby. Data engineering special issue on workflow systems, 1995.

[ESV03] Cristian Estan, Stefan Savage y George Varghese. Automatically inferring patterns of resource consumption in network traffic. En *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, páginas 137–148, New York, NY, USA, 2003. ACM.

[FABK03] Nick Feamster, David Andersen, Hari Balakrishnan y M. Frans Kaashoek. Measuring the effects of Internet path faults on reactive routing. En *Proc. ACM SIGMETRICS*, San Diego, CA, Junio 2003.

[FCC+03] Yun Fu, Jeffrey S. Chase, BrentÑ. Chun, Stephen Schwab y Amin Vahdat. SHARP: an architecture for secure resource peering. En *SOSP*, páginas 133–148, 2003.

[FGL+96] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch y Alex Shvartsman. Eventually-serializable data services. En *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, páginas 300–309, New York, NY, USA, 1996. ACM.

[FK96a] Ian Foster y Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[FK96b] Ian Foster y Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[FK98] Ian Foster y Carl Kesselman. Computational grids, 1998.

[FK03] Ian Foster y Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[FLP85] Michael J. Fischer, Nancy A. Lynch y Michael S. Paterson. Impossibility of distributed consensus with one faulty process, 1985.

[FNSY96] Donald F. Ferguson, Christos Nikolaou, Jakka Sairamesh y Yechiam Yemini. *Economic models for allocating resources in computer systems*, páginas 156–183. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[FPS01] Joan Feigenbaum, Christos H. Papadimitriou y Scott Shenker. Sharing the cost of multicast transmissions. *J. Comput. Syst. Sci.*, 63:21–41, Agosto 2001.

[FPSS05] Joan Feigenbaum, Christos Papadimitriou, Rahul Sami y Scott Shenker. A BGP-based mechanism for lowest-cost routing. *Distrib. Comput.*, 18:61–72, Julio 2005.

[FS02] Joan Feigenbaum y Scott Shenker. Distributed algorithmic mechanism design: recent results and future directions. En *Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications*, DIALM '02, páginas 1–13, New York, NY, USA, 2002. ACM.

[FSP00] George Fankhauser, David Schweikert y Bernhard Plattner. Service level agreement trading for the differentiated services architecture. Reporte Técnico 59, Swiss Federal Institute of Technology (ETH), Zurich, Enero 2000.

[FT91] D Fudenberg y J Tirole. Game theory, 1991.

[FV02] Yun Fu y Amin Vahdat. SLA-based distributed resource allocation for streaming hosting systems. En *7th International Workshop on Web Content Caching and Distribution*, Agosto 2002.

[FW05] Wenfei Fan, Zhaohui Wu y Jun Yang 0001, editores. *Advances in Web-Age Information Management, 6th International Conference, WAIM 2005, Hangzhou, China, October 11-13, 2005, Proceedings*, volume 3739 of *Lecture Notes in Computer Science*. Springer, 2005.

[Gal77] R Gallager. A minimum delay routing algorithm using distributed computation. *IEEE Transactions on Communications*, 25(1):73–85, 1977.

[Gam88] D Gambetta. Can we trust trust, 1988.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil y Dennis Shasha. The dangers of replication and a solution. En Jagadish y Mumick [JM96], páginas 173–182.

[Gif79] David K. Gifford. Weighted voting for replicated data. En *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, páginas 150–162, New York, NY, USA, 1979. ACM.

[GL02] Seth Gilbert y Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, Junio 2002.

[GM95] Ashish Gupta y Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[GMB85] Hector Garcia-Molina y Daniel Barbara. How to assign votes in a distributed system. *J. ACM*, 32:841–860, Octubre 1985.

[GMRR01] Ashish Gupta, Inderpal Singh Mumick, Jun Rao y Kenneth A. Ross. Adapting materialized views after redefinitions: techniques and a performance study. *Inf. Syst.*, 26(5):323–362, 2001.

[GMS92] H. Garcia-Molina y K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4:509–516, Diciembre 1992.

[GÖ05] Lukasz Golab y M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. En *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, páginas 658–669, New York, NY, USA, 2005. ACM.

[GR92] Jim Gray y Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[Gra85] Jim Gray. Why do computers stop and what can be done about it?, 1985.

[Gre06] Les Green. Service level agreements: an ontological approach. En *Proceedings of the 8th international conference on Electronic commerce: The new e-commerce: innovations for conquering current barriers, obstacles and limitations to conducting successful business on the internet*, ICEC '06, páginas 185–194, New York, NY, USA, 2006. ACM.

[GS05] Sven Graupner y Akhil Sahai. Policy-based resource topology design for enterprise grids. En *CCGRID* [DBL05], páginas 390–397.

[GSS01] Rick Greenwald, Robert Stackowiak y Jonathan Stern. *Oracle Essentials: Oracle9 i, Oracle8 i and Oracle8*. Second edition, Junio 2001.

[Han92] EricÑ. Hanson. Rule condition testing and action execution in ariel. En *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, páginas 49–58, New York, NY, USA, 1992. ACM.

[HBR+05] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Mike Stonebraker y Stan Zdonik. High-availability algorithms for distributed stream processing. *Data Engineering, International Conference on*, 0:779–790, 2005.

[HCH+99] EricÑ. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park y Albert Vernon. Scalable trigger processing. *Data Engineering, International Conference on*, 0:266, 1999.

[hHBR+03] Jeong hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Çetintemel, Michael Stonebraker y Stan Zdonik. A comparison of stream-oriented high-availability algorithms. Reporte Técnico, Brown CS, 2003.

[HHS+02] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward y Paul Webster. The anatomy of a context-aware application. *Wirel. Netw.*, 8:187–197, Marzo 2002.

[HHW97] Joseph M. Hellerstein, Peter J. Haas y Helen J. Wang. Online aggregation. En Peckham [Pec97], páginas 171–182.

[HID03a] Alon Y. Halevy, Zachary G. Ives y AnHai Doan, editores. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 2003.

[HID03b] Alon Y. Halevy, Zachary G. Ives y AnHai Doan, editores. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 2003.

[Hol95] D. Hollingsworth. Workflow management coalition - the workflow reference model. Reporte Técnico, Workflow Management Coalition, Enero 1995.

[HSB+02] Mark Hapner, Rahul Sharma, Rich Burridge, Joseph Fialli y Kim Haase. *Java Message Service API tutorial and reference: messaging for the J2EE platform*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[HXcZ07] Jeong-Hyon Hwang, Ying Xing, Ugur Çetintemel y Stanley B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. En *ICDE* [DBL07b], páginas 176–185.

[IDR07] Zachary G. Ives, Amol Deshpande y Vijayshankar Raman. Adaptive query processing: Why, how, when, and what next? En Koch et al. [KGG+07], páginas 1426–1427.

[ILW+00] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu y Marc Friedman. Adaptive query processing for internet applications. *IEEE Data Eng. Bull.*, 23(2):19–26, 2000.

[Inc11] Mesquite Software Inc. CSIM 20, development toolkit for simulation and modeling. http://www.mesquite.com/, Enero 2011.

[Jac01] Matthew O. Jackson. Mechanism theory, 2001.

[JHvS07] Gian Paolo Jesi, David Hales y Maarten van Steen. Identifying malicious peers before it's too late: A decentralized secure peer sampling service. En *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems (SASO'07)*, páginas 237–246, Boston, MA, 2007. IEEE Computer Society.

[JM96] H. V. Jagadish y Inderpal Singh Mumick, editores. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. ACM Press, 1996.

[JMR05] Theodore Johnson, S. Muthukrishnan y Irina Rozenbaum. Sampling algorithms in a stream operator. En Özcan [Özc05], páginas 1–12.

[JMS95] H. V. Jagadish, Inderpal Singh Mumick y Abraham Silberschatz. View maintenance issues for the chronicle data model. En *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California* [DBL95], páginas 113–124.

[JS03] Yuhui Jin y Rob Strom. Relational subscription middleware for internet-scale publish-subscribe. En *Proceedings of the 2nd international workshop on Distributed event-based systems*, DEBS '03, páginas 1–8, New York, NY, USA, 2003. ACM.

[KAGM96] Mohan Kamath, Gustavo Alonso, Roger Günthör y C. Mohan. Providing high availability in very large worklflow management systems. En *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, páginas 427–442, London, UK, 1996. Springer-Verlag.

[KBG08] YongChul Kwon, Magdalena Balazinska y Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1:574–585, Agosto 2008.

[KBH+88] Leonard Kawell,Jr., Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie y Irene Greif. Replicated document management in a group communication system. En *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, CSCW '88, página 395, New York, NY, USA, 1988. ACM.

[KCW+07] R. Kotla, A. Clement, E. Wong, L. Alvisi y M. Dahlin. Zyzzyva: Speculative byzantine fault tolerance. En *Proc. of the ACM Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, Octubre 2007. ACM.

[KGG+07] Christoph Koch, Johannes Gehrke, MinosÑ. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas y Erich J. Neuhold, editores. *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. ACM, 2007.

[KGM95] Ben Kao y Héctor García-Molina. Advances in real-time systems. capítulo An overview of real-time database systems, páginas 463–486. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[KKK05] Sachin Katti, Balachander Krishnamurthy y Dina Katabi. Collaborating against common enemies. En *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, IMC '05, páginas 34–34, Berkeley, CA, USA, 2005. USENIX Association.

[KL03] Alexander Keller y Heiko Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manage.*, 11:57–81, Marzo 2003.

[KLN09] Murali Kodialam, Wing Cheong Lau y Thyaga Nandagopal. Identifying RFID tag categories in linear time. En *Proceedings of the 7th international conference on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, WiOPT'09, páginas 66–71, Piscataway, NJ, USA, 2009. IEEE Press.

[KS89] James F. Kurose y Rahul Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Trans. Comput.*, 38:705–717, Mayo 1989.

[KS03] M. Frans Kaashoek y Ion Stoica, editores. *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22,2003, Revised Papers*, volume 2735 of *Lecture Notes in Computer Science*. Springer, 2003.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows y Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, 1992.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, Mayo 1998.

[LBK01] Philip M. Lewis, Arthur J. Bernstein y Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2001.

[LFSC03] Kevin Lai, Michal Feldman, Ion Stoica y John Chuang. Incentives for cooperation in peer-to-peer networks, 2003.

[LM02] William Lehr y Lee W. McKnight. Show me the money: contracts and agents in service level agreement markets. *Info - The journal of policy, regulation and strategy for telecommunications*, 4:24–36, Enero 2002.

[LÖ09] Ling Liu y M. Tamer Özsu, editores. *Encyclopedia of Database Systems*. Springer US, 2009.

[LP08] Sébastien Lahaie y David C. Parkes. A modular framework for iterative combinatorial auctions. *SIGecom Exchanges*, 7(2), 2008.

[LRWZ06] Ling Liu, Andreas Reuter, Kyu-Young Whang y Jianjun Zhang, editores. *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. IEEE Computer Society, 2006.

[LS03] Alberto Lerner y Dennis Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. En *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, páginas 345–356. VLDB Endowment, 2003.

[LS05] Zhengqiang Liang y Weisong Shi. Enforcing cooperative resource sharing in untrusted peer-to-peer environment. *ACM Journal of Mobile Networks and Applications (MONET) special*, 10:2005, 2005.

[LSC91] Guy M. Lohman, Amílcar Sernadas y Rafael Camps, editores. *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*. Morgan Kaufmann, 1991.

[LSHW00] Kwok-Wa Lam, Sang H. Son, Sheung-Lun Hung y Zhiwei Wang. Scheduling transactions with stringent real-time constraints. *Inf. Syst.*, 25:431–452, Septiembre 2000.

[LT03] David B. Lomet y Mark R. Tuttle. A theory of redo recovery. En Halevy et al. [HID03b], páginas 397–406.

[LWZ04] Yan-Nei Law, Haixun Wang y Carlo Zaniolo. Query languages and data models for database sequences and data streams. En *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, páginas 492–503. VLDB Endowment, 2004.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MAA+95] C. Mohan, D. Agrawal, G. Alonso, A. El Abbadi, R. Guenthoer y M. Kamath. Exotica: a project on advanced transaction management and workflow systems. *SIGOIS Bull.*, 16:45–50, Agosto 1995.

[MAM+95] Alonso Mohan, G. Alonso, C. Mohan, R. Gunthor, D. Agrawal, A. El Abbadi y M. Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. páginas 1–18, 1995.

[Mar94] S Marsh. Formalising trust as a computational concept. En *PhD thesis in the university of Stirling*, 1994.

[MD00] M. S. Miller y K. E. Drexler. Markets and computation: Agoric open systems. Reporte Técnico, Agoric Inc., 2000.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach y Jeff Kramer. Specifying distributed software architectures. páginas 137–153. Springer-Verlag, 1995.

[Mel02] Jim Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc., New York, NY, USA, 2002.

[MFGH88] Thomas Malone, Richard Fikes, Kenneth Grant y Michael Howardt. Enterprise: A market-like task scheduler for distributed computing environments. *The Ecology of Computation*, páginas 177–205, 1988.

[MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein y Wei Hong. The design of an acquisitional query processor for sensor networks. En *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, páginas 491–502, New York, NY, USA, 2003. ACM.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein y Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30:122–173, Marzo 2005.

[MHM99] Pattie? Guttman Maes, Robert H? y Alexandrou Moukas. Agents that buy and sell. *Communications of the ACM*, (42):81–91, 1999.

[Mil85] D. L. Mills. Network Time Protocol (NTP). RFC 958, Septiembre 1985. Obsoleted by RFCs 1059, 1119, 1305.

[Mil88] Ross M. Miller. Market automation: self-regulation in a distributed environment. *SIGOIS Bull.*, 9:299–308, Abril 1988.

[MLT+05] David Maier, Jin Li, Peter Tucker, Kristin Tufte y Vassilis Papadimos. Semantics of data streams and operators. En *in ICDT*, páginas 37–52, 2005.

[MNG05] Amit Manjhi, Suman Nath y Phillip B. Gibbons. Tributaries and deltas: efficient and robust aggregation in sensor network streams. En Özcan [Özc05], páginas 287–298.

[MR99] Achour Mostéfaoui y Michel Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. En *In Proceedings of the 13th International Symposium on Distributed Computing (DISC'00)*, páginas 49–63, Bratislava, Slavak Republic, 1999.

[MSHR02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein y Vijayshankar Raman. Continuously adaptive continuous queries over streams. En *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, páginas 49–60, New York, NY, USA, 2002. ACM.

[MVLL05] Lisha Ma, Stratis Viglas, Meng Li y Qian Li. Stream operators for querying data streams. En Fan et al. [FW05], páginas 404–415.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin

Rosenstein y Rohit Varma. Query processing, approximation, and resource management in a data stream management system. En *CIDR*, 2003.

[Mye81] Roger Myerson. Optimal auction design. *Mathematics of Operations Research*, (6):58–73, 1981.

[NN10] Hiroshi Nishida y Thinh Nguyen. A global contribution approach to maintain fairness in P2P networks. *IEEE Transactions on Parallel and Distributed Systems*, 21:812–826, 2010.

[NPFI08] Farrukh Nadeem, Radu Prodan, Thomas Fahringer y Alexandru Iosup. A framework for resource availability characterization and on-line prediction in large scale computational grids. Reporte Técnico TR-0130, Institute on Resource, Management and Scheduling, CoreGRID - Network of Excellence, Abril 2008.

[NPS03a] Chaki Ng, David C. Parkes y Margo Seltzer. Strategyproof computing: Systems infrastructures for self-interested parties. En *In Workshop on Economics of Peer-to-Peer Systems*, 2003.

[NPS03b] Chaki Ng, David C. Parkes y Margo Seltzer. Virtual worlds: fast and strategyproof auctions for dynamic resource allocation. En *Proceedings of the 4th ACM conference on Electronic commerce*, EC '03, páginas 238–239, New York, NY, USA, 2003. ACM.

[NR01] Noam Nisan y Amir Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35(1-2):166–196, Abril 2001.

[NR07] Noam Nisan y Amir Ronen. Computationally feasible VCG mechanisms. *J. Artif. Intell. Res. (JAIR)*, 29:19–47, 2007.

[NW82] R R Nelson y S G Winter. An evolutionary theory of economic change, 1982.

[NWD03] Tsuen-Wan Ngan, Dan S. Wallach y Peter Druschel. Enforcing fair sharing of peer-to-peer resources. En Kaashoek y Stoica [KS03], páginas 149–159.

[OBGA00] And Scheduling On, Rajkumar Buyya, Jonathan Giddy y David Abramson. An evaluation of economy-based resource trading. En *Sweep Applications, The Second Workshop on Active Middleware Services (AMS 2000), In conjunction with HPDC 2001*. Kluwer Academic Press, 2000.

[OGP03] David Oppenheimer, Archana Ganapathi y David A. Patterson. Why do internet services fail, and what can be done about it? 2003.

[OJW03] Chris Olston, Jing Jiang y Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. En *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, páginas 563–574, New York, NY, USA, 2003. ACM.

[Ols03] Christopher Alden Remi Olston. *Approximate replication*. PhD thesis, Stanford, CA, USA, 2003. AAI3090652.

[OS95] Gultekin Ozsoyoglu y Richard Thomas Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 7:513–532, Agosto 1995.

[Özc05] Fatma Özcan, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 2005.

[PACR03] Larry Peterson, Tom Anderson, David Culler y Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33:59–64, Enero 2003.

[Par02] David C. Parkes. Price-based information certificates for minimal-revelation combinatorial auctions. En *Revised Papers from the Workshop on Agent Mediated Electronic Commerce on Agent-Mediated Electronic Commerce IV, Designing Mechanisms and Systems*, páginas 103–122, London, UK, 2002. Springer-Verlag.

[PD99] Norman W. Paton y Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31:63–103, Marzo 1999.

[Pec97] Joan Peckham, editor. *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. ACM Press, 1997.

[PLS+06a] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh y Margo Seltzer. Network-aware operator placement for stream-processing systems. En *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, páginas 49–60, Washington, DC, USA, 2006. IEEE Computer Society.

[PLS+06b] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh y Margo I. Seltzer. Network-aware operator placement for stream-processing systems. En Liu et al. [LRWZ06], página 49.

[PMBT01] Nissanka B. Priyantha, Allen K. L. Miu, Hari Balakrishnan y Seth Teller. The cricket compass for context-aware mobile applications. En *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, páginas 1–14, New York, NY, USA, 2001. ACM.

[Pop02] Lucian Popa, editor. *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. ACM, 2002.

[PS04] David C. Parkes y Jeffrey Shneidman. Distributed implementations of vickrey-clarke-groves mechanism. En *AAMAS* [DBL04a], páginas 261–268.

[PSL+05] Peter Pietzuch, Jeffrey Shneidman, Jonathan Ledlie, Matt Welsh, Margo Seltzer y Mema Roussopoulos. Evaluating DHT-based service placement for stream-based overlays, 2005.

[PSW+04] Peter Pietzuch, Jeffrey Shneidman, Matt Welsh, Margo Seltzer y Mema Roussopoulos. Path optimization in stream-based overlay networks, 2004.

[PTL+04] Johan A. Pouwelse, Jacco R. Taal, Reginald L. Lagendijk, Dick H. J. Epema y Henk J. Sips. Real-time video delivery using peer-to-peer bartering networks and multiple description coding. En *SMC (5)*, páginas 4599–4605, 2004.

[Rab89] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36:335–348, 1989.

[Rac11] Rackspace. Rackspace managed hosting (managed SLA). www.rackspace.com/managed_hosting, 2011.

[Rat07] Olga Vladi Ratsimor. *Opportunistic Bartering of Digital Goods and Services in Pervasive Environments*. PhD thesis, University of Maryland, Baltimore County, Agosto 2007.

[RD01] Antony Rowstron y Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. En *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, páginas 188–201, New York, NY, USA, 2001. ACM.

[RD10] Rodrigo Rodrigues y Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, 2010.

[RH02] Vijayshankar Raman y Joseph M. Hellerstein. Partial results for online query processing. En *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, páginas 275–286, New York, NY, USA, 2002. ACM.

[Roe99] Martin Roesch. Snort - lightweight intrusion detection for networks. En *Proceedings of the 13th USENIX conference on System administration*, LISA '99, páginas 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[SA85] Richard Snodgrass y Ilsoo Ahn. A taxonomy of time databases. En *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, páginas 236–246, New York, NY, USA, 1985. ACM.

[SAL+96] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin y Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[San93] Tuomas Sandholm. An implementation of the contract net protocol based on marginal cost calculations. En *Proceedings of the eleventh national conference on Artificial intelligence*, AAAI'93, páginas 256–262. AAAI Press, 1993.

[SBS+00] Tetsuya Shirai, John Barber, Mohan Saboji, Indran Naick y Bill Wilkins, editores. *DB2 Universal Database in Application Development Environments*. Prentice Hall, 2000.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, Diciembre 1990.

[Sch93] Fred B. Schneider. *What good are models and what models are good?*, páginas 17–26. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[SDBL07] Rob Strom, Chitra Dorai, Gerry Buttner y Ying Li. SMILE: distributed middleware for event stream processing. En *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, páginas 553–554, New York, NY, USA, 2007. ACM.

[SGMvM03] Akhil Sahai, Sven Graupner, Vijay Machiraju y Aad P. A. van Moorsel. Specifying and monitoring guarantees in commercial grids through SLA. En *CCGRID* [DBL03], página 292.

[SH98] Mark Sullivan y Andrew Heybey. Tribeca: a system for managing large databases of network traffic. En *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, páginas 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[SHCF03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran y Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. *Data Engineering, International Conference on*, 0:25, 2003.

[SK91] Michael Stonebraker y Greg Kemnitz. The POSTGRES next generation database management system. *Commun. ACM*, 34:78–92, Octubre 1991.

[SL90] Amit P. Sheth y James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22:183–236, 1990.

[SLR96] Praveen Seshadri, Miron Livny y Raghu Ramakrishnan. The design and implementation of a sequence database system. En *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, páginas 99–110, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek y Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. En *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, páginas 149–160, New York, NY, USA, 2001. ACM.

[SP03] Michael L. Scott y Larry L. Peterson, editores. *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. ACM, 2003.

[SPAM91] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal y C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. En Lohman et al. [LSC91], páginas 469–478.

[SS05] Yasushi Saito y Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37:42–81, Marzo 2005.

[Ste97] R Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[Str04] R. Strom. Fault-tolerance in the SMILE stateful publish-subscribe system. En *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, Edinburgh, Scotland, UK, Mayo 2004.

[Sul96] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. En Vijayaraman et al. [VBMS96], páginas 594–606.

[SW04a] Utkarsh Srivastava y Jennifer Widom. Flexible time management in data stream systems. En *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '04, páginas 263–274, New York, NY, USA, 2004. ACM.

[SW04b] Utkarsh Srivastava y Jennifer Widom. Memory-limited execution of windowed stream joins. En *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, páginas 324–335. VLDB Endowment, 2004.

[SY85] Robert E. Strom y Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3:204–226, 1985.

[TcZ+03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack y Michael Stonebraker. Load shedding in a data stream manager. En *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, páginas 309–320. VLDB Endowment, 2003.

[Tec11] Crossbow Technology. Products: Wireless sensor networks. http://www.xbow.com/asset-tracking/products, Enero 2011.

[TGNO92] Douglas Terry, David Goldberg, David Nichols y Brian Oki. Continuous queries over append-only databases. En *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, páginas 321–330, New York, NY, USA, 1992. ACM.

[TGO99] Kian-Lee Tan, Cheng Hian Goh y Beng Chin Ooi. Online feedback for nested aggregate queries with multi-threading. En *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, páginas 18–29, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[TL03] Douglas Thain y Miron Livny. Building reliable clients and servers. En Ian Foster y Carl Kesselman, editores, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[TMS03] Peter A. Tucker, David Maier y Tim Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Eng. Bull.*, 26(1):33–40, 2003.

[TMSF03] Peter A. Tucker, David Maier, Tim Sheard y Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15:555–568, Marzo 2003.

[TMSS07] Peter A. Tucker, David Maier, Tim Sheard y Paul Stephens. Using punctuation schemes to characterize strategies for querying over data streams. *IEEE Trans. Knowl. Data Eng.*, 19(9):1227–1240, 2007.

[Tro99] W. Trockel. Integrating the nash program into mechanism theory. Working Papers 305, Bielefeld University, Institute of Mathematical Economics, 1999.

[TTL05] Douglas Thain, Todd Tannenbaum y Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[TTP95] Douglas B. Terry, Marvin Theimer y Karin Petersen. Managing update conflicts in bayou, a weakly connected replicated storage system. páginas 172–183, 1995.

[UF01] Tolga Urhan y Michael J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. En *Proceedings of the 27th*

International Conference on Very Large Data Bases, VLDB '01, páginas 501–510, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[Urb03] Randy Urbano. Oracle streams replication administrator´s guide, 10g release 1. http://download.oracle.com/docs/cd/B13789_01/server.101/b10728.pdf, Diciembre 2003.

[Vah03] Amin Vahdat. Future directions in distributed computing. capítulo Dynamically provisioning distributed systems to meet target levels of performance, availability, and data quality, páginas 127–131. Springer-Verlag, Berlin, Heidelberg, 2003.

[VBMS96] T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan y Nandlal L. Sarda, editores. *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. Morgan Kaufmann, 1996.

[VCCS03] Vivek Vishnumurthy, Sangeeth Chandrakumar, Sangeeth Ch y Emin Gün Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing, 2003.

[Ver99] Dinesh Verma. *Supporting Service Level Agreements on IP Networks*. Macmillan Technical Publishing, 1999.

[Vic61a] W Vickrey. Counterspeculation, auctions and competitive sealed tenders. journal of finance, 1961.

[Vic61b] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16:8–37, Marzo 1961.

[VN02] Stratis D. Viglas y Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. En *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, páginas 37–48, New York, NY, USA, 2002. ACM.

[vSG10] Maarten van Steen Gian Paolo Jesi, Alberto Montresor. Secure peer sampling. *Elsevier Computer Networks - Special Issue on Collaborative Peer-to-Peer Systems*, 2010.

[W3C02] W3C. W3C architecture domain. http://www.w3.org/2002/ws/, Junio 2002.

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey y Donald F. Ferguson. *Web Services Platform Architecture*. Prentice Hall PTR, 2005.

[WF89] J. Widom y S. J. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems. *SIGMOD Rec.*, 18:36–45, Septiembre 1989.

[WF90] Jennifer Widom y S. J. Finkelstein. Set-oriented production rules in relational database systems. En *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, páginas 259–270, New York, NY, USA, 1990. ACM.

[WHH+92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart y W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Trans. Software Eng.*, 18(2):103–117, 1992.

[WJ95] Michael Wooldridge y Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[WS04] Martin B. Weiss y Seung Jae Shin. Internet interconnection economic model and its analysis: Peering and settlement. *Netnomics*, 6:43–57, Abril 2004.

[Wus02] Edward Wustenhoff. Service Level Agreement in the Data Center. *Sun BluePrint Online*, Abril 2002.

[XHcZ06] Ying Xing, Jeong-Hyon Hwang, Ugur Çetintemel y Stanley B. Zdonik. Providing resiliency to load variations in distributed stream processing. En Dayal et al. [DWL+06], páginas 775–786.

[XZNX08] Lijuan Xiao, Yanmin Zhu, Lionel M. Ni y Zhiwei Xu. Incentive-based scheduling for market-like computational grids. *IEEE Transactions on Parallel and Distributed Systems*, 19:903–913, 2008.

[YB06] Chee Shin Yeo y Rajkumar Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Softw. Pract. Exper.*, 36:1381–1419, Noviembre 2006.

[YG02] Yong Yao y Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[YG03a] Yong Yao y Johannes Gehrke. Query processing in sensor networks. En *CIDR*, 2003.

[YG03b] Yong Yao y Johannes Gehrke. Query processing in sensor networks. En *CIDR*, 2003.

[YG09] Yong Yao y Johannes Gehrke. Continuous queries in sensor networks. En Liu y Özsu [LÖ09], páginas 488–492.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph y John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz y Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Reporte Técnico, EECS Department, University of California, Berkeley, Abril 2001.

Índice Temático

A

Asignación aceptable 114

C

Consistencia Eventual 62
Contrato de precio acotado . . . 131
Contrato de precio fijo 115
Contrato de precio fijo revisado 122
cuasiaceptable 133

E

estado erróneo 55

F

fallas
- bizantinas 57
- clasificación 55
- definición 54
- francas 56
- temporales 56
- terminología 54
- transitorias 56

O

Origen de datos 42

P

Prefijo de una secuencia de tuplas 42

S

stream 42
Sufijo de una secuencia de tuplas 43

T

transición errónea 55
tupla 41, 42

V

vecino downstream 50
vecino upstream 50