UNIVERSIDADE DE VIGO

Dpto. de Enxeñería Telemática
ETSE de Telecomunicación

TESIS DOCTORAL

ARQUITECTURA DE EXTENSIÓN PARA LA TECNOLOGÍA DE SERVICIOS WEB

Autor: Julio Fernández Vilas
Ingeniero en Informática

Director: José J. Pazos Arias
Doctor Ingeniero de Telecomunicación

Marzo de 2.006

ARQUITECTURA DE EXTENSIÓN PARA LA TECNOLOGÍA DE SERVICIOS WEB

TESIS DOCTORAL

ARQUITECTURA DE EXTENSIÓN PARA LA TECNOLOGÍA DE SERVICIOS WEB

Autor: D. Julio Fernández Vilas

Director: Dr. D. José J. Pazos Arias

TRIBUNAL CALIFICADOR

Presidente: Dr. D.

Vocales:

Dr. D.

Dr. D.

Dr. D.

Secretario: Dr. D.

CALIFICACIÓN:

Vigo, a de de .

A papá, a mamá,
promotores involuntarios.

A Bea, a Paula,
conscientes sufridoras.

Agradecimientos

Gracias a Dr. José J. Pazos Arias por darme la oportunidad y el apoyo en este trabajo de tesis. Gracias por tu confianza en ese desconocido que tenía una idea.

Mis especiales agradecimientos a Dra. Ana Fernández Vilas por su apoyo desde antes de empezar el trabajo, por su ánimos, por su paciencia con ese usuario de Windows que soy, por sus ideas, y su apoyo más allá de la familia. Ah, y \begin{gracias} por tu ayuda con este extraño mundo de *tags* \end{gracias}.

A Bea, por apoyarme incondicionalmente, por animarme a empezar cada proyecto (éste entre otros muchos) y aguantar mis largas estancias delante del portátil.

A Paula, gracias por haber llegado en medio de este largo proyecto y trabajar conmigo todas esas tardes de invierno. Después de todo, ambos proyectos comenzasteis en las mismas fechas.

Mis agradecimientos a mamá, Rosi, Lucia, Eva, así como a sus respectivas familias (presentes y futuras).

Gracias también a los compañeros de trabajo, pues en este largo proyecto hay también largas discusiones sobre implementación, sobre la conveniencia del XML, sobre el futuro de los servicios web o sobre etimología o globalización.

Gracias a los amigos de Coruña, a los de Cangas, a los de Pontevedra y a algún individuo de Vigo por haberse interesado en el proyecto.

Y al igual que aquellos que celebran su onomástica el día de Todos Los Santos, gracias a todos aquellos que no figuren aquí y esperaban sentirse públicamente agradecidos. A ellos: perdón y gracias.

Después de todo, la estabilidad y la fuerza necesarias para acometer cualquier proyecto debe surgir de una situación emocional que no se podría alcanzar si todo lo que te rodea, además de las personas, no lo alimenta. Por ello, y sin saberlo, también han ayudado Depeche Mode, Keane, Rod Stewart, Moby, Silje Nergaard y Diana Krall, entre otros.

Resumen

El trabajo de esta tesis se sitúa en el área de los **servicios web**. Éstos conforman una tecnología (un conjunto de ellas en realidad) muy reciente, y en pleno estado de desarrollo.

Los servicios web representan un nuevo avance en las técnicas de desarrollo de software en plataformas distribuidas. Si en su día la evolución de *módulos* a *clases* supuso un gran avance en cuanto a la utilización de técnicas de ingeniería aplicadas al desarrollo de software, la posterior evolución desde las *clases* hasta los *componentes* supuso no solamente un avance en la forma de desarrollar, sino también en la forma en que se diseñan y construyen las aplicaciones, basadas, a partir de ese momento, en paradigmas completamente diferentes a los utilizados hasta entonces, como puede ser la arquitectura cliente/servidor.

El paso siguiente en esta cadena evolutiva ha consistido en migrar desde el uso de componentes basados en tecnologías propietarias, y fuertemente acoplados, hacia un nuevo tipo de componentes, cuyas principales características son el uso de estándares abiertos y el débil acoplamiento.

Desde su nacimiento, los servicios web han seguido un camino marcado por entidades independientes de fabricante y asociaciones de distintos fabricantes, lo que supone un éxito de los procesos de estandarización, pues ayudan al desarrollo de la tecnología de una forma controlada, a la vez que garantizan a los usuarios de dicha tecnología una vida más larga de sus desarrollos en lo que a compatibilidad y evolución tecnológica se refiere.

El funcionamiento de un servicio web es bien sencillo de explicar al consistir en una interacción directa entre una aplicación cliente y una aplicación proveedora que se basa en dos pilares fundamentales: HTTP como transporte de nivel bajo y SOAP como lenguaje para definir los mensajes que se intercambian entre ambas entidades. Ambos pilares son dos estándares claramente establecidos.

Para que una aplicación cliente pueda funcionar, debe producirse antes, en tiempo de desarrollo normalmente, un proceso de enlace o *bind*, por el cual la aplicación cliente "asimila" los esquemas o formatos de los mensajes a intercambiar con el servicio web, así como la lista de operaciones o métodos disponibles.

Sin embargo, el modo de funcionar de los servicios, ligados al mismo tiempo a otros estándares ya consolidados, ha dado lugar a **nuevos problemas** por una parte, y **nuevas oportunidades de mejora** por otra. Los nuevos problemas no son en realidad nuevos, sino que son los asociados al nacimiento de una nueva tecnología que solventa de otra manera problemas que ya habían sido solucionados antes con otras tecnologías. ¿Qué

quiere decir esto? Pues que problemas como la transaccionalidad, el asincronismo, y otros muchos que ya estaban resueltos en otras tecnologías como las basadas en componentes, han de ser resueltos nuevamente siguiendo las especificaciones de la tecnología de servicios web.

Algunos de estos "nuevos problemas" ya han sido solucionados, otros están en vías de solucionarse, y otros están en vías de estudio.

Pero al tiempo que surgen nuevos problemas, también surgen nuevas oportunidades. El hecho de que la tecnología de servicios web se base enteramente en estándares y haya sido apoyada por la práctica totalidad de los fabricantes de software permite que cualquiera pueda colaborar en el desarrollo de dicha tecnología, mediante mecanismos de extensión, o, simplemente, extenderla para uso propio, con la intención de dar cobertura a necesidades que no estén cubiertas por los estándares.

De un conjunto de problemas y carencias que hemos detectado en la tecnología, que serán explicados a lo largo de esta tesis, ha partido la idea de desarrollar una extensión de la tecnología de servicios web basada en el uso de componentes intermedios, que conviertan una interacción directa en una interacción indirecta.

El objetivo básico de esta indirección es en realidad obtener control desde un punto externo a las propias invocaciones. Una vez que un elemento intermedio disponga de esa oportunidad de control, una gran cantidad de servicios de valor añadido podrán ser ofrecidos tanto a entidades cliente como a entidades proveedoras.

Nuestra propuesta de arquitectura de extensión se basa, como hemos dicho, en la creación de elementos intermedios. Dichos elementos serán los responsables de comunicarse de forma directa con las entidades cliente y con las entidades proveedoras. Ambos extremos en este nuevo esquema de invocaciones se comunicarán entre sí mediante una *interacción indirecta*.

Los factores críticos del diseño de nuestra propuesta serán, acorde a lo dicho hasta ahora: la extensibilidad y la compatibilidad.

La extensibilidad es muy importante en nuestra arquitectura. Es de hecho el punto de partida, y se basa en la habilidad de la tecnología de servicios web para ser una tecnología extensible. A través de distintos mecanismos, cada una de las entidades intervinientes en una arquitectura de este tipo puede extenderla según lo necesite. Pero al mismo tiempo, la extensibilidad puede ser utilizada para crear nuevos estándares basados en extensiones de los ya existentes.

La compatibilidad es posiblemente el punto más relevante a considerar a la hora de extender la tecnología de servicios web. Es impensable la creación de nuevos estándares incompatibles con la tecnología existente. Debido a ello nuestra propuesta representa, en gran medida, una abstracción de los

servicios web tal como son concebidos actualmente. Y gracias a este *status* abstracto de nuestra arquitectura, permitimos que sea compatible tanto con la tecnología actual como con cualquier modificación que pudiera aparecer.

Distintos problemas serán identificados y analizados a largo de este trabajo. Problemas como la aplicación de las técnicas de alta disponibilidad o de calidad de servicio a los servicios web. Problemas inherentes al propio funcionamiento de los servicios web, como puede ser la gestión de múltiples proveedores o el tratamiento de errores de invocación, serán también identificados.

Por otra parte, daremos soluciones alternativas que servirán para apoyar la resolución de problemas ya identificados y resueltos en la actualidad, como pueden ser los problemas asociados a la seguridad de los servicios web o los problemas de rendimiento. Las soluciones para estos tipos de problemas serán aplicadas mediante la utilización de distintos roles de los componentes intermedios propuestos en nuestra solución.

Adicionalmente, añadiremos a la tecnología de servicios web nuevas formas de trabajar basadas en técnicas de intermediación, creando nuevas oportunidades de negocio y nuevas arquitecturas que permitirán la entrada de terceras entidades encargadas de ofrecer servicios web con valor añadido.

Para dar solución a los problemas que acabamos de presentar, a lo largo de este trabajo explicaremos las ideas de las que parte nuestra propuesta. Desarrollaremos y construiremos paso a paso la nueva arquitectura de extensión a partir de dichas ideas. Esta arquitectura se basa en dos conceptos fundamentales: la virtualización y el uso de componentes intermedios. Confrontaremos nuestra propuesta con cada uno de los problemas o características de los servicios web actuales. Y, finalmente, presentaremos los resultados de un conjunto de pruebas de simulación que demuestra la viabilidad de nuestra propuesta, a la vez que sirve para demostrar la conveniencia de su adopción.

Abstract

The ideas exposed inside this thesis are located in the web services area, a very new technology (actually, a wide range of technologies) in an early stage of development. Web services represent one step beyond in the software development techniques in the distributed platform environment. The previous evolution from *modules* to *classes* meant a great advance in the use of software engineering techniques applied to software development. On the other hand, the recent evolution from *classes* to *components* has meant not only an advance in the way applications are developed, but also an advance in the way applications are designed and created. From that moment, applications have started to be based in a set of completely different paradigms from those used before, for instance the server/client architecture.

The next step in this evolutionary chain has been the migration from the use of strongly coupled components based in proprietary technologies to a new kind of components whose main features are the use of open standards and loosely-coupled interfaces. At this respect, web services have evolved according to the guidelines established by vendor-independent third-party entities and several associations of vendors. As a result of this kind of standardization process web services technology can evolve in a controlled manner. So, the users can adopt this technology in a secure way, that is, they can develop applications that will survive for a very long time in terms of technological evolution and compatibility.

The way web service technology runs is easy to understand, it consists on a direct interaction between a client application and a provider application. This invocation is based on two well-known mainstays: HTTP as a standard low level transport layer and SOAP as high level language for describing the exchange of messages between clients and providers. Previously, for a client application to run successfully, a *bind* process must be performed at design time. With this binding, the client application assimilates the schemas used by the exchanged messages (message formats) as well as the list of available operations or methods.

Nevertheless, the way web services are used has caused new problems as well as new improvement opportunities. The problems are not new in fact; they are old problems that have arisen again in a new technology (web services) used to solve (in a different way) problems that had been solved previously in other technologies. What does this mean? This means that problems like transactionality or asynchronism, which have already been solved in other technologies like component-based ones, have to be solved again by obeying the specifications stated by web services technology. Despite the fact that some of these "new problems" have already been solved for web services, some others are in process of being solved, and what is more some others are still in an early stage of study.

When new problems arise, new opportunities come out. The fact that the entire web services technology is completely standard-based, together with the fact that it has been fully supported by software vendors, allows anybody to collaborate in the development process. This collaboration can be carried out by using extension mechanisms or by proprietarily extending it to cover any unsolved aspect of the technology.

With the aim of resolving a set of problems and lacks we have discovered and studied in web services technology, an extension of the technology is propose in this thesis. To be precise, we propose to extend the technology by using intermediary elements that allows turning the abovementioned direct interaction into an indirect interaction. The main objective of this form of indirection is to gain control over invocations from a point which is external to that invocations. Once an intermediary element has gained the control over an invocation, it can be the source of a lot of value-added features for clients and providers. The new intermediary elements, essential part of our extension architecture, will be the responsible of supporting the direct communications with the client and the provider entities. Both ends of this new invocation model will exchange messages by using an *indirect interaction*.

The key success factors of the design of our purpose will be, in concordance with what we have stated until now, extensibility and compatibility.

Extensibility is really essential in our architecture, in fact it is the starting point of our purpose. Extensibility represents the ability of web services to be an extensible technology. Through the use of different mechanisms, any party involved in an architecture like this can extend the architecture as needed. But, at the same time, extensibility can be used to build new standards from existent ones.

Compatibility is possibly the most relevant topic to consider when extending the web services technology. The construction of new standards incompatible with current technology is really unthinkable in the web services world. Due to this fact, our purpose represents basically an abstraction of web services as they are understood these days. It is this abstract status of our architecture what allows achieving one of our aims: our purpose is compatible with current technology and is ready (and open) to be adapted to any future variation that could appear.

Several different problems will be identified and addressed all along this thesis: problems like the application of high availability or quality-of-service techniques to the web services architecture but also problems inherent to the way of working of web services, for instance the simultaneous use of more than one provider or the management of invocation errors.

At the same time, we offer alternative and/or complementary solutions

to other problems already identified in the current architecture, like problems related to security or performance in web services. The solutions to this kind of problems will be applied by distinguishing different roles for the intermediary components we propose.

Additionally, we will add new ways or working to the web services technology by using intermediation techniques. The idea of intermediation offers new business opportunities and new architectures which allow third-party entities to participate by offering value-added web services.

In order to solve the problems that we have just explained, all along this work we expose the ideas from which our purposes depart. Then, we develop and construct, step by step, the new extension-based architecture which is based on the use of two main distinguishing points: virtualization and intermediary elements. Also, we confront our purpose with each one of the problems or features of the current web services. Finally, we present the results of a set of simulations which prove the ability of our purpose to solve the problems we have identified. At the same time, we justify the convenience of the adoption of a solution like the one we propose.

Índice

I Introducción **1**

1. Introducción **3**

1.1. El origen 3

1.2. Etapas de la web 4

1.3. Usos de la web 5

1.4. Computación distribuida 7

1.5. Servicios Web 8

1.5.1. Construcción de un servicio web. 10

1.5.2. Utilización de un servicio web. 11

1.5.3. Características de los servicios web. 12

1.6. Carencias de los servicios web 12

1.6.1. Necesidades Futuras. 14

1.7. Estructura de la memoria 15

2. Estado del arte **17**

2.1. Arquitectura actual 17

2.2. Roles actuales 23

2.3. Modelos de invocación 27

2.4. Control y gestión de Errores 31

2.5. Gestión de proveedores 37

2.6. Alta disponibilidad 39

2.7. Calidad de Servicio 45

2.8. Composición 49

2.9. Acuerdos de Nivel de Servicio 54

2.10. Esquemas de intercambio de mensajes 58

2.11. Rendimiento 64

II Virtualización 67

3. Virtualización 69

3.1. Primera propuesta: Virtualización 69

3.2. Segunda propuesta: Nuevos roles 74

3.3. Tercera propuesta: Arquitectura en red 77

3.4. Motor Simple 80

3.5. Motor Independiente 82

3.5.1. Motor Independiente Público 82

3.5.2. Motor Independiente Privado 83

3.6. Motor Integrado 85

4. Motor Independiente 89

4.1. Motor Independiente 89

4.2. Escenarios de uso 91

4.2.1. El motor en la red privada del cliente 91

4.2.2. El motor en la red privada del proveedor 92

4.2.3. El motor en Internet 93

4.3. Encadenamiento de motores 94

5. Ciclo de Vida 97

5.1. Ciclo de vida tradicional 97

5.1.1. Ciclo de Desarrollo (Proveedor) 97

5.1.2. Ciclo de Desarrollo (Cliente) 98

5.1.3. Ciclo de Producción 99

5.1.4. Mantenimiento 100

5.2. Ciclo de vida de un servicio virtual 101

5.2.1. Ciclo de Desarrollo 102

5.2.2. Ciclo de Producción 103

5.2.3. Mantenimiento 105

5.3. Conclusiones 106

III Servicios Web Virtuales **109**

6. VWS **111**
6.1. VWS 111
6.2. Lenguaje de descripción 113

7. Lenguaje VWSDL **115**
7.1. Estructura de un documento 115
7.1.1. Elemento `types` 116
7.1.2. Elemento `provider` 118
7.1.3. Elementos `stylesheet` y `transform` 118
7.1.4. Elemento `method` 120
7.1.5. Elemento `service` 121
7.2. Alcance (*scope*) 121
7.2.1. Elemento `cache` 123
7.2.2. Elemento `event` 124
7.2.3. Elemento `define` 125

8. Lenguaje VWSEL **127**
8.1. Necesidad de extensión 127
8.2. VWSEL 130
8.3. Estructura de un documento VWSEL 130
8.3.1. Elemento `select` 131
8.3.2. Elemento `cache` 134
8.3.3. Elemento `async` 136
8.3.4. Elemento `get` 138

9. Arquitectura de un motor **141**
9.1. Arquitectura 141

IV Rating **147**

10. Rating **149**
10.1. Rating 149
10.2. Expresiones 152
10.3. Facetas 156
10.4. Extensiones a las expresiones 159

10.5. Extensiones a la arquitectura 162
10.6. Selección 165
10.7. Calificación 165
10.7.1. Toma de Medidas 166
10.7.2. Almacenamiento 167
10.8. Repositorios 168

V Aplicaciones de la arquitectura propuesta 173

11. Alta Disponibilidad 175
11.1. Introducción 175
11.2. Solución propuesta 177
11.3. Diseño e Implementación 178
11.3.1. Construcción de un *cluster* de servicios web 178
11.3.2. Selección de nodos 179
11.4. Gestión de un *cluster* 182
11.4.1. Mantenimiento del *cluster* 182
11.4.2. Detección de errores 183
11.4.3. Escalabilidad 184
11.5. Validación de la propuesta 186
11.6. Conclusiones 191

12. Calidad de Servicio 195
12.1. Introducción 195
12.2. Calificación y Evaluación 196
12.3. Validación de la propuesta 197
12.4. Conclusiones 199

13. Mejoras de rendimiento 201
13.1. Introducción 201
13.2. Arquitectura 202
13.3. Implementación 203
13.4. Métodos de *caching* 204
13.5. Implementación del *caching* con VWS 206
13.5.1. Primera: HoCA de 3 o más niveles 206
13.5.2. Segunda: 2LCA con TTL 207

13.5.3. Tercera: 2LCA con sondeo 207
13.5.4. Cuarta: 2LCA con invalidación relajada 209
13.5.5. Quinta: 2LCA con invalidación segura 211
13.6. Validación de la propuesta 213
13.7. Conclusiones 218

14. Asincronismo 221
14.1. Introducción 221
14.2. Propuestas 222
14.2.1. Asincronismo de servicios síncronos 223
14.2.2. Asincronismo entre pares 225
14.3. Conclusiones 231

15. Roles del Motor 235
15.1. Introducción 235
15.2. Propuestas 236
15.2.1. *Proxy* 236
15.2.2. Cortafuegos (*Firewall*) 237
15.2.3. *Broker* 238
15.2.4. Arquitecturas complejas 240
15.3. Conclusiones 241

16. Detección y Gestión de errores 243
16.1. Introducción 243
16.2. Propuestas 244
16.3. Validación de la propuesta 246
16.4. Conclusiones 246

17. Gestión de Proveedores 247
17.1. Introducción 247
17.2. Propuestas 248
17.3. Validación de la propuesta 250
17.4. Conclusiones 250

18. Reducción del Nivel de Acoplamiento 253
18.1. Introducción 253
18.2. Propuestas 255

18.3. Conclusiones 256

19. Otras Aplicaciones **259**

19.1. Acuerdos de Nivel de Servicio 259

19.1.1. Introducción 259

19.1.2. Propuestas 260

19.1.3. Conclusiones 261

19.2. Composición 261

19.2.1. Introducción 261

19.2.2. Propuestas 262

19.2.3. Conclusiones 263

19.3. Publicación y Suscripción con VWS 264

19.3.1. Introducción 264

19.3.2. Propuestas 265

19.4. Conclusiones 265

VI Conclusiones **267**

20. Conclusiones y líneas futuras **269**

20.1. Conclusiones 269

20.2. Líneas futuras 272

VII Apéndices **275**

Apéndice A. SimpleTxForm **277**

A.1. SimpleTxForm (STF) 277

Apéndice B. Lenguaje VWSDL **283**

B.1. Introducción al lenguaje 283

B.2. Esquema del lenguaje 284

Apéndice C. Lenguaje VWSEL **291**

C.1. Introducción al lenguaje 291

C.2. Esquema del lenguaje 292

Apéndice D. Prototipo **295**

D.1. Arquitectura 295

D.2. Características Técnicas 295

D.3. Características Funcionales 296

D.4. Experiencias 297

Bibliografía **299**

PARTE I

Introducción

CAPÍTULO 1

Introducción

1.1. El origen

Cuando las primeras computadoras empezaron a utilizarse de forma comercial, el ordenador era concebido como una gran máquina capaz de automatizar la resolución de complejos problemas matemáticos. El tiempo ha cambiado este modelo, porque, si bien siguen existiendo grandes ordenadores para este mismo propósito, han surgido nuevos paradigmas en el diseño y construcción de sistemas de información. Lo que en su día se denominó microinformática y que hoy en día se conoce como informática distribuida, ha cambiado tremendamente el panorama tecnológico.

Las técnicas de construcción de computadoras han discurrido a lo largo de los últimos 20 años por dos caminos bien diferentes. Por un lado, los grandes ordenadores (*mainframe*) han seguido fabricándose y mejorándose continuamente, constituyendo el mejor ejemplo de lo que se denominan sistemas tolerantes a fallos (*fault tolerant*). Por otra parte, la aparición de la microinformática a principios de los años 80, con la llegada al mercado de los primeros ordenadores personales, ha causado una revolución en el mundo del hardware de ordenadores, dando pie al nacimiento de la actual computación distribuida.

A medida que los ordenadores personales crecían en potencia, surgía la necesidad de conectarlos y compartir recursos. Para ello se crearon las redes de área local o LAN (*Local Area Network*). Estas redes, tenían un ámbito limitado, con un ancho de banda elevado (10 ó 16 Mbps). Las LAN eran utilizadas normalmente dentro de entornos corporativos, con el objetivo de compartir recursos (impresoras, ficheros, aplicaciones, etc.) dentro de una misma empresa. La continua evolución de los equipos domésticos, junto con la de los sistemas medios (*midrange*), creó una necesidad que, posiblemente,

no hubiera surgido si la informática se basase exclusivamente en los grandes ordenadores: conectividad e interoperabilidad.

Esta necesidad, unido a la proliferación de las redes de área local, fue lo que causó la necesidad de utilizar un protocolo de comunicaciones estándar. Esta necesidad fue cubierta por el protocolo IP (*Internet Protocol*). El protocolo IP sirvió de base para la construcción de otros protocolos como el TCP (*Transmision Control Protocol*), protocolos orientados a conexión que son la base de los sistemas de comunicaciones construidos sobre Internet hoy en día.

La red IP se utilizó durante mucho tiempo como red de comunicación entre ordenadores militares, y posteriormente entre los ordenadores de las universidades norteamericanas, hasta que en 1.993 el acceso a Internet se hizo universal. A partir de este momento, la explosión de la web ha sido la causa de nuevas formas de concebir la computación y la comunicación, así como de nuevas formas de hacer negocio.

Sin duda alguna, el verdadero catalizador de la explosión en el uso de la web fue la aparición del protocolo HTTP (*Hypertext Transfer Protocol*) y el lenguaje HTML (*Hypertext Markup Language*). Ambos, diseñados por el CERN (*Centre Europeen de Recherche Nucleaire*), son las herramientas que permitieron la construcción de las páginas web, y una forma sencilla de publicar y acceder a información a través de Internet.

1.2. Etapas de la web

Cada vez con más frecuencia la palabra Internet es sustituida por el término Web, creando además cierta confusión. La Web (simplificación del término *World Wide Web*) es en realidad un subconjunto de Internet: el que hace referencia al protocolo HTTP y a los documentos creados con lenguaje HTML. Desde que el término Web fue acuñado por primera vez hasta hoy, la web ha sufrido grandes transformaciones. Podemos diferenciar varias etapas en función del uso que se hace de la información publicada:

- Web estática
- Web ligeramente interactiva
- Web fuertemente interactiva
- Web programable

La **web estática** constituye la primera etapa de la web. Después del relativo éxito de formatos como el utilizado por el protocolo Gopher, la

aparición del lenguaje HTML permitió construir páginas ricas en formato, que permitían incluir distintos tipos de letra, colores, imágenes, sonidos, etc., y un elemento muy importante: los *hyperlinks*, enlaces que permitían saltar de una página a otra sin necesidad de que el usuario (navegante) tuviese que conocer dirección alguna.

En esta primera era de la web, las páginas se construían de una manera rudimentaria, con editores de texto, que rápidamente fueron sustituidos por herramientas gráficas que permitieron un diseño más cómodo. Básicamente, la primera era de la web se caracterizaba por el uso y publicación de información estática.

La segunda etapa, **web ligeramente interactiva**, se caracterizó por la posibilidad de añadir a las páginas cierta interactividad que permitía recoger información que los usuarios introducían a través de su navegador. Todo ello a base de *scripts* denominados CGI (*Common Gateway Interface*).

En la tercera etapa, **web fuertemente interactiva**, los *scripts* evolucionaron al poder ser creados utilizando lenguajes como el *Perl*, lo que proporcionaba una mayor potencia y flexibilidad al desarrollo. En esta etapa, las "aplicaciones" web eran algo más complejas, pero su desarrollo era tedioso debido a los sistemas utilizados.

Por último, llegamos a la etapa de la **web programable**. Quizá sea un término demasiado ambicioso, pero, en cualquier caso, lo que se quiere representar es que el desarrollo de las páginas se puede realizar utilizando lenguajes de programación más avanzados que los *scripts*. La aparición de las páginas ASP (*Active Server Pages*), JSP (*JavaServer Pages*), o formatos menos extendidos como PHP (*PHP: Hypertext Preprocessor*) o CFM (*Cold Fusion Markup*), han permitido desarrollar interfaces visuales con aspectos y diseños mucho más amigables y con mayor interactividad que cualquiera de los sistemas anteriores, con la ventaja de poder utilizar lenguajes de programación y herramientas de desarrollo mucho más potentes que un simple editor de textos.

1.3. Usos de la web

La utilización que los usuarios y las empresas han hecho de la web ha estado siempre directamente relacionada con las etapas mencionadas anteriormente. Las universidades y entidades no dedicadas al comercio como actividad principal, adoptaron la web como medio de comunicación en las primeras etapas. La empresa privada entró en la web cuando empezó a apreciar las posibilidades que ofrecía como medio para hacer nuevos negocios o para extender su actividad actual.

Si bien podemos encontrar una cantidad de empresas (relativamente

baja) que accedieron a la web en los primeros momentos, sólo lo hacían como un medio más de llegar a sus clientes de siempre o a nuevos clientes. Es decir, utilizaban la web como un soporte por el que mostrar sus productos, pero no se realizaba ningún tipo de transacción económica. Podría entenderse como un sistema de comunicación o publicidad.

Fue en la cuarta etapa de la web cuando la empresa privada se decidió en masa a hacer negocio por la red. Una vez que el desarrollo de aplicaciones web se estandarizó, y surgieron potentes herramientas de desarrollo que permitieron integrar las aplicaciones web con las aplicaciones de negocio, empezó la era del comercio electrónico. Este tipo de comercio se basó inicialmente en aplicaciones web, que pueden ser vistas como un conjunto de páginas que permiten, por un lado, que el usuario interactúe con la aplicación y, por otro, que las empresas establezcan relaciones con sus clientes dondequiera que se encuentren.

Es la era del B2C [74, 126], es decir, *Business to Consumer*. El término alude inequívocamente a la relación que se crea entre la empresa y sus clientes. Mediante el uso de aplicaciones B2C, las empresas construyen portales de Internet dedicados a sus clientes, gracias a los cuales sus negocios se ven favorecidos por un nuevo canal de acceso al mercado. Además, el B2C extiende las posibilidades de la empresa a, virtualmente, cualquier parte del mundo, gracias al efecto globalizador que representa la Internet.

Como extensión del B2C, se ha trabajado ampliamente en el B2E [135] (*Business to Employee*), término que se refiere a los portales corporativos de recursos humanos. Y también el B2P, el *Business to Partner*, término que apunta a la relación entre empresas y sus proveedores, e íntimamente relacionado con el *e-procurement* [101].

La era B2C, que todavía sigue en expansión, ha dado paso, poco a poco a la era B2B [51, 84, 76, 115, 140]. Si bien hace tiempo que se habla de *Business to Business*, hasta ahora lo único que se ha construido en este sentido son portales de empresa para dar soporte a las relaciones comerciales entre entidades. Dentro del B2B, y desde un punto de vista más técnico, también se trabaja con desarrollos ad-hoc, normalmente basados en técnicas de *web scrapping* (acceder a una página web desde un programa y extraer el contenido a base de analizar y eliminar el código HTML que contiene, recuperando el texto que se busca), o también sistemas de intercambio de información (e incluso de operaciones) a través de HTTP, sin utilizar HTML, es decir, sólo invocación sin formato de presentación. Todas estas técnicas, con mayor o menor éxito, son las que poco a poco han ido fortaleciendo la idea del B2B, y al mismo tiempo, creando nuevas necesidades.

1.4. Computación distribuida

Paralelamente al desarrollo de la web, los diseñadores y fabricantes de lenguajes y entornos de desarrollo han centrado parte de sus esfuerzos en dotar sus herramientas de facilidades que permitan el desarrollo de aplicaciones distribuidas. Dado el avance que suponían las redes locales, dichos fabricantes introdujeron mejoras en sus herramientas para permitir, de forma sencilla, realizar invocaciones desde un programa que se ejecuta en una máquina a otro que se ejecuta en otra distinta.

Primero fueron los lenguajes orientados a objetos. Y después, esos objetos pudieron ser accedidos desde máquinas remotas. Normalmente, los objetos accedidos de forma remota (a través de una red) se denominan componentes.

La razón para todo ello era conseguir los siguientes objetivos:

- **Escalabilidad**, es decir, a medida que la carga de trabajo crece, las aplicaciones pueden crecer gracias a que pueden ejecutarse en más ordenadores al mismo tiempo.
- **Disponibilidad** es una tasa que refleja la relación entre el número de veces que necesitamos utilizar un componente y las veces en que estaba disponible. La mayor disponibilidad de un sistema se alcanza gracias a la distribución de las aplicaciones, utilizando técnicas de reparto de carga (*workload balancing*).
- **Interoperabilidad** es uno de los objetivos más ansiados, y se refiere a la posibilidad de intercomunicar plataformas diferentes (sistema operativo, entorno de ejecución, etc.).
- **Acceso a recursos remotos**. Gracias a los componentes distribuidos, una aplicación que invoca un componente remoto puede, indirectamente, acceder a recursos que no están disponibles localmente. Un ejemplo típico son los componentes especializados en acceso a datos.

DCOM [82] (*Distributed Component Object Model*) y EJB [117] (*Enterprise JavaBeans*) son dos ejemplos antagónicos del éxito de este tipo de arquitecturas, que permiten construir aplicaciones que ofrecen esas características.

DCOM es el modelo de desarrollo distribuido propuesto por Microsoft, según el cual un componente DCOM (versión en red de lo que se denomina un objeto COM (*Component Object Model*) [116]) o ActiveX puede ser manipulado desde una máquina remota, independientemente del lenguaje utilizado, siempre que éste sea un lenguaje de programación creado por

Microsoft (Visual C, VisualBASIC, etc.), o, cuando menos, sea un lenguaje que permita construir programas que corran sobre el sistema operativo Windows (como C Builder de Borland). La arquitectura de aplicaciones distribuidas que propone Microsoft para los desarrollos construidos utilizando DCOM es DNA [68, 99] (*Distributed interNet Applications*). El nombre lo dice todo: aplicaciones distribuidas por la red. DNA es una arquitectura de aplicaciones que se basa en un diseño en 3 capas: capa de presentación, capa de lógica de negocio y capa de acceso a datos.

Por su parte, EJB es un modelo de desarrollo distribuido diseñado para ser ejecutado sobre la plataforma Java, que se compone de un JDK (*Java Development Kit*) y un JRE (*Java Runtime Environment*), que permiten desarrollar y compilar programas el primero, y ejecutarlos en cualquier sistema operativo el segundo. Podemos decir que EJB nos permite invocar componentes que se ejecuten sobre cualquier sistema operativo, pero existe una limitación implícita que no se contempla en la frase anterior: que esos componentes han de ser desarrollados en Java. El modelo de desarrollo EJB se extiende con CORBA (*Component Object Request Broker Architecture*) y el protocolo IIOP (*Internet InterORB Protocol*), que permite que los servidores de componentes se comuniquen entre ellos.

Existen en el mercado algunas empresas dedicadas a proveer mecanismos de interoperabilidad entre ambas plataformas, pero, no dejan de ser meros sistemas de conversión de formato y protocolo, proporcionando únicamente soluciones temporales a un problema de interoperabilidad que debe ser resuelto de otro modo.

1.5. Servicios Web

Por una parte, los avances en tecnologías y desarrollos web, por otra, la evolución en las técnicas de desarrollo distribuido, y por otra, la necesidad de integrar y hacer interoperables las múltiples plataformas, lenguajes y protocolos utilizados actualmente, han sido los tres factores fundamentales que han dado origen a la tecnología de servicios web (*web services*) [112, 61, 125, 121, 129, 44]. Esta tecnología surge del éxito de la familia de lenguajes basados en XML (*eXtensible Markup Language*) [142, 56, 55, 43, 128] y del lenguaje SOAP (*Simple Object Access Protocol*) [94] en particular.

¿Qué es un servicio web? Pues aunque no hay una definición estándar, la que más se aproxima, y que debiera ser tomada como tal, es la que nos ofrece el World Wide Web Consortium (W3C) [32] que, literalmente, dice: *"A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based*

messages conveyed by Internet protocols". Es decir, que un servicio web es un sistema software identificado por un URI (1), cuyos interfaces (2) y enlaces (3) son definidos y descritos utilizando el lenguaje XML. Su definición puede ser encontrada (4) por otros sistemas software, que pueden interactuar (5) con el servicio web de una forma predefinida, utilizando mensajes (6) escritos en XML que viajan utilizando protocolos de Internet (7).

Analicemos la frase punto por punto:

1. Un URI (*Universal Resource Identifier*) es un identificador único que permite localizar cualquier recurso en la red. La idea es muy similar a la de las URL's (*Universal Resource Locator*).

2. Los interfaces de un servicio web representan las distintas formas de invocar al servicio. Cuando nos movemos en términos de computación distribuida el vocablo invocar o invocación son preferidos respecto a llamar o llamada, que son más habituales cuando hablamos de objetos y no de componentes.

3. Los enlaces representan distintas formas de realizar las invocaciones. Los enlaces normalmente hacen referencia a protocolos concretos de Internet: RPC (*Remote Procedure Call*), HTTP/GET, SOAP, etc.

4. Uno de los objetivos fundamentales de las arquitecturas de servicios web es la de crear software a base de componentes que son localizados en la web, sin preocuparnos de quién los hace o cómo funcionan, es decir, un verdadero enfoque de ingeniería de software sobre Internet.

5. El término interactuar (*interact*) hace referencia a tres elementos básicos en el uso de un servicio web: los parámetros de entrada, la invocación, y los parámetros de retorno. Interactuar con un servicio web es invocarlo pasándole unos parámetros y recibiendo otros, que son el resultado de la invocación, además de otros efectos colaterales que el propio servicio web haya podido causar en su ejecución.

6. Los mensajes, que son siempre documentos XML, son normalmente escritos utilizando SOAP, que es un lenguaje basado en XML que permite describir invocaciones y los parámetros asociados a éstas.

7. La referencia a los "protocolos de Internet" que se hace dentro de la definición es muy importante, ya que, indirectamente, está aludiendo al grado de estandarización que se puede alcanzar con la tecnología de servicios web.

El uso de los servicios web es cada vez más común debido fundamentalmente a las facilidades que aportan a la hora de integrar sistemas heterogéneos, unido con la sencillez con la que son utilizados gracias, principalmente, a la utilización de entornos de desarrollo integrados (IDE, *Integrated*

Development Environment) que permiten, desde la misma plataforma, desarrollar aplicaciones de negocio, aplicaciones web o integrar servicios web de terceros.

1.5.1. Construcción de un servicio web.

El elemento fundamental de la tecnología de servicios web es el lenguaje WSDL [63] (*Web Services Definition Language*). Este lenguaje basado en XML se utiliza para definir un servicio web, a partir de una conjunto de elementos:

- Tipo de datos.
- Mensaje.
- Operación.
- Tipo de puerto.
- Enlace (*binding*).
- Puerto.
- Servicio.

Cada uno de estos elementos se representa, en un documento WSDL, dentro de su correspondiente marcador (*tag*) XML, y todos ellos componen la definición de un servicio de la siguiente manera: dentro de un **servicio** web podemos publicar varios **puertos**, que serán vistos por el cliente como métodos del servicio. Las peticiones que lleguen al servicio web serán atendidas en una URL determinada de acuerdo con un **enlace** específico. Cada puerto, independientemente del enlace que utilice, ha de ser descrito previamente como un **tipo de puerto** determinado, que es en realidad un conjunto de **operaciones** elementales. La comunicación entre cliente y servicio se realiza mediante esas operaciones, que han de ser definidas como un conjunto de **mensajes** que, a su vez, son construidos utilizando **tipos de datos** concretos.

Si queremos construir un servicio web, a partir de un módulo de negocio que hemos desarrollado previamente, deberemos realizar dos operaciones fundamentales:

1. Generar el documento que define al servicio que queremos publicar, es decir, el WSDL, y publicarlo.

2. Construir los *wrappers* que envuelvan nuestra lógica de negocio. El término *wrapper* (envoltorio) se refiere a un componente software que permite aislar nuestro software de aplicación de toda la infraestructura de comunicaciones.

Cualquiera de los dos pasos mencionados, se puede realizar de una manera más o menos automática utilizando herramientas, o el mismo IDE con el que hemos desarrollado nuestra lógica de negocio.

1.5.2. Utilización de un servicio web.

Si queremos desarrollar una aplicación que utilice un servicio web determinado deberemos seguir los siguientes pasos:

1. Buscar el servicio (y proveedor) que mejor se adapte a nuestras necesidades. Esto, normalmente, se realiza utilizando un directorio UDDI [26] (*Universal Description, Discovery and Integration*). Dentro de un entorno corporativo es factible la utilización de repositorios privados, cuando los servicios web se utilicen dentro de un sistema EAI [105, 122, 123] (*Enterprise Application Integration*).

2. Integrar el servicio en nuestro IDE. Es decir, nuestro IDE generará los *proxies* y los pondrá a nuestra disposición para que los utilicemos en nuestros programas según nos convenga. Si no disponemos de IDE, lo habitual es utilizar algún generador automático de *proxies*. El término *proxy* no debe ser confundido con el que se refiere a un servidor que realiza funciones de proveedor de acceso y *caching* de Internet. En esta ocasión, el término se refiere a un componente que aísla nuestros programas de la infraestructura de comunicaciones. Es decir, realiza la misma función que el *wrapper* pero en el lado del cliente.

3. Una vez que disponemos de todo lo necesario, sólo tendremos que escribir en nuestros programas las llamadas a los servicios web, como si se tratase de una llamada a un método o una función.

Los usos más habituales de los servicios web hoy en día se dan en el campo de la integración de aplicaciones, EAI, y en el campo del B2B. Es decir, como sistema para alcanzar la interoperabilidad en entornos heterogéneos (EAI), y como medio de estandarizar las comunicaciones (y los negocios electrónicos) entre empresas (B2B).

1.5.3. Características de los servicios web.

Los servicios web presentan ciertas características inherentes a la infraestructura sobre la que han sido diseñados:

- Acoplamiento débil. El nivel de acoplamiento entre cliente y proveedor es muy bajo (menor que el de los componentes y muchísimo menor que el de los objetos), debido a la gran cantidad de capas intermedias que se utilizan y a que aíslan la lógica de negocio de la infraestructura de comunicaciones, tanto en el lado de la aplicación cliente como en el del componente servidor.

- No utiliza estados. La mensajería de los sistemas de servicios web es muy sencilla: mensaje de entrada y mensaje de salida. No se guarda ningún tipo de estado que permita referenciar una invocación anterior.

- Entrega no fiable. Al utilizar HTTP como medio de transporte para los mensajes, no está garantizada la entrega ni la unicidad de los mensajes.

Estas y otras características son las que hacen que la tecnología no sea lo suficientemente madura como cabría esperar de una tecnología de computación distribuida reciente.

Todas estas características vienen dadas porque uno de los objetivos de diseño de la arquitectura de los servicios web es la compatibilidad con la infraestructura de Internet existente.

1.6. Carencias de los servicios web

Aunque los servicios web conforman una tecnología de desarrollo muy prometedora, no constituyen sin embargo una técnica suficientemente madura. Existen diversos factores que hacen que los servicios web no puedan, de momento, ofrecer todo lo necesario para construir aplicaciones web "completas", es decir, carecen todavía de ciertas funcionalidades ya existentes en otras tecnologías, como seguridad en las invocaciones o encriptación. Incluso propiedades tan básicas como el mantenimiento de la transaccionalidad (las cuatro propiedades que forman el ACID [75] (*Atomicity, Consistency, Integrity and Durability*)), implementadas hace tiempo en otras arquitecturas distribuidas, no están todavía disponibles o están en estadios primarios de su desarrollo.

Existen ciertas necesidades que los servicios web no satisfacen. Las más importantes son:

- Transaccionalidad. Poder invocar más de un servicio web dentro de una unidad de trabajo y al finalizar la misma, poder decidir si se desea deshacer los cambios realizados (*rollback*) o darlos por válidos (*commit*) en su conjunto.
- Transacciones de larga duración. Poder invocar un servicio web cuya respuesta será devuelta dentro de un intervalo de tiempo largo y no predecible.
- Seguridad. Distintos aspectos involucran la seguridad dentro de los servicios web. Algunos de ellos son:
  - Fiabilidad de los mensajes (*reliability*): garantizar la entrega de un único mensaje entre cliente y servidor.
  - Confidencialidad: Los mensajes intercambiados entre ambos extremos deben viajar encriptados.
  - Integridad. Los mensajes deben viajar firmados, para evitar que puedan ser modificados por terceros.

Debido a la complejidad que supone construir todos los protocolos adicionales necesarios para completar la tecnología, el consorcio (W3C) ha dividido la arquitectura de los servicios web en dos partes [53]: arquitectura básica y arquitectura extendida. En la arquitectura básica, debemos disponer de los elementos necesarios para:

- Describir servicios web.
- Publicarlos y encontrarlos.
- Invocarlos.

En la arquitectura extendida, debemos disponer de mecanismos que nos permitan mensajería asíncrona, *caching*, enrutamiento de mensajes, mantenimiento de sesiones, etc.

Paralelamente al desarrollo de ambas arquitecturas, otras entidades, fundamentalmente empresas de software (IBM, Microsoft, Ariva, Tibco,...), están continuamente desarrollando complementos y nuevos estándares que les permitan completar las funcionalidades de sus productos. Muchas veces, así ocurrió en el pasado y ocurrirá en el futuro, los estándares desarrollados por alguna o varias de estas entidades son donados al consorcio, para dotarlos de mayor oficialidad.

Estándares como WS-Security [47], WS-Addressing [54] o WS-Transactions [38] están siendo desarrollados e implementados por fabricantes de software. Esto conlleva el riesgo adicional de perder una de las claves del éxito de toda esta tecnología: la utilización de estándares independientes de fabricante.

1.6.1. Necesidades Futuras.

Paralelamente al desarrollo que siga la arquitectura extendida, es necesario trabajar en otros problemas que, aunque no son básicos para el funcionamiento de la tecnología en primera instancia, serán clave en su éxito. Algunos de estos problemas, que todavía han de ser resueltos son:

- Calidad de servicio: Poder disponer de distintos niveles de calidad que nos permitan gestionar las invocaciones de servicios.

- Alta disponibilidad: Para construir aplicaciones de misión crítica (*critical mission*) en las que el rendimiento o la disponibilidad son factores críticos, debemos disponer de mecanismos dentro de la propia tecnología de servicios web que doten a nuestras aplicaciones de esas y otras características.

- Selección de proveedor: Poder cambiar de proveedor dinámicamente no es una necesidad de ejecución, pero sí representa, cada vez con más frecuencia en un mundo de aplicaciones interconectadas, una necesidad de negocio.

- Asincronismo: las aplicaciones cliente deben poder invocar a los servicios web de forma síncrona o asíncrona indistintamente, dependiendo del modelo que más se ajuste a dichas aplicaciones.

- La facilidad para cambiar de proveedor estará determinada por la independencia en la utilización de esquemas. Es decir, será necesario poder cambiar el esquema que sigue la mensajería entre cliente y proveedor para que una misma aplicación pueda dialogar con distintos proveedores (que presumiblemente utilizarán esquemas de mensajería distintos) sin necesidad de ser modificada.

- Composición de servicios: Dados los altos niveles de estandarización de los servicios web, surgirán nuevas necesidades de desarrollo basadas en la creación de flujos (*workflows*) de ejecución de servicios web. De esta forma, se podrán construir bloques de negocio (*business blocks*) a partir de bloques más sencillos, los servicios web.

Este trabajo propone soluciones a estos tipos de problemas, que surgirán tras la puesta en explotación de sistemas críticos que den soporte a aplicaciones basadas en servicios web.

Las soluciones que nosotros aportemos han de cumplir dos restricciones básicas: la compatibilidad y la extensibilidad.

1.7. Estructura de la memoria

Después de esta introducción, el resto de esta memoria está organizada en 6 partes:

1. Introducción, donde, además de la aproximación al problema que se acaba de realizar en este capítulo, se analizará el estado del arte en el área de los servicios web. Este análisis se centrará en estudiar sus carencias en determinados campos. La introducción al problema del que es objeto esta tesis se descompondrá en dos partes. En primer lugar haremos una introducción histórica que servirá para justificar el origen de la tecnología sobre la que vamos a trabajar, los servicios web, y el punto en que nos encontramos hoy en día. En segundo lugar, trataremos de forma individual distintos aspectos de la tecnología de servicios web que deben ser motivo de estudio debido a su temprano estado de desarrollo dentro del conjunto de la tecnología.

2. Virtualización. Se presentarán nuestras propuestas basadas en la virtualización y se analizará como incluirla en la arquitectura actual de los servicios web. Esta parte de la memoria consta de tres capítulos bien diferenciados. En primer lugar se explica la aplicación de las técnicas de virtualización a los servicios web, dando lugar a los nuevos roles de la arquitectura de servicios web. En el segundo de los capítulos se tratarán las peculiaridades de un rol concreto de la nueva arquitectura que se propone: el motor. Y en último lugar, el tercer capítulo de esta parte estudiará el ciclo de vida de los nuevos servicios web virtuales, comparándolos con el ciclo de vida de los servicios web estándar.

3. En la tercera parte se introducirán los servicios web virtuales, así como los lenguajes que hemos desarrollado para darles soporte dentro de nuestra propuesta de arquitectura. Se tratará, en un primer capítulo, el concepto base de servicio web virtual y el mecanismo utilizado para su construcción. En otros dos capítulos se introducirán y estudiarán los dos lenguajes que permiten la implementación de la nueva arquitectura de servicios web virtuales. En primer lugar un lenguaje para la descripción de servicios web virtuales (tiempo de desarrollo), y en segundo lugar el lenguaje para la especificación de atributos de las invocaciones de dichos servicios (tiempo de ejecución).

4. Hemos dedicado una parte independiente del trabajo a estudiar las expresiones y su uso, dada su importancia dentro de la solución propuesta. El mecanismo de expresiones es el que habilita la descripción e implementación de servicios web virtuales en base a servicios web reales.

5. Una vez presentada la base teórica de nuestra propuesta, pasaremos a analizar su aplicabilidad. Ésta será tratada mediante el estudio de las distintas formas de aplicación de nuestra propuesta a cada uno de los problemas en concreto que serán tratados a lo largo de este trabajo. Dichos problemas son: alta disponibilidad, calidad de servicio, mejoras de rendimiento, asincronismo, flexibilidad de los roles del motor, detección y gestión de errores con nuestra arquitectura, gestión de proveedores de servicios web y reducción del nivel de acoplamiento. Adicionalmente, se tratarán, con menor profundidad, otras aplicaciones de nuestra propuesta que, por su complejidad, son consideradas líneas futuras de trabajo a partir de esta tesis doctoral.

6. Por último se presentará un resumen de las conclusiones que han sido presentadas a lo largo del trabajo, así como las líneas futuras de extensión de este trabajo.

CAPÍTULO 2

Estado del arte

2.1. Arquitectura actual

El consorcio de la web (W3C), junto con la IETF (*Internet Engineering Task Force*) [10], son las dos entidades más importantes en la normalización de las tecnologías asociadas a la web, ya que cualquier desarrollo publicado por una empresa privada, por grande que sea ésta, no será un estándar, aunque pueda llegar a serlo si su uso se extiende. Las directrices del consorcio en materia de servicios web pasan por definir dos arquitecturas de implementación y una arquitectura global para la tecnología.

Como hemos mencionado anteriormente, existen dos arquitecturas de implementación: una básica y otra extendida. Esta diferenciación de arquitecturas es, posiblemente, algo temporal, ya que la arquitectura básica ofrece solamente un conjunto de funcionalidades mínimas. Es de esperar que, en el futuro, todas las aplicaciones se construyan contemplando los requerimientos de la arquitectura extendida.

La arquitectura básica se corresponde con los estamentos iniciales de la tecnología de servicios web [61, 121, 112], en donde lo único que se requiere es un protocolo de invocación interoperable. Así, construir servicios web o aplicaciones basadas en servicios web es muy sencillo, ya que los tres únicos requerimientos de la arquitectura son:

- Intercambio de mensajes. Entre el cliente y el proveedor de un servicio debe existir un sistema de mensajería basado en XML que permita intercambiar mensajes.

- Descripción de servicios. Utilizando WSDL debemos poder describir cualquier servicio web para que pueda ser utilizado por un cliente.

- Publicación y descubrimiento de servicios. Si bien es un requerimiento de la arquitectura, no está suficientemente estandarizado y, en cualquier caso, no es necesaria una estandarización cuando se aplica la tecnología a los sistemas de integración de aplicaciones (EAI), que es el uso principal de la arquitectura básica.

Dentro de la arquitectura básica se distinguen **3 únicos roles** que son: el cliente o solicitante (*requestor*) que es la entidad que necesita un servicio; el proveedor (*provider*) que es la entidad que ofrece el servicio; y las agencias o repositorios de servicios, que son entidades en las que se publican y se buscan servicios web. El ciclo de vida de un servicio web, dentro de la arquitectura básica (figura 2.1), es el siguiente:

- El proveedor construye un servicio y genera una descripción para ese servicio.
- El proveedor publica el servicio en una agencia, con la finalidad de ofrecerlo a cualquiera que lo necesite.
- El cliente necesita un servicio y lo busca en una agencia.
- Cuando un cliente ha encontrado un servicio, debe realizar un proceso de enlace (*bind*) para integrarlo en sus aplicaciones.
- Finalmente el cliente interactúa con el servidor realizando invocaciones (que contienen parámetros) y recibiendo respuestas (que contienen resultados o parámetros de retorno).

En cada uno de los procesos que acabamos de describir se utiliza un protocolo o un lenguaje determinado. Entre ellos, los más importantes son el WSDL como lenguaje de descripción de servicios, el SOAP [125, 129] como lenguaje de especificación de invocaciones y el HTTP como protocolo de transporte, que será el que se utilice para hacer progresar los mensajes del cliente al proveedor y viceversa. Pero sin duda, la base de todos los lenguajes utilizados en esta nueva tecnología es el XML.

Como vemos, los requerimientos de la arquitectura básica son muy elementales, constituyendo en realidad, los componentes que darán soporte al resto de funcionalidades. La arquitectura extendida es bastante más compleja que la básica. Engloba la arquitectura básica más una serie de funcionalidades adicionales. Entre ellas, las más importantes son:

- **Mensajería asíncrona**. Posibilidad de enviar y recibir mensajes de forma asíncrona. Al estar basada en protocolos orientados a conexión, la arquitectura básica sólo permite enviar y recibir mensajes de forma síncrona.

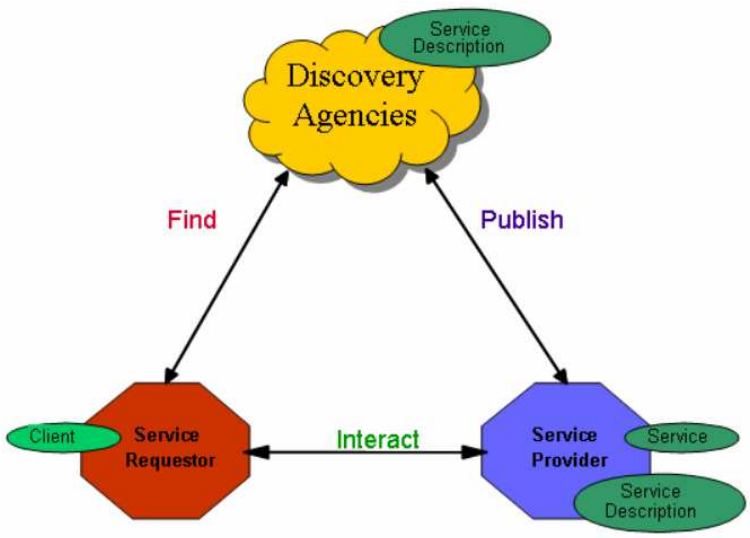

Figura 2.1. Procesos y entidades de la arquitectura actual.

- **Transacciones de larga duración**. De cara al futuro, cuando los servicios web puedan interrelacionarse con procesos manuales o con *workflows* de ejecución de servicios, será necesario disponer de un sistema que permita realizar invocaciones cuyo tiempo de respuesta sea superior al habitual para una conexión sobre la red.

- **Autenticación**. Poder comprobar la identidad del cliente que solicita el servicio.

- **Confidencialidad**. Garantizar que los mensajes que se intercambian entre cliente y proveedor no son leídos por ninguna otra tercera entidad.

- **Integridad**. Los mensajes que viajan entre cliente y proveedor deben tener garantizada su integridad, de tal forma que cada parte pueda determinar si el mensaje ha sufrido modificaciones desde que salió de su origen hasta que alcanzó su destino.

- **Mensajes adjuntos** (*Attachments*). Posibilidad de enviar en los mensajes otros objetos añadidos que puedan viajar unidos a él (firmas digitales, por ejemplo).

Existen muchos más requerimientos, y el tiempo y la evolución de los estándares que vayan apareciendo hará que surjan todavía más, que deberán ser satisfechos para mantener los servicios web como una arquitectura viva y adaptable. En la figura 2.2 se muestra un esquema que representa los requerimientos más críticos de la arquitectura extendida.

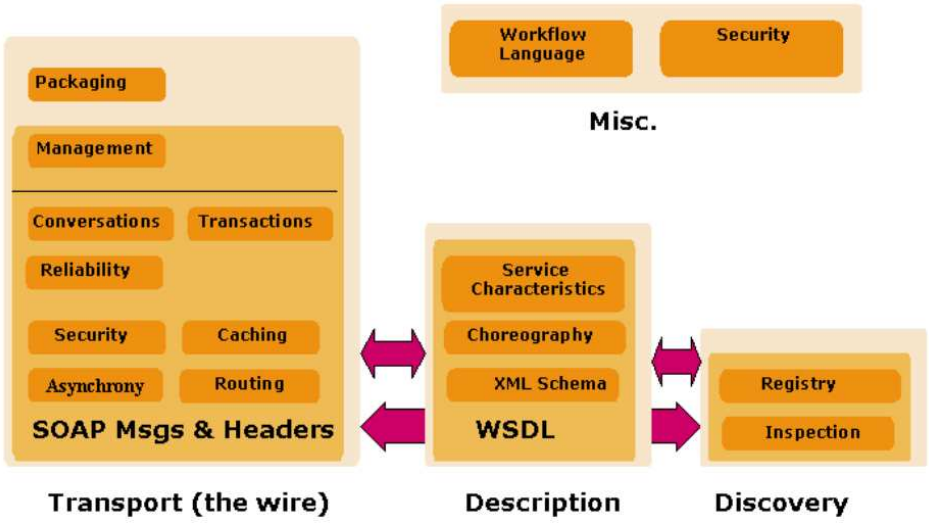

Figura 2.2. Requerimientos de la arquitectura extendida.

Después de haber definido las arquitecturas básica y extendida, el consorcio ha publicado un nuevo documento [53] que contempla una única arquitectura global para los servicios web. En esta nueva arquitectura (ver figura 2.3) se contempla también el papel de los humanos en el proceso de publicación y descubrimiento de servicios web. Si bien, las ideas iniciales giraban en torno al descubrimiento automático, basándose en sistemas de taxonomías [26] y semántica de servicios [108], la lógica lleva a pensar que ese tipo de comportamientos está todavía demasiado lejos y, de momento, determinados procesos han de ser realizados manualmente, sobre todo, aquellos que impliquen firmas de contratos, acuerdos de nivel de servicio o niveles de calidad de servicio.

En la nueva arquitectura propuesta por el consorcio, la separación entre arquitectura básica y extendida pasa a un segundo plano, ya que se centra en definir una pila de lenguajes y protocolos necesarios para completar la tecnología. De acuerdo a esta pila, en el nivel más bajo están los sistemas de comunicaciones, necesarios para dar soporte al sistema de mensajería. En el segundo nivel está la mensajería y todos los protocolos adicionales necesarios para hacer que ésta sea fiable, segura, etc. En el tercer nivel están las descripciones de los servicios, y en el nivel más alto todo lo relacionado con automatización de procesos, agregación, *workflow*, etc.

La arquitectura propuesta (figura 2.4) es una continuidad de la existente, pero contemplando muchas más posibilidades de crecimiento y con un esquema más compacto, que define y utiliza conceptos concisos en consonancia con la arquitectura, es decir, no se basa simplemente en definir lenguajes y protocolos.

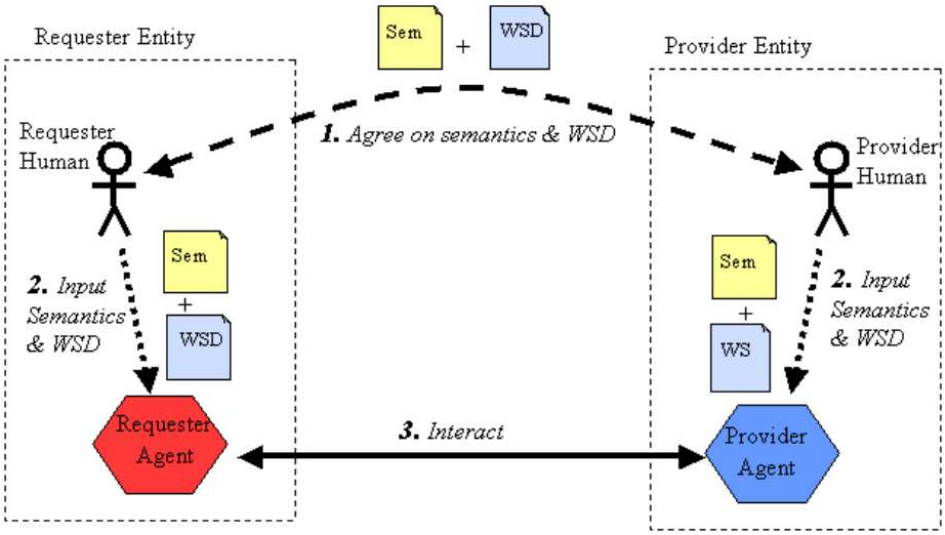

Figura 2.3. Roles en la arquitectura de servicios web.

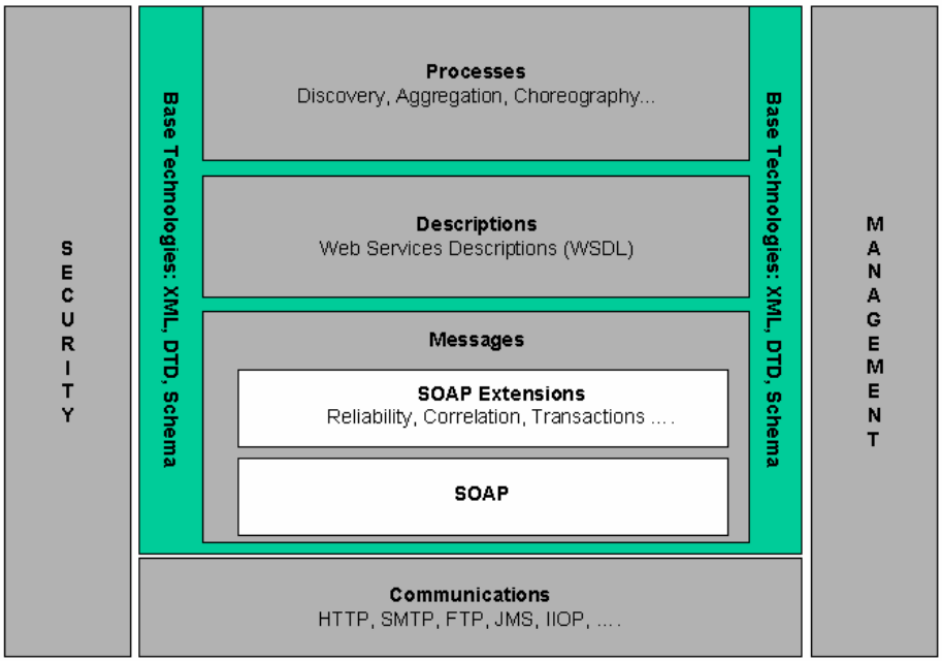

Figura 2.4. Composición de tecnologías en la arquitectura extendida.

En la actualidad hay grupos de investigación, pertenecientes a entidades públicas y privadas, trabajando en los distintos niveles de la arquitectura. Sin duda alguna, los puntos desde los que mayor movimiento se aprecia provienen de la empresa privada. Existen numerosas iniciativas de Microsoft, IBM, WS-I [34] (*Web Services Interoperability*), etc., en todos los niveles de la pila. De hecho, y ya ha ocurrido con anterioridad, las iniciativas de las entidades privadas se traducen, en ocasiones, en donaciones a las entidades públicas. Estándares como SOAP o WSDL, ambos base de la arquitectura de servicios web, fueron en su día iniciativas privadas donadas posteriormente al consorcio (W3C), quien actualmente los administra (versiones nuevas, publicación de documentación...).

Dentro de la nueva arquitectura, como hemos comentado anteriormente, podemos distinguir claramente cuatro niveles en la pila de lenguajes y protocolos:

- Nivel 1. El nivel más bajo se centra en la definición de cómo se ha de adaptar la arquitectura propuesta a los actuales sistemas de comunicaciones basados en IP, como HTTP, SMTP (*Simple Mail Transfer Protocol*) o IIOP.
- Nivel 2. Es el nivel dedicado a la mensajería. A medida que ascendemos en la pila, nos vamos alejando más de las cuestiones más físicas (como la transmisión de los mensajes) y nos centramos más en la arquitectura y en su implementación. En el nivel de mensajería, la arquitectura se preocupa de definir los lenguajes utilizados para escribir los mensajes, que es el método elegido para hacer que las invocaciones y los datos de los clientes lleguen a los servidores o proveedores. En este nivel se utilizan lenguajes como SOAP (existente ya en la arquitectura básica), a la vez que se hacen nuevas propuestas para el manejo de transacciones o entrega fiable de mensajes (garantizar entrega, no duplicidad, etc.)
- Nivel 3. Descripciones. En este nivel la arquitectura se preocupa de definir la forma en que los distintos servicios han de ser descritos, es decir, estamos hablando de WSDL.
- Nivel 4. Procesos. Es el nivel más alto de la pila, y se centra en la definición de estándares que permitan crear nuevos servicios por agregación de servicios existentes más simples. Entran a formar parte de este nivel todos aquellos lenguajes y estándares dedicados a la definición de *workflows*, es decir, cómo componer un proceso de negocio utilizando componentes más sencillos.

Los niveles más bajos están en pleno desarrollo, estando ya disponibles versiones de software de las distintas especificaciones que son aptas para

construir sistemas de explotación. Por ejemplo, HTTP/1.1, con SOAP/1.2 y WSDL/1.1, forman un trío de tecnologías ampliamente extendido y en uso en aplicaciones basadas en servicios web que funcionan en sistemas de producción.

Evidentemente, el consorcio sigue trabajando en los estándares en uso, ya sea adaptándolos a los cambios introducidos en la arquitectura globalmente, o mejorándolos para dotarlos de mayores capacidades. Así, SOAP está siendo completado y ampliado en lo que el consorcio denomina XMLP [35] (*XML Protocol*) mientras que el grupo de trabajo dedicado a servicios web está trabajando en la publicación de la siguiente versión de WSDL, la versión 1.2.

En los niveles más altos de la arquitectura, los estándares no están tan definidos como en los niveles bajos. Aquí, la iniciativa privada supera en agilidad a la iniciativa pública, debido fundamentalmente, a la relación más directa entre la tecnología y el negocio que se observa en los niveles altos de la pila. En estos niveles es donde se puede apreciar una ausencia mucho mayor de trabajos, ya que los pocos que hay están muy orientados hacia iniciativas privadas, principalmente empresas que están desarrollando (y a veces proponiendo a los organismos de normalización) lenguajes que les permitan completar sus productos orientados a la construcción de plataformas de desarrollo y explotación de servicios web.

Todas estas características que no están presentes en la tecnología persiguen un objetivo común: **garantizar la mayor fiabilidad posible en los sistemas de producción**. De esta forma se conseguirá que la tecnología de servicios web sea una tecnología completa, que dé soporte a problemas que ya están solventados hoy en día dentro de otras tecnologías. Mientras las arquitecturas propuestas por el consorcio se centran en definir la implementación, es decir, cómo se ha de plasmar la arquitectura definida en un conjunto de lenguajes y protocolos que permitan utilizar servicios web de forma fiable y segura, nosotros propondremos una nueva arquitectura, compatible con las actuales que permite, además de ofrecer todas las funcionalidades actuales, construir sistemas de producción estables y fiables.

Se pueden detectar ciertas carencias en los trabajos actuales relacionadas con la calidad de servicio, alta disponibilidad, acuerdos de nivel de servicio o la gestión de errores. En las siguientes secciones estudiaremos como se encuentra cada uno de estos temas (y algunos más) en la actualidad.

2.2. Roles actuales

Un aspecto muy importante a tener en cuenta, en lo que a las arquitecturas actuales se refiere, es la separación de roles y la interpretación que

debe hacerse de cada uno de ellos. Aunque es clara la distinción de los roles básicos (directorio, cliente y proveedor) [53], gran parte de la tecnología gira alrededor del servicio web ofrecido por un proveedor. Es decir, desde el punto de vista de los roles de la arquitectura, el proveedor está íntimamente ligado al servicio web. Se manejan en realidad como un único rol, denominado proveedor de servicio (*service provider*). Varios de los problemas de la estructura actual tienen su origen en esta forma de concebir los roles. En la propuesta actual, se considera que la relación entre cliente y proveedor se establece de dos formas:

1. Enlace o *binding*. El enlace es un proceso que se realiza en tiempo de desarrollo, y consiste en adaptar el software del cliente a la definición o descripción de un servicio web.

2. Invocación. La invocación tiene lugar en tiempo de ejecución, y es el proceso mediante el cual, un cliente que está en ejecución invoca a un servicio web.

En la versión revisada de la arquitectura del consorcio, estos procesos se aglutinan bajo una definición todavía menos concisa: interacción. De acuerdo a estas interacciones entre cliente y proveedor, *el binding* se realiza de forma estática, lo que predefine la forma en la que ha de realizarse la invocación. Esto plantea algunas lagunas y problemas no resueltos todavía:

- No podemos realizar *binding* dinámico o, por lo menos, no automáticamente, ya que sería necesario desarrollar módulos específicos dedicados a esta labor para las aplicaciones cliente. Es decir, construir software que realice el *binding* en tiempo de ejecución. Existen opciones basadas en el uso de metadatos [29], que permiten realizar enlace dinámico, pero siempre con la obligación de utilizar metadatos para tratar los datos contenidos en los mensajes recibidos o enviados.

- Una vez que hemos enlazado nuestra aplicación con un servicio web determinado, la ejecución del cliente quedará ligada al proveedor del servicio web que enlazamos inicialmente, viéndonos en la necesidad de enlazar un nuevo servicio si cambiamos de proveedor. Esto significaría tener que desarrollar de nuevo partes del código del cliente para adaptarlas al interfaz propuesto por el nuevo servicio web. Aunque es posible cambiar la ubicación del servidor que provee el servicio, esto en realidad sólo serviría para aquellos proveedores que hayan definido el servicio web exactamente de la misma manera, es decir, los mismos parámetros, los mismos nombres, los mismos tipos de datos, etc.

- Si en una aplicación cliente queremos utilizar más de un proveedor para un mismo servicio nos veremos obligados a enlazar cada servicio web en nuestra aplicación.

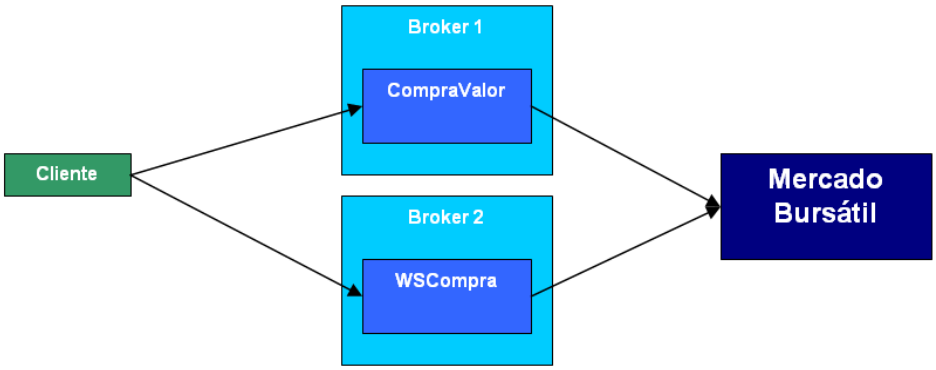

Figura 2.5. Roles de broker.

Supongamos la existencia de una empresa que utiliza en su operativa habitual una aplicación software que ha sido desarrollada dentro de la propia empresa. Supongamos ahora que la empresa necesita incorporar a sus aplicaciones la posibilidad de operar en bolsa. Para ello, busca en Internet un *broker* que disponga de un servicio web que le permita realizar operaciones bursátiles directamente desde sus aplicaciones. Una vez firmados los contratos necesarios, el *broker* proporciona a la empresa un documento WSDL con la descripción de un servicio web denominado "ComprarValor". Su funcionamiento es el siguiente: cada vez que el servicio web es invocado, el *broker* envía una orden de compra a un mercado bursátil.

La empresa de software incorpora el WSDL a su proceso de desarrollo, realiza el *binding* y, a continuación, incorpora el servicio de compra de valores a sus aplicaciones.

Múltiples motivos pueden hacer que la empresa decida incorporar un segundo *broker* a sus aplicaciones (figura 2.5), por ejemplo: coste de las comisiones, disponibilidad, tiempo de respuesta, etc. Para ello, tendrá que volver a realizar un *binding*, agregar el nuevo servicio web y desarrollar de nuevo las partes del código de la aplicación necesarias para dar soporte al nuevo servicio web.

De esta forma, cualquier cambio de proveedor implicará modificaciones en la aplicación. Asimismo, determinados cambios en el servicio web implicarán también modificaciones en las aplicaciones. Secundariamente, el hecho de utilizar más de un proveedor obliga a los desarrolladores a escribir código adicional en los programas para gestionar las invocaciones. Éste código será el responsable de realizar la elección de proveedor, de realizar el tratamiento de errores o de reintentar las invocaciones en caso de error.

Gran parte de los problemas que acabamos de describir provienen del hecho de mezclar el rol de proveedor de servicio con el de proveedor de ser-

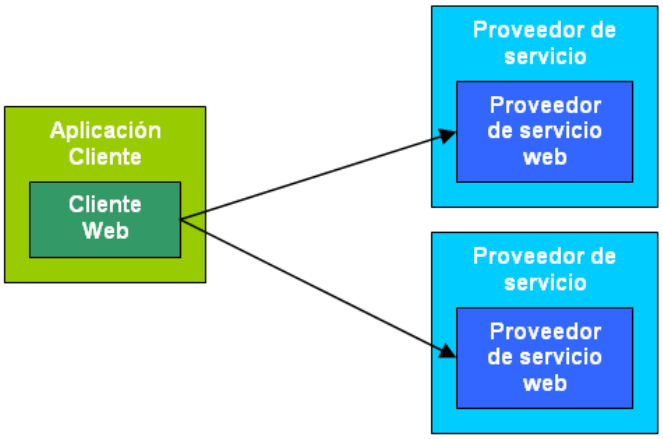

Figura 2.6. Proveedores de servicio y proveedores de servicio web.

vicio web. Cuando hablamos de arquitecturas orientadas a servicios, SOA [49, 138] (*Service Oriented Architecture*), normalmente hablamos de clientes y de proveedores, siempre relacionándolos con servicios. Cuando implementamos un sistema basado en SOA utilizando servicios web, tendremos que cambiar la terminología y pasar a utilizar los términos "cliente web" y "proveedor de servicio web" (figura 2.6).

En la arquitectura revisada del consorcio se hace patente una diferencia entre las entidades y los agentes. En la parte cliente se utilizan los términos "*Requester Entity*", "*Requester Human*" y "*Requester Agent*" para designar tres componentes: la entidad cliente, el humano que negocia el servicio y el agente que lo ejecuta (véase figura 2.3). La entidad cliente es una empresa que necesita un servicio. El "humano solicitante" es la persona encargada de buscar y negociar con los proveedores las características del servicio que está buscando. Y el "agente solicitante" es el componente encargado de solicitar la ejecución de un servicio. En la parte del proveedor encontramos otros tres componentes parejos: "*Provider Entity*", "*Provider Human*", "*Provider Agent*", que son, respectivamente, la empresa, el humano y el agente que, conjuntamente, forman lo que denominamos comúnmente proveedor.

A pesar de diferenciar tres componentes dentro del proveedor, la asimilación de roles entre el "agente proveedor" y la "entidad proveedora" permanece.

En definitiva, la verdadera confusión, y causa final de todos estos problemas que acabamos de describir, tiene su origen en la no separación entre el rol de "proveedor de servicio web" del de "proveedor de servicio", que son dos conceptos similares, pero que aluden a términos bien distintos. Esta **igualación de roles** se percibe inicialmente desde un punto de vista arquitectural, en la definición de los componentes de la arquitectura. Pero dicha

equiparación trasciende el plano arquitectónico y se acaba plasmando en la estructura y en el funcionamiento de las aplicaciones desarrolladas bajo el paradigma de los servicios web.

Dentro de la arquitectura de servicios web actual, cuando se utiliza el término proveedor o proveedor de servicio, realmente se está haciendo referencia al "proveedor de servicio web".

Nosotros propondremos la creación de dos nuevos roles en la arquitectura: el proveedor de servicio y la entidad cliente. La introducción de estos dos nuevos roles nos conducirá a la necesaria separación, con la correspondiente asignación de funciones, de los roles de proveedor de servicio y proveedor de servicio web.

2.3. Modelos de invocación

En la actual arquitectura de servicios web, el modelo de invocación utilizado es el que podríamos denominar "**invocación directa**". Es decir, el cliente realiza una invocación directamente hacia el proveedor, sin ningún elemento intermedio más que los componentes que forman la infraestructura de comunicaciones.

Este modo de trabajar presenta algunos problemas, debido a la dependencia que se crea entre cliente y proveedor:

1. **Disponibilidad limitada**. La disponibilidad de la aplicación cliente dependerá, al 100 %, de la disponibilidad que pueda ofrecer el proveedor del servicio.

2. **Gestión de la calidad de servicio y/o acuerdos de nivel de servicio**. Dichos controles deben ser realizados por el cliente, por el proveedor o por ambos para garantizar el cumplimiento de los niveles preestablecidos.

3. **Enrutamiento de peticiones**. Al ser directa la relación entre ambos extremos, las peticiones no pueden ser encaminadas a otros puntos que puedan dar el mismo servicio. Esto está directamente relacionado con el proceso de *binding* estático que se realiza cuando se desarrollan las aplicaciones.

Soluciones a este tipo de problemas pueden obtenerse utilizando un elemento intermedio, es decir, un componente nuevo en la arquitectura que permita recibir peticiones de invocación y redirigirlas a los proveedores. De esta forma estaremos ante un modelo de "**invocación indirecta**", donde

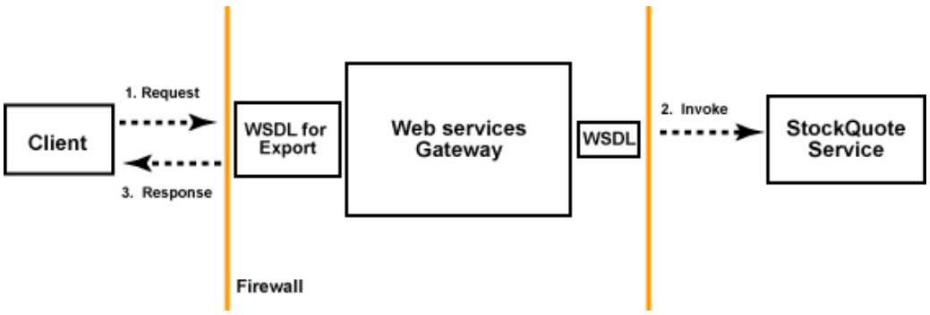

Figura 2.7. WSG como firewall.

las peticiones no se realizan directamente del cliente al proveedor, sino a través de ese *nodo intermedio*.

El nodo intermedio es el responsable de recibir las peticiones, analizarlas, localizar el proveedor y redirigirlas convenientemente según unas reglas predefinidas. Dependiendo de la habilidad del nodo intermedio para manejar las peticiones, podremos construir sistemas más o menos "inteligentes" que nos permitirán, entre otras cosas, mejorar la disponibilidad u obtener un sistema de calidad de servicio que podrá ser gestionado desde el cliente, desde el nodo intermedio o desde el proveedor.

Actualmente existen en el mercado varias soluciones que utilizan esta arquitectura. Entre ellas cabe citar *Web Services Gateway* (WSG) [137], desarrollado por IBM, que actualmente se distribuye como un componente del *WebSphere Application Server* (WAS) [31] (también de IBM), que es un servidor de aplicaciones basado en Java.

El WSG permite redirigir las peticiones de invocación recibidas en la pasarela (*gateway*) hacia servicios web implementados "al otro lado de la red", así como redirigir las peticiones de invocación generadas en una Intranet hacia Internet. Es decir, actúa como un *proxy* de servicios web. Esta vez la palabra *proxy* debe ser entendida de modo similar a como se entiende un "*proxy* de web". Un sistema de este tipo aísla la red interna de la externa, con el fin de proteger la red de ataques externos indeseados y de facilitar el acceso al exterior a través de un único punto.

Dos tipos básicos de peticiones se resuelven en el WSG. Por un lado, las peticiones entrantes (figura 2.7), es decir, peticiones de fuera de la red (parte externa del *firewall*), que son recibidas y enrutadas a algún servidor de la red privada (Intranet). En cierto modo, actúa como un "*proxy* inverso", al igual que los utilizados en tecnología de aplicaciones web estándar. Por otra parte, el WSG puede actuar como un servidor *proxy* de salida, es decir, las peticiones llegan de alguna aplicación cliente del interior de la red y son enrutadas al exterior, atravesando el *firewall*. El esquema de las peticiones

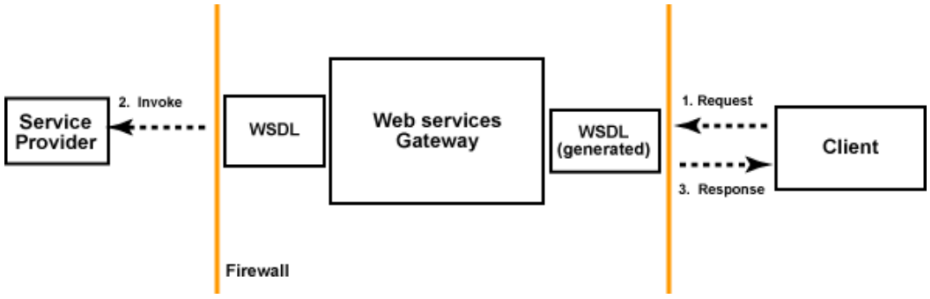

Figura 2.8. WSG como *proxy*.

salientes se muestra en la figura 2.8.

El WSG hace algunas cosas más, como independizar el protocolo utilizado por el cliente del utilizado por el proveedor. Por ejemplo, podemos poner en un servidor de la red interna un servicio web que sea invocado utilizando protocolo HTTP/POST, mientras los clientes hacen las invocaciones utilizando el protocolo SOAP sobre HTTP. Para poder hacer estos cambios de protocolo, y para gestionar el enrutamiento de peticiones correctamente, el WSG maneja, para cada servicio web utilizado, dos documentos WSDL: uno que es el que se corresponde con el servicio web real y otro ficticio que es el que se publica o exporta en el WSG.

Aunque el WSG es un buen sistema para gestionar los múltiples protocolos web que existen hoy en día, así como para conseguir una protección efectiva de los servidores en los que se ejecutan servicios web, no aporta funcionalidades adicionales fácilmente implementables en el nodo intermedio. De hecho, los servicios web reales deben ser publicados tal cual en el WSG, solamente aplicando al documento WSDL cambios relativos a la especificación de protocolo o a la localización del servidor.

Otro ejemplo de enrutador lo encontramos en *EntireX XML Mediator* [7]. En XML Mediator, Software AG [21] (la empresa que lo desarrolla y distribuye) hace especial énfasis en el manejo del XML (de los datos al fin y al cabo), preocupándose en menor medida por la implementación. De acuerdo a la arquitectura propuesta (figura 2.9), dentro de la gama de productos de EntireX, el XML Mediator actúa como pasarela recibiendo peticiones SOAP que después de analizar enruta al componente encargado de resolver la petición.

Además de enrutar peticiones (que podrán estar o no escritas en SOAP), XML Mediator permite realizar cambios en los datos para que el componente que los reciba los pueda tratar tal y como espera recibirlos, independizando de esta forma la estructura de los datos que manejan los clientes y la estructura de los datos que manejan los proveedores.

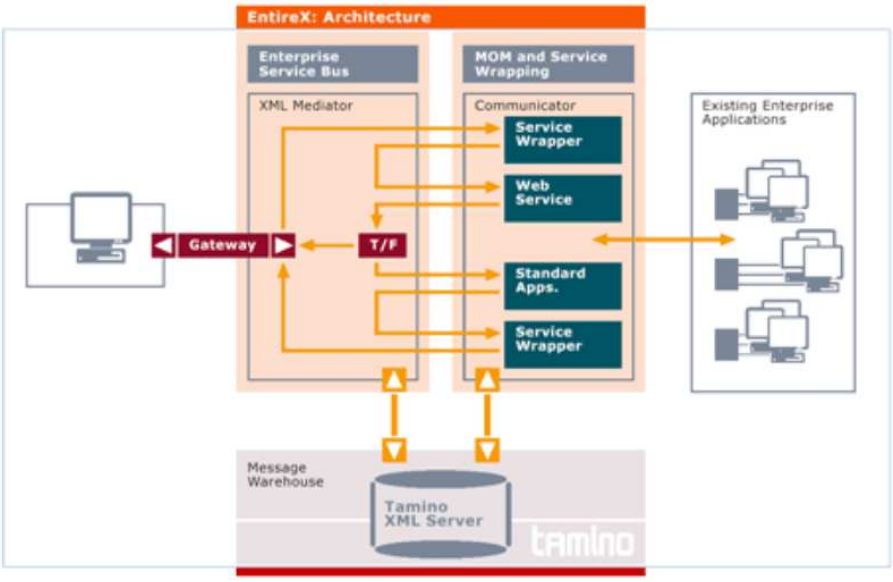

Figura 2.9. Arquitectura de EntireX.

Otros productos como el WS-DBC [36] (*Web Services Domain Boundary Controller*) se centran especialmente en temas de seguridad. El WS-DBC se comporta como un *firewall* específicamente diseñado para el manejo de mensajes SOAP, de tal forma que cuando queramos desarrollar un servicio web no tengamos que preocuparnos por el entorno de seguridad. Todos los aspectos relacionados con ella serán manejados por el WS-DBC. De esta forma, situando el *firewall* de servicios web en la DMZ (*DeMilitarized Zone*), conseguimos que toda nuestra lógica de negocio se ejecute en un entorno seguro (en la Intranet), mientras el *firewall* se encarga de gestionar aspectos como: autorización, autenticación y auditoria. El WS-DBC (figura 2.10) actúa como un enrutador de mensajes SOAP (tanto entrantes como salientes), gestionando la seguridad de los mensajes entrantes, y reenviando los mensajes a los servidores encargados de procesar las peticiones. El WS-DBC también puede actuar como *proxy* de servicios web, es decir, los mensajes SOAP originados en una Intranet son reenviados hacia Internet a través del *firewall* de servicios web de la DMZ.

A la hora de manejar mensajes SOAP cifrados o firmados, el WS-DBC puede utilizar varios estándares de seguridad actuales, como el WS-Security [47], SAML [19, 66] (*Security Assertion Markup Language*), o *XML Digital Signature* [50].

Como acabamos de comentar, distintos problemas deben ser resueltos de distintas formas dependiendo de la solución que apliquemos. Por ejemplo, podemos solventar problemas de seguridad mediante el uso de un nodo

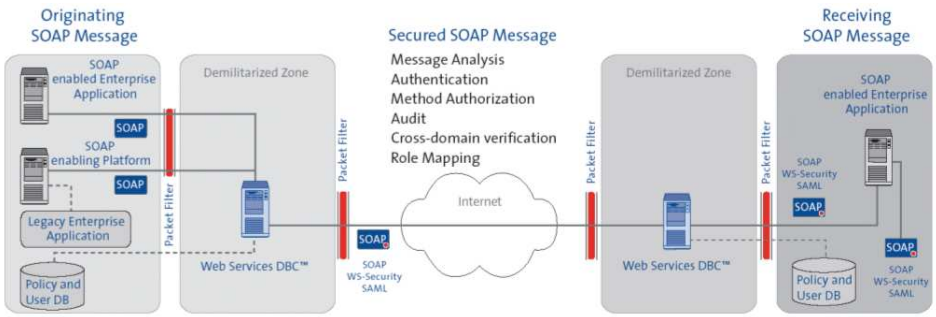

Figura 2.10. Estructura de WSDBC.

fronterizo (como en WS-DBC), o problemas de acoplamiento entre los esquemas seguidos por los clientes y los seguidos por los proveedores (como en WebSphere o en XMLMediator). Ambos problemas son resueltos por distintos proveedores de software de manera conceptualmente similar: introduciendo un componente intermedio encargado de realizar las funciones adecuadas a la resolución de cada tipo de problema.

El componente intermedio cambia el modelo de invocación de directo a indirecto. **Nuestra propuesta se basa en el uso del modelo de invocación indirecto** para ofrecer soluciones a todos los problemas planteados en esta sección, no solamente los de seguridad o acoplamiento. Sin duda, las soluciones ofrecidas comercialmente son buenas, pero plantean un problema grave: son propietarias. Cada fabricante resuelve un subconjunto de los problemas utilizando su propia tecnología, y esto no es válido en un mundo de estándares abiertos, ya que las soluciones deben hacer uso de un sistema estandarizado.

Nuestra propuesta para solventar los tipos de problemas que acabamos de comentar, se basa en utilizar un lenguaje estándar, abierto, que permite implementar el modelo de invocación indirecto. De esta forma, estamos en disposición de implementar técnicas de aislamiento como las utilizadas en WS-DBC o WebSphere, o técnicas de manejo de mensajes como las utilizadas en XMLMediator.

2.4. Control y gestión de Errores

El ciclo de vida habitual de una aplicación cliente que utiliza servicios web implica, entre otros, los siguientes procesos:

1. Búsqueda de proveedor y servicio.
2. Localización de la descripción del servicio (WSDL).

3. Realización de un *binding* del servicio, construyendo los *proxies* que van a permitir las invocaciones.

El proceso de enlace (*binding*) implica, normalmente, ejecutar un generador de código que construirá un conjunto de módulos (normalmente clases) en las que se definen:

- La clase de los datos de entrada a cada método que hayamos enlazado.
- La clase de los datos de salida, es decir, el objeto que devuelve un método después de la invocación.
- Los métodos invocables, con sus parámetros de entrada y sus parámetros de salida, que han de corresponder, necesariamente, con las clases mencionadas previamente.

Una vez que se han generado las clases *proxy*, en los programas de la aplicación cliente se podrá definir los objetos correspondientes a las clases que representan el servicio web dentro del entorno de desarrollo local (ver ejemplo 2.1), incluyendo la definición de las clases con los datos de entrada y salida.

Cuando una aplicación cliente realiza una invocación a un servicio web, debe controlar y manejar cualquier situación de error que se produzca. Para ello, la siguiente lista de comprobación debería ser implementada para cada invocación:

1. Invocar servicio.
2. Comprobar error obtenido.
3. Si se ha producido un error dictaminar si se puede reintentar la invocación.
4. Reintentar la invocación un número finito de veces.
5. Si no se ha logrado realizar la invocación con éxito, devolver un error al nivel lógico superior, o enviar un mensaje de error al usuario si la aplicación se está ejecutando en el nivel más alto.

Si con el tiempo decidimos dejar de utilizar un proveedor (porque tiene una disponibilidad muy baja, costes elevados, o la razón que sea), tendremos que "retocar" la aplicación para eliminar las referencias al proveedor actual y añadir las referencias al nuevo. El proceso, aunque no es excesivamente costoso, exige realizar cierto mantenimiento en las aplicaciones. La complejidad de dicho proceso dependerá en gran medida de la arquitectura

```
public class sample {
  dclbapc1.TSDCR049 ws = new dclbapc1.TSDCR049();
  dclbapc1.TSDCR049IN wsIn = new dclbapc1.TSDCR049IN();
  dclbapc1.TSDCR049OUT wsOut = new dclbapc1.TSDCR049OUT();

  wsIn.TRANSACCION="TC49";                        // Transacción
  wsIn.DNI="35307304";                            // DNI
  ws.Url="http://dclbapc/tsdc001/invoke.sif";     // URL servidor

  try {
    wsOut=ws.call(wsIn);
    TextBox1.Text="El saldo es "+Convert.ToString (wsOut.SALDO);
  }
  catch (Exception ex) {
    TextBox1.Text="Se ha producido un error: "+ex.Message;

  }
}


public class TSDCR049IN {
  public string TRANSACCION;
  public string DNI;
  public long CLAVE;
}


public class TSDCR049OUT {
  public float SALDO;
}
```

Ejemplo 2.1: Aplicación cliente que invoca al servicio TSDCR049.

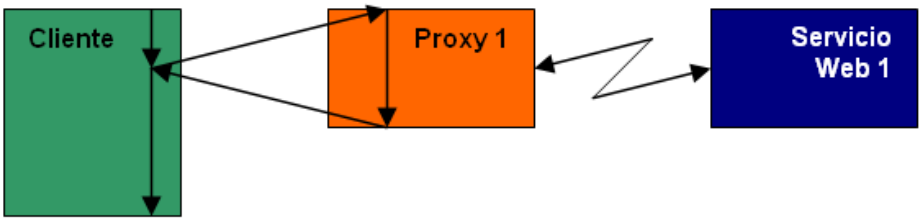

Figura 2.11. Invocación estándar con *proxy*.

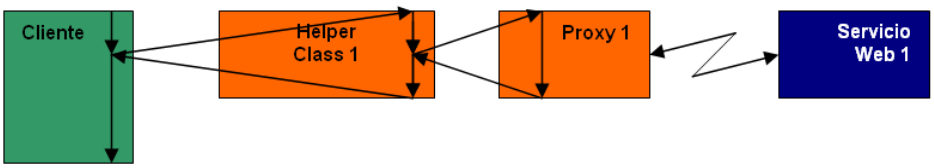

Figura 2.12. Invocación con *proxy* y clases de ayuda.

de desarrollo que hayamos utilizado en nuestras aplicaciones. Por ejemplo, si nos hemos preocupado de incluir las llamadas a los servicios web dentro de clases propias que reciban y devuelvan estructuras de datos definidas en nuestra aplicación, bastará con cambiar el contenido de esas clases. Si por el contrario, nuestras aplicaciones utilizan directamente las clases *proxy* (figura 2.11), podemos encontrarnos con algunos trabajos adicionales a la hora de adaptar la aplicación.

Si hemos utilizado una "Helper Class" (clases de ayuda intermedias que no forman parte de la lógica de negocio, sino que simplemente facilitan de alguna forma el desarrollo), cuando se cambie el proveedor, sólo habrá que retocar esta clase, para que reciba los nuevos parámetros, y los entregue a la aplicación cliente tal y como lo hacía con el servicio web utilizado anteriormente (figura 2.12).

Si no hemos utilizado una clase de ayuda, nos encontraremos con que el *proxy* generado para el nuevo servicio tendrá otros nombres de variables y/o utilizará otros tipos de datos distintos de los utilizados por el primer servicio web, con lo que tendríamos que adaptar la lógica de negocio.

Es habitual disponer de más de un proveedor para realizar un servicio, de tal forma que si uno falla, se pueda reintentar la ejecución del servicio con otro proveedor. Esto ocurre así en el mundo real, y de la misma forma debería poder ser plasmado en el software cuando desarrollamos aplicaciones siguiendo los paradigmas de las arquitecturas SOA.

Así pues, con el objetivo de aumentar la disponibilidad de una aplicación, o simplemente de dotarla de una mayor flexibilidad, podemos optar por añadir más proveedores a la aplicación cliente, de tal forma que se rein-

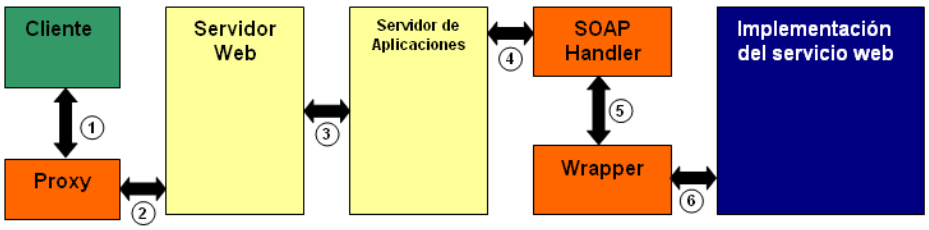

Figura 2.13. Puntos de fallo en la arquitectura estándar.

tente la ejecución del servicio con otro proveedor cuando se detecte que ha fallado una invocación. Siguiendo con nuestro ejemplo, deberíamos añadir a nuestra aplicación cliente un segundo servicio web. Para ello, realizaremos la búsqueda, después el *binding* y, finalmente, adaptaremos el código fuente de nuestra aplicación para que contemple el nuevo servicio web. Una vez hecho esto, la lista de acciones a tomar para realizar la gestión de errores debería ser similar a la siguiente:

1. Invocar servicio.
2. Comprobar error obtenido.
3. Si se ha producido un error dictaminar si se puede reintentar la invocación.
4. Reintentar la invocación un número finito de veces.
5. Si disponemos de más proveedores repetir el proceso para el siguiente proveedor.
6. Si no se ha logrado realizar la invocación con éxito, devolver un error al nivel superior, o mostrar un mensaje de error al usuario si la aplicación se está ejecutando en el nivel más alto.

Para disponer de una forma ágil, sencilla y eficaz de manejo de errores, los procesos de control y gestión de errores deben ir asociados a la forma en la que éstos se producen y a la forma en la que se propagan. En cada punto intermedio de la arquitectura debe realizarse una gestión de errores, ya que en los servicios web los errores que se pueden producir en cualquier punto de una invocación son automáticamente propagados hacia el cliente.

En la figura 2.13 podemos ver, esquemáticamente, una implementación de la arquitectura de servicios web. En cada punto numerado encontramos un punto de fallo. El punto ① es el único que pueden manejar las aplicaciones cliente, normalmente a través del uso de excepciones. Dentro del punto ①, la aplicación cliente realiza una llamada al *proxy* y espera un resultado o

una excepción. Si ésta se produce, la aplicación cliente tendrá que proveer los mecanismos adecuados para reintentar la invocación, enviar un error al usuario de la aplicación o propagar la excepción.

En el punto ② los errores se pueden producir por problemas de red, ya que la comunicación entre *proxy* y servidor web se realiza a través de una red IP. Típicamente, problemas físicos de red, problemas de rutas, o problemas de más alto nivel, producidos por errores en el protocolo HTTP (falta de autorización, problemas del servidor, configuración incorrecta del servidor web,...).

En los puntos ③ y ④, donde no hay intervención de software desarrollado por el cliente ni por el proveedor, los errores que se pueden producir se deben típicamente a problemas de configuración o de disponibilidad.

En el punto ⑤, el manejador de peticiones SOAP invocará la clase *wrapper*, que extraerá el mensaje (proceso de deserialización) e invocará al componente encargado de ejecutar la lógica de negocio (esto se hace en el punto ⑥). En el punto ⑤ los problemas se suelen producir por problemas de disponibilidad de la aplicación del proveedor del servicio. Y en el punto ⑥, es donde más posibilidades de error encontramos, ya que depende enormemente de la calidad del software desarrollado para la implementación del servicio. Ello es debido a que, típicamente, el desarrollo de la lógica de negocio incluye varios componentes distribuidos, como servidores de bases de datos, servidores de ficheros o de aplicaciones.

La aplicación cliente debe gestionar todos estos tipos de errores y necesita distinguir unos de otros. Para ello, dispone habitualmente de un único mecanismo: el manejo de excepciones. Mediante este sistema, el *proxy* lanzará una excepción para indicar los errores producidos en los puntos ①, ②, ③ y ④. Los errores producidos en la parte del servidor no pueden ser gestionados directamente. En su lugar, el lenguaje SOAP permite incluir errores mediante el uso del elemento "fault" y los sub-elementos "faultcode" y "faultstring". Cuando un mensaje SOAP incorpora una excepción, el *proxy* del cliente deberá deserializarlo y generar la excepción correspondiente para que pueda ser capturada por la aplicación cliente. Cuando una aplicación cliente ha detectado un error capturando una excepción, deberá actuar en consecuencia: reintentando la invocación, enviando un mensaje de error o propagando la excepción.

El uso de excepciones, cada vez más generalizado, lo encontramos fundamentalmente en lenguajes orientados a objetos, como C++, Java o C#. En otros lenguajes, habrá que utilizar los mecanismos provistos por el propio lenguaje para gestionar los errores convenientemente.

Típicamente, la estructura de programación "*try-catch-finally*", original del C++, es la forma más utilizada de capturar excepciones. En los ejem-

plos de la figura 2.1, se realiza una invocación dentro del bloque "try", y se capturan las posibles excepciones dentro del bloque "catch", donde son procesadas. Este sistema será el adecuado para programar los sucesivos reintentos de invocación sobre el mismo u otro servicio web, siempre que nos veamos obligados a programar la lógica de control de errores dentro de las aplicaciones cliente.

Como vemos, **la complejidad en el desarrollo de software aumenta a medida que necesitamos añadir más lógica a los programas**. Si optamos por mejorar la disponibilidad de las aplicaciones cliente mediante la introducción de código específico de control de errores y manejo de más de un proveedor, tendremos que enlazar más de un servicio web y desarrollar lógica dentro de los programas para gestionar las invocaciones. En definitiva, los procesos de desarrollo y de mantenimiento se ven afectados, resultando más difíciles de llevar a cabo.

Utilizando nuestra propuesta de lenguaje abierto único, podremos construir servicios web cuya implementación se basa en la utilización de varios proveedores. Esto permitirá reducir la indisponibilidad de las aplicaciones cliente sin complicar los procesos de desarrollo.

2.5. Gestión de proveedores

A medida que la tecnología de servicios web se vaya estandarizando, será adoptada cada vez por un mayor número de proveedores. Es decir, las entidades clientes dispondrán de más oportunidades para construir sus aplicaciones, ya que distintos proveedores ofrecerán los mismos servicios web para resolver el mismo problema, pero diferenciándose en el coste, la velocidad, la disponibilidad, etc.

Esta facilidad para encontrar proveedores puede suponer una complicación adicional para los desarrolladores de aplicaciones cliente, ya que cada servicio web utilizará sus propios parámetros y sus propios esquemas de mensajería. Esto obliga a realizar, como ocurría con la gestión de errores, desarrollos ad-hoc para cada proveedor que se desee utilizar.

Por tanto, para que las entidades cliente se beneficien del abaratamiento de los servicios, o de su mejora de calidad, deberán abordar procesos de desarrollo más costosos.

En la actualidad, los mecanismos que utilizan las empresas para ser ágiles en el desarrollo de aplicaciones, y poder adaptarse rápidamente a los cambios, se basan fundamentalmente en la utilización de motores de flujo, de motores de ejecución de procesos que independicen la aplicaciones cliente de las proveedores. Herramientas como BizTalk [12], WebMethods Enterprise Services Platform [30], Gentran Integration Suite [9] o WebSphe-

re Integration Edition permiten definir interfaces que exponen la lógica de negocio para ser utilizada por las aplicaciones corporativas. Dicha lógica es implementada utilizando distintos mecanismos, como agregación, desarrollo de componentes y/o clases, etc. Es decir, distintas formas de implementar para publicar, finalmente, un servicio web.

Esta forma de trabajar permite a los desarrolladores mantener el interfaz de sus aplicaciones y modificar la implementación, donde pueden además utilizar otros proveedores de servicios. Así, la agregación de servicios se reducirá, dependiendo del mecanismo de implementación utilizado, a desarrollar adaptadores que dialoguen con los nuevos proveedores.

No cabe duda de que esta forma de trabajar, a parte de no estar basada en el uso de estándares, simplemente traslada el trabajo de adaptación desde las aplicaciones cliente a los componentes intermedios.

La utilización de múltiples proveedores ha sido tratada en [95]. Si bien este trabajo no se preocupa especialmente por la implementación, incluye una buena disección de las distintas formas en las que se puede construir servicios web multi-agente. Distintas formas de selección deberán ser tenidas en cuenta dependiendo de si se selecciona el agente antes de la ejecución de un servicio (preproceso) o si se hace después de obtener los resultados (postproceso). Entre ellas, los sistemas de voto o la selección del resultado más rápido obtenido suelen ser los más comunes.

En [57] se propone una arquitectura de implementación de un sistema multi-proveedor. Dicha arquitectura se basa en la utilización de dos tecnologías diferentes pero complementarias. Por un lado, los servicios web que se exponen para su uso por entidades cliente son construidos utilizando BPEL4WS (*Business Process Execution Language for Web Services*) [69, 110], es decir, se utiliza agregación para construir servicios web. Por otra parte, la implementación de cada uno de los servicios web involucrados en la ejecución del *workflow* especificado en el documento BPEL4WS se realiza utilizando técnicas de sistemas multi-agente (*Multiagent Systems*). Para integrar todo, se propone envolver (*wrap*) los servicios web llamados desde el *workflow* dentro de agentes proactivos. Es decir, la interacción entre clientes y *workflow* se consigue mediante el uso de tecnología de servicios web, mientras que la integración entre motor de ejecución de *workflow* y proveedores se realiza utilizando tecnología de sistemas multi-agente.

Si bien, la solución planteada es factible (de hecho ha sido implementada), deja abiertos muchos puntos de la arquitectura propuesta para ser implementados de diferentes formas, sobre todo en lo que se refiere a integración entre motor y proveedores. Es decir, es una buena experiencia de implementación de BPEL4WS, pero no es posible estandarizarla, ya que en realidad es una implementación ad-hoc.

Una forma posible de implementar sistemas multi-agente se basa en la utilización de un intermediario, que será el responsable de seleccionar e invocar al agente apropiado. Alternativamente, se puede implementar un sistema multi-agente dotando a los consumidores de servicios web (las aplicaciones cliente) de la lógica necesaria para realizar la invocación y la selección. Una tercera opción consiste en publicar servicios web que sean los responsables de realizar las invocaciones multi-agente y exponer su lógica como un único servicio.

Nuestra propuesta en esta área de utilización de servicios web, desde la óptica de un sistema multi-agente, se basa en la utilización de un intermediario que actúa como agente único ante las entidades cliente, y realiza las actividades relacionadas con la implementación multi-agente de forma transparente para los clientes.

Además, nuestra idea de implementación de sistema multi-agente se basa siempre en la utilización de servicios web, es decir, los clientes se comunican con los intermediarios utilizando tecnología de servicios web; pero además, el intermediario se comunica con los múltiples agentes utilizando la misma tecnología.

2.6. Alta disponibilidad

Sin duda alguna, al hablar de sistemas de producción, uno de los temas que más preocupa es la disponibilidad del sistema. Cuando implementamos aplicaciones de misión crítica queremos que la disponibilidad y la fiabilidad de nuestro sistema sean lo mayor posible. Alcanzar la máxima disponibilidad no es algo trivial, y debería ser considerado como una meta deseable, aunque no sea alcanzable en su totalidad. Es decir, el coste asociado a una disponibilidad continua del 100 % podría no compensar los beneficios obtenidos. En su lugar, debemos considerar la disponibilidad como una meta a perseguir, pero evaluando siempre coste y beneficio. En la arquitectura actual de los servicios web, la disponibilidad no está contemplada. En su lugar, las empresas proveedoras de infraestructura (servidores de aplicaciones, servidores web, etc.) proponen la mejora de disponibilidad como una mejora en la implementación de los servicios web.

Una vez desarrollado el componente que implementa la lógica de negocio que constituirá el servicio web, su puesta en funcionamiento podrá ser realizada de diferentes formas:

1. El servidor web, servidor de aplicaciones y servicio web se ejecutan en el mismo servidor (figura 2.14).

2. El servidor web se ejecuta en un servidor, mientras que el servidor

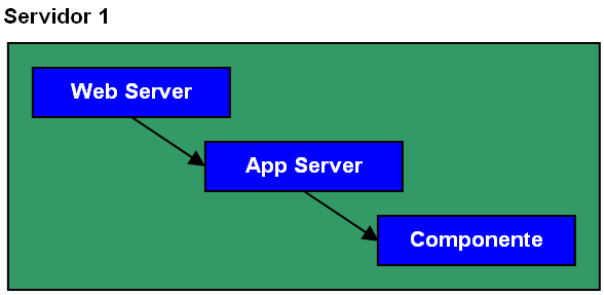

Figura 2.14. Configuración del proveedor con un servidor.

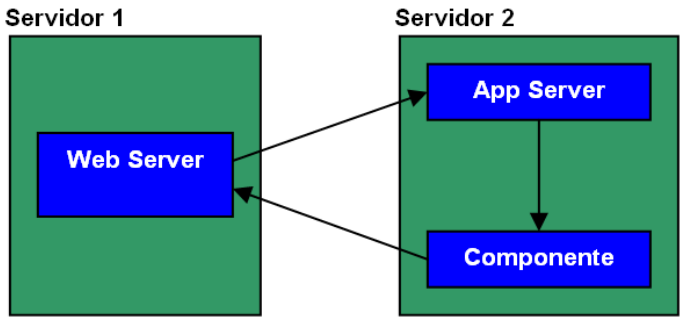

Figura 2.15. Configuración del proveedor con dos servidores.

de aplicaciones y el servicio web se ejecutan en otro servidor. Este estructura se suele utilizar por motivos de seguridad, para aislar el servidor que recibe las conexiones TCP [92] del exterior del servidor en el que se ejecutan los componentes (figura 2.15).

3. Los tres elementos (servidor web, servidor de aplicaciones y servicio web) se ejecutan en distintos servidores (figura 2.16).

Básicamente, la diferencia entre cada una de las tres estructuras radica en la localización en la que se ejecutan las cosas, y esto afectará a la disponibilidad de dos formas:

- Utilizar una única máquina reduce los puntos de fallo, pero el sistema tendrá demasiados puntos únicos de fallos (*single point of failure*).
- Utilizar más de una máquina aumenta el número de puntos de fallo, pero permite mejorar la disponibilidad a través de la redundancia.

Si queremos mejorar la disponibilidad de un sistema, en la actualidad disponemos de dos caminos alternativos, aunque a veces puedan llegar a ser complementarios:

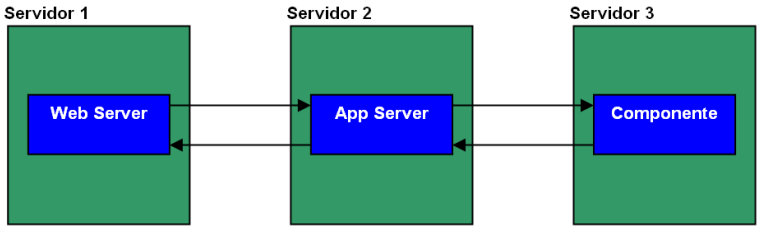

Figura 2.16. Configuración del proveedor con tres servidores.

- Sistemas tolerantes a fallos (*fault tolerant*) [91, 130].
- Sistemas *cluster* [124, 81].

En un sistema tolerante a fallos todos los componentes del sistema son redundantes (memoria, procesador, controladores de entrada/salida...). Si por algún motivo, un componente falla, el sistema, globalmente, seguirá funcionando, ya que el trabajo que realizaba el componente fallido pasará a ser realizado por otro componente redundante dentro del mismo sistema. Un sistema *cluster* está compuesto por varios servidores que se comunican a través de una red de área local. Si un componente del *cluster* falla, el *cluster* seguirá funcionando, ya que el resto de componentes se responsabilizarán de la ejecución de la carga de trabajo del nodo del *cluster* que ha fallado.

Típicamente, el primer modelo de implementación de servicios web (servidor web, servidor de aplicaciones y componente de negocio en la misma máquina) se encuentra asociado a los sistemas tolerantes a fallos, ordenadores *mainframe* en los que todos los componentes son redundantes. El tercer modelo de implementación (servidor web, servidor de aplicaciones y componente de negocio en distintas máquinas) lo encontraremos típicamente en sistemas *cluster* de servidores. En este tipo de implementaciones el *cluster* se suele dividir en tres capas, o bien crear tres sistemas *cluster*. Cada uno de los *clusters* se encargará de realizar una labor específica: servidor web, servidor de aplicaciones o componente de negocio.

En realidad es posible encontrar cualquier configuración intermedia de un sistema de alta disponibilidad, ya que los grandes sistemas se pueden agrupar en *clusters* (Sysplex [114] o GDPS [8]), y los sistemas *cluster* se construyen a partir de nodos que disponen de componentes redundantes (controladoras RAID [62, 120], sistemas multiprocesador...).

A través del uso de cualquiera de los dos medios que acabamos de describir (*cluster* o tolerancia a fallos), se consigue mejorar la disponibilidad del sistema utilizado para implementar la arquitectura de servicios web. Esta mejora se consigue a base de una solución tecnológica, no a través de una mejora en la arquitectura de los servicios web. Es decir, se mejora la implementación del servicio, pero no el servicio en sí mismo.

Un servicio web puede ser visto como la suma de tres elementos: un servidor web, un servidor de aplicaciones y un componente (el que implementa la lógica de negocio). Desde esta óptica, la mejora de disponibilidad se reduce a mejorar la disponibilidad de cada uno de los componentes, logrando de esta manera proteger la implementación del servicio (la suma de los tres elementos).

Como resultado, los proveedores protegen sus sistemas para ofrecer la mayor disponibilidad posible. Pero el punto de vista del cliente es completamente distinto. Si, por algún fallo en algún componente del sistema del proveedor, éste deja de responder a las invocaciones, lo que el cliente percibe es una caída del sistema, es decir, pasa a disponibilidad cero, ya que no tiene otra forma de ejecutar el servicio. Por ejemplo, si lo que falla es la lógica del componente de negocio, aunque la infraestructura esté funcionando correctamente, para el cliente la disponibilidad será cero.

Problemas similares a estos aparecen a la hora de escalar un sistema. Por las mismas razones antes expuestas, en las arquitecturas actuales, a la hora de escalar un servicio web, la solución generalmente adoptada consiste en escalar su implementación. Una vez más la solución se basa en trabajar sobre la implementación del servicio, y no sobre el servicio en sí mismo. La escalabilidad también se ve afectada por los dos tipos de soluciones aplicables a los problemas de disponibilidad: *clustering* y tolerancia a fallos.

En realidad, todos los problemas referentes a lo que es la explotación de servicios web a gran escala no están todavía contemplados en la arquitectura actual. Para todos ellos, el enfoque se basa siempre en la implementación del servicio, y no en el interfaz, que es lo que ven las aplicaciones cliente.

Como resultado de estas carencias, las aplicaciones cliente no tienen garantizada una disponibilidad mínima, al estar ligados a un proveedor concreto, que puede en cualquier momento incumplir las condiciones del contrato firmado con un cliente, al no poder ofrecer el servicio. Si los clientes pudiesen cambiar automáticamente de proveedor al detectar una situación anómala, la disponibilidad del servicio mejoraría, y esto se haría independientemente de la disponibilidad de la implementación.

Las implementaciones que encontramos actualmente siguen dos caminos bien diferentes. Por un lado, una alianza entre IBM y CISCO ha dado como resultado la arquitectura HAWS [71] (*High Availability Web Services*) para entornos *mainframe*. La solución HAWS se corresponde con una arquitectura tolerante a fallos, en la que se utiliza una infraestructura de red provista por CISCO, y uno o varios ordenadores centrales S/390 de IBM.

En la solución HAWS se introduce un elemento repartidor de tráfico, el MNLB (*MultiNode Load Balancing*, ahora sustituido por los productos de la serie Content Services [4]), dos *routers* de gran capacidad (los Catalyst en la

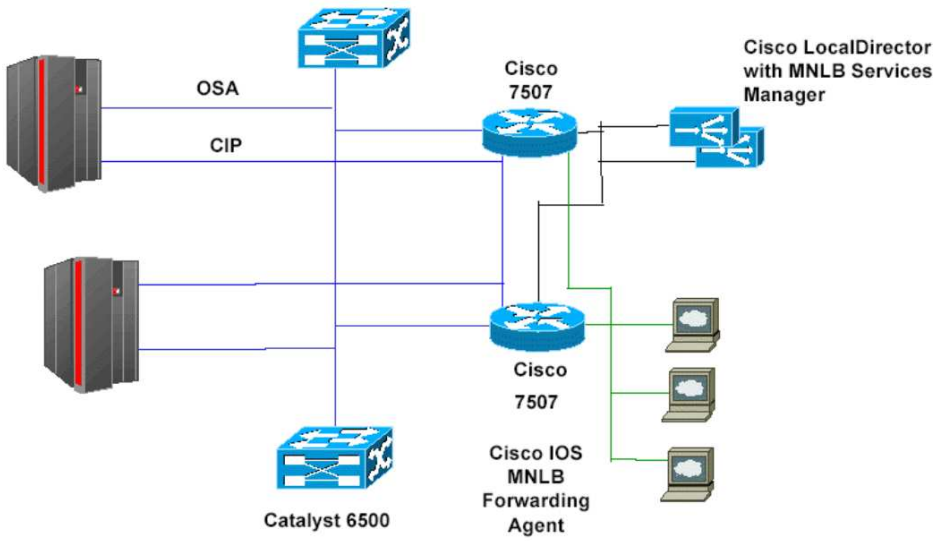

Figura 2.17. Estructura de un sistema HAWS.

figura 2.17), y como *back-end* de toda la solución dos sistemas S/390 configurados en Sysplex. La solución es escalable, pudiendo aumentar el número de repartidores, *routers* o *mainframes*. Como hemos dicho anteriormente, esta solución se basa en mejorar la disponibilidad de la implementación que, finalmente, reside completamente en los servidores centrales, ya que la infraestructura de red se dedica exclusivamente a la gestión de protocolos de bajo nivel (IP, TCP,...). La solución hardware se apoya en una solución software: el WLM [48] (*Work Load Manager*) y el *Cisco Agent for OS/390*. La combinación de ambos permite a los repartidores (a través del agente) decidir en todo momento cual de los servidores del Sysplex es el más adecuado para enrutar una petición. El agente de CISCO obtiene la información del WLM, que es el componente software encargado de gestionar la carga de trabajo del Sysplex.

Entre las soluciones de disponibilidad basadas en *cluster*, encontramos dos tendencias claramente diferenciadas: las basadas en IIS [16] (*Internet Information Server*) y las basadas en la plataforma Java, normalmente J2EE (*Java 2 Enterprise Edition*).

Cuando se utiliza plataforma Windows, el servidor web utilizado suele ser el IIS. Este servidor web es un elemento software del tipo *cluster-aware*, es decir, que puede instalarse sobre los servicios *cluster* de Microsoft (MSCS [13], *Microsoft Cluster Services*) o de cualquier otro ISV (como Veritas [27],...). De esta forma se puede crear un *cluster* de servidores web con los que se puede repartir la carga.

Desde el punto de vista del desarrollo, la tendencia en entornos Microsoft

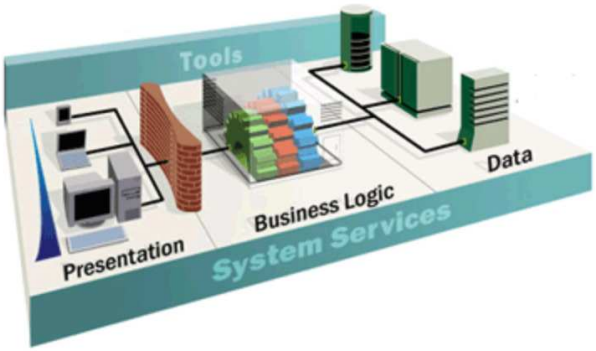

Figura 2.18. Arquitectura DNA.

es utilizar DNA. DNA es una arquitectura de aplicaciones en tres capas: presentación, lógica de negocio y acceso a datos (figura 2.18).

La capa de presentación se realiza dentro del IIS, típicamente utilizando páginas ASP. La capa de lógica de negocio se desarrolla normalmente utilizando DCOM: componentes software que son invocados de forma distribuida. Y la capa de acceso a datos se desarrolla también en DCOM, pero como componentes separados de los de lógica de negocio, que simplemente implementan el acceso a las bases de datos; es simplemente una técnica más para independizar la lógica de negocio del proveedor de bases de datos que se esté utilizando.

Esta arquitectura de desarrollo, bastante exitosa a finales de los 90, fue cediendo terreno a arquitecturas basadas en Java por múltiples razones. Las dos más importantes son sin duda la portabilidad y las capacidades del lenguaje. Por un lado, la arquitectura DNA se desarrolla mayoritariamente en VisualBASIC, que es una evolución más o menos orientada a objetos del BASIC (no tiene herencia ni polimorfismo), mientras que Java es un lenguaje verdaderamente orientado a objetos. Por otro, la plataforma DCOM es propietaria y, salvo extraños experimentos comerciales [23], no la encontraremos fuera de Windows, mientras que la plataforma Java está disponible para cualquier plataforma, incluida Windows.

La plataforma Java, desde el punto de vista de servidores de aplicaciones, tiene una arquitectura similar a la de Microsoft. Está organizada también en tres capas, pero los protocolos y componentes utilizados son completamente diferentes. La lógica de presentación se realiza utilizando JSP, y los componentes utilizados para desarrollar la lógica de negocio son EJB.

Como vemos, tanto en las soluciones de alta disponibilidad basadas en sistemas *fault-tolerant*, como en las basadas en sistemas *cluster*, lo que se protege es la implementación del servicio web.

A la hora de mejorar la disponibilidad de un servicio web, como hemos visto, no basta con proteger la implementación. El interfaz, que es lo que ven las aplicaciones cliente, debe ser igualmente protegido. ¿Cómo se puede proteger algo tan intangible como el interfaz? Simplemente separando el interfaz de la implementación, es decir, haciendo que el punto en el que se reciben las peticiones de los clientes y el punto en el que se resuelven sean distintos.

Para alcanzar esta separación, proponemos el uso de la invocación indirecta, lo que permitirá presentar el interfaz en una capa intermedia (con la que dialogan las aplicaciones cliente), y la implementación en una capa alejada de los clientes, que es invocada desde la capa intermedia.

De esta forma, se puede agregar capacidad de proceso a la capa intermedia, para que sea capaz de, ofreciendo un único interfaz, invocar distintas implementaciones residentes en distintos puntos o nodos de la red.

2.7. Calidad de Servicio

El término *Calidad de Servicio* (*Quality of Service*, QoS) abarca un conjunto de técnicas y tecnologías utilizadas con el fin de poder ofrecer resultados predecibles. Típicamente, encontramos el término QoS asociado a las tecnologías de transmisión de datos. Por ejemplo, se suele utilizar las técnicas de calidad de servicio para garantizar un ancho de banda determinado, una tasa de errores limitada o un porcentaje de disponibilidad de una red.

La calidad de servicio es un concepto que en realidad se aplica a cualquier elemento hardware o software que provea un servicio. Por ejemplo, como acabamos de ver, una red es un servicio que provee un medio de transporte para paquetes de datos. Desde este punto de vista, un servicio web es también susceptible de ser analizado desde una óptica de calidad de servicio.

Tomemos como ejemplo un servicio web de acceso a las cotizaciones de un mercado bursátil. El mayor interés de las aplicaciones cliente que utilizan ese servicio se centrará en la disponibilidad. Pero ese no es el único factor que influye en el buen funcionamiento de las aplicaciones. Otros factores como el tiempo de respuesta, o el rendimiento (entendido como número de peticiones servidas por unidad de tiempo) también serán cruciales a la hora de tener que elegir entre un proveedor u otro.

Las variables que acabamos de ver (rendimiento, tiempo de respuesta, etc.) pueden ser utilizadas para medir la calidad de servicio de un servicio web determinado. Pero estos factores no son los únicos que interesan al cliente. El cliente también estará afectado por variables que miden la calidad

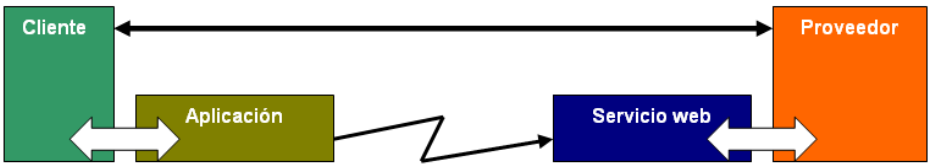

Figura 2.19. Proveedores y sus servicios.

de servicio del proveedor. Es decir, una vez más encontramos la necesidad de establecer una separación clara entre el proveedor del servicio web y el proveedor del servicio.

Normalmente, cuando se habla de calidad de servicio, se tiende a establecer un conjunto de métricas que permita cuantificar esa calidad respecto de un servicio determinado. Pero eso puede ser insuficiente, ya que muchas veces, la elección de un servicio web u otro, dependerá de las garantías que nos ofrezca el proveedor de dicho servicio, o de características propias del proveedor, ajenas al servicio web. Es decir, necesitamos poder diferenciar **métricas de servicio** de **métricas de proveedor**.

Una forma fácil de ver la necesidad de esta separación consiste en evaluar dos variables de calidad de servicio a la vez. A modo de ejemplo tomemos dos variables concretas: disponibilidad y coste. La disponibilidad de un servicio es algo fácilmente cuantificable. La ecuación 2.1 nos muestra una forma sencilla de calcularla.

$$Disponibilidad = 100 * InvocacionesExitosas/InvocacionesTotales \tag{2.1}$$

Por otro lado queremos evaluar también el coste de invocación de un servicio. Nos estamos refiriendo ahora al coste real (en unidades de moneda) que nos factura el proveedor del servicio por cada petición que le hacemos. Como hemos dicho antes, la variable coste, como una variable asociada a un proveedor, representa también una métrica de calidad de servicio.

Desde el punto de vista de las aplicaciones cliente, conocer el valor de la variable "disponibilidad" es importante. Pero más lo es aún, conocerlo en función del coste, es decir, poder disponer de una medida de calidad de servicio que relacione la disponibilidad y el coste. Este razonamiento es extensible a cualquier número y mezcla de variables.

Independientemente del número de variables que el cliente quiera manejar, individual o conjuntamente, habrá que tener en cuenta la necesidad que el cliente tiene de poder gestionar por separado la calidad de servicio del servicio web y la calidad de servicio del proveedor del servicio web.

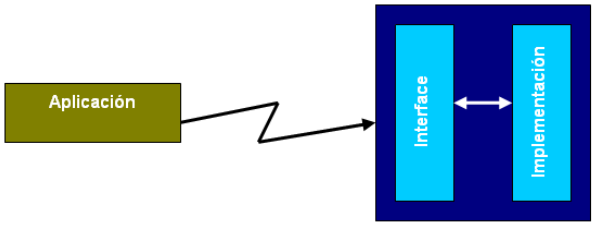

Figura 2.20. Interfaz e implementación.

A la hora de definir métricas de calidad, debemos tener en cuenta un factor muy importante, que es, como hemos dicho antes, la separación entre servicio y proveedor (figura 2.19). A menudo se tiende a definir métricas de todo tipo que permiten medir el funcionamiento del servicio que se está utilizando. En el caso de una red de datos, está bastante claro que las métricas se utilizan para cualificar el comportamiento de la propia red. Pero una aplicación que utilice esa red deberá preocuparse también por el comportamiento de la aplicación con la que está dialogando al otro lado de la red.

Algo similar ocurre con el servicio web y el proveedor de dicho servicio desde el punto de vista del cliente: el **interfaz** del servicio tiene unas métricas que lo definen, pero la **implementación** del servicio tiene otras bien distintas. El interfaz de un servicio web es la parte del servicio con la que dialogan las aplicaciones cliente, mientras la implementación (figura 2.20), que es algo invisible desde el punto de vista del cliente, es la parte del servicio afectada más directamente por el proveedor. Un ejemplo claro, lo podemos encontrar cuando queremos medir la calidad de servicio que nos ofrecen dos proveedores distintos que ofrecen sendos servicios web, y los servicios web son iguales (desde el punto de vista del cliente).

Cada vez encontramos más trabajos relacionados con QoS en servicios web, pero todos ellos están dedicados a la definición de métricas. En general, se definen métricas y unidades de medida que han de servir como base para una calificación del servicio web. Además, en los trabajos sobre métricas, no se diferencia entre calidad de servicio del servicio web y calidad de servicio del proveedor de dicho servicio web.

En [109] se definen varias métricas (disponibilidad, seguridad o tiempo de respuesta), como medidas de la calidad de un servicio determinado. Estas métricas deben ser controladas por algún agente monitor, que observe en todo momento las invocaciones y sus respuestas. Según [109], disponer de métricas es un primer paso en la consecución de un modelo de QoS, que debiera, en el caso de relaciones de larga duración entre cliente y proveedor, materializarse en el uso de acuerdos de nivel de servicio (SLA, *Service Level Agreement*). La gestión de las variables de QoS, así como el cumplimiento

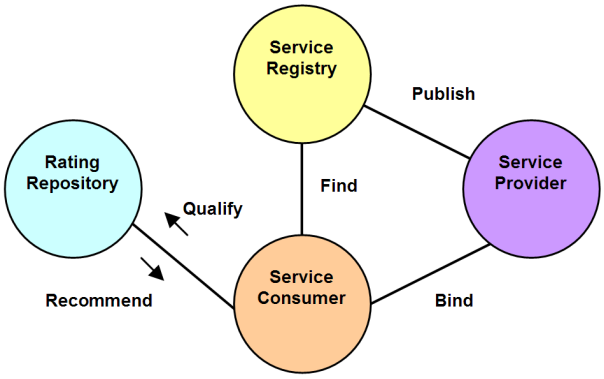

Figura 2.21. Entidades en la arquitectura estándar con repositorio.

de las SLA firmadas por ambas partes, deberá ser controlado por un agente monitor externo.

En [67] se analiza la forma en la que se puede mejorar la calidad de servicio de los servicios que se ofrecen a través de la web, entendiéndolos como servicios en general y no como componentes invocables. Distintos métodos de mejora de QoS se centran básicamente en mejorar la disponibilidad y el tiempo de respuesta. Para ello, se utilizan técnicas de *caching*, replicación de servidores o replicación de documentos. Aunque los servicios web comparten la mayor parte de su infraestructura con las aplicaciones comunes de la web, estas soluciones no son, generalmente, aplicables.

La replicación de documentos no es aplicable para los servicios web, ya que en su invocación nos interesa la implementación de dicho servicio. Para mejorar la disponibilidad del servicio se pueden utilizar técnicas de replicación de servidores, de tal forma que las invocaciones sean servidas desde el servidor más adecuado en cada momento. Por último, las técnicas de *caching* son aplicables en entornos muy concretos. Por ejemplo, supongamos la existencia de un servicio web que ofrece las cotizaciones de bolsa al cierre del mercado. Durante un largo período de tiempo (hasta el día siguiente), dichas cotizaciones no variarán, con lo que podría aplicarse algún mecanismo de *caching* para acelerar los tiempos de respuesta.

Las técnicas de mejora en la utilización de la red (ancho de banda) también tratadas en [67], así como la disminución de los tiempos de respuesta, son aplicables por igual a los servicios web y a los usos habituales de la web, ya que ambos comparten la misma infraestructura.

Otros trabajos en el ámbito de la empresa privada [132, 107] se centran también en definir métricas para poder construir un modelo de calidad de servicio. Dichas métricas, una vez más, se centran en calificar el servicio web.

En [141] se muestra un mecanismo teórico que permite la calificación de los servicios web. Según el sistema propuesto, a la estructura tradicional de la arquitectura de servicios web (cliente, proveedor y registro), se añade un cuarto componente, el repositorio de calificaciones (figura 2.21). Este repositorio es accedido de dos maneras. En primer lugar, los clientes de servicios web añaden las calificaciones que han obtenido a partir de llamadas a un servicio web y un proveedor dados. El repositorio computa y almacena datos históricos.

En segundo lugar, nuevos clientes de servicios web acceden al repositorio para obtener información acerca de servicios web. Dicha información es información histórica, obtenida de las calificaciones que otros clientes han otorgado a ese servicio y proveedor.

En general, los trabajos publicados referentes a QoS para servicios web, así como las soluciones disponibles, sólo definen métricas asociadas a las invocaciones; no contemplan en los modelos de QoS la posibilidad de incluir métricas que definan al proveedor del servicio, con lo que la calidad de servicio percibida por las aplicaciones cliente estará siempre ligada a la calidad de servicio de un servicio web.

Esto está directamente relacionado con la ya mencionada separación de roles. Para que las entidades cliente puedan obtener una medida precisa de la calidad de servicio, será necesario que dicha medida incluya información referente a un servicio web y al proveedor de dicho servicio web.

Nuestra propuesta de invocación indirecta **permite caracterizar un servicio web mediante un sistema de expresiones matemáticas** en las que se incluyen variables que representan el comportamiento histórico del servicio web así como el del proveedor de servicio.

La solución que nosotros proponemos permite a las aplicaciones cliente especificar sus preferencias a la hora de utilizar un servicio web u otro sin necesidad de que las aplicaciones cliente estén enlazadas con más de un servicio web. Además, las preferencias de las entidades clientes pueden reflejar información relativa tanto a los servicios web como a las entidades que los ofrecen (los proveedores de servicio).

2.8. Composición

Uno de los puntos calientes en la tecnología de servicios web es, sin duda, la composición de servicios. Los sistemas (teóricos y prácticos) que permiten realizar dicha composición se basan en utilizar algún lenguaje de modelado que les permita crear nuevos servicios web mediante la agregación de servicios web más simples.

Figura 2.22. Evolución del desarrollo distribuido.

El principal objetivo de la composición es el de crear una infraestructura capaz de dar soporte a las necesidades de negocio de la forma que más se acople al mismo. Es decir, construir bloques de software cada vez más grandes que se aproximen a los procesos de negocio. Esta evolución se ha venido acentuando a lo largo del tiempo, desde las clases de los lenguajes de programación hasta los actuales servicios web (figura 2.22). Dicha evolución ha disminuido el acoplamiento entre elementos, a la vez que nos ha llevado a construir elementos software cada vez más grandes en lo que se refiere al proceso que realizan.

Los trabajos actuales en esta materia se están centrando en definir un lenguaje de descripción que permita definir cual ha de ser la estructura interna de un servicio compuesto, y el modo en el que han de realizarse las tareas intermedias con el fin de alcanzar la ejecución del servicio. Dicho lenguaje debería contemplar al menos los siguientes componentes:

- **Estaciones de proceso**: puntos intermedios del servicio compuesto en los que se ha de realizar algún proceso.
- **Flujo de control**: Especificación del orden en el que han de ser invocadas las estaciones de proceso.
- **Estados**: El lenguaje debe permitir una definición y un control exhaustivo de cada uno de los estados en los que puede encontrarse cada elemento del servicio web compuesto, así como el estado del propio servicio.
- **Motor**: Es el responsable de gestionar cada uno de los elementos anteriores.

Dentro del área de la creación de *workflows* (flujos de ejecución de tareas), la iniciativa de la empresa privada es muy fuerte, adelantándose a menudo a las iniciativas institucionales como las del consorcio. Es por ello que los trabajos en este campo a veces se limitan a implementaciones que se materializan en productos comerciales.

Por este mismo motivo el área de la composición corre el peligro de perder uno de los valores más apreciados de los servicios web: la utilización de estándares. Ello se debe a que las iniciativas no están orientadas a

completar la arquitectura o a publicar estándares o recomendaciones, sino a construir productos comerciales que, por otra parte, resultan muy atractivos a las empresas privadas que desean modelar sus procesos de negocio de una forma más sencilla que la que ya conocen: el desarrollo de software. Independientemente de los objetivos que mueven a los equipos de desarrollo de las grandes empresas, deben tenerse en cuenta algunos trabajos que se están desarrollando en la actualidad.

Uno de los proyectos más importantes en el campo de la composición de servicios es el BPEL4WS, codirigido, principalmente, por IBM y Microsoft, y acompañados en su aventura por Siebel, SAP y Bea. BPEL4WS es el resultado de la unión de los esfuerzos de Microsoft e IBM por construir un lenguaje de definición de procesos. La experiencia obtenida de los trabajos realizados dentro del proyecto XLANG [134] (el lenguaje de definición de *workflows* de Microsoft) unido con los resultados del WSFL [103] (*Web Services Flow Language*) de IBM, ha sido utilizada para crear este nuevo lenguaje de definición de procesos. BPEL4WS ha sido sometido a consideración por los comités técnicos de OASIS [18] (*Organization for the Advancement of Structured Information Standards*), más concretamente al *Web Services Business Process Execution Language TC*, y no ha sido hasta la fecha enviado al consorcio, lo que puede plantear algunos problemas en el proceso de estandarización del lenguaje.

BPEL4WS, ha sido diseñado para incorporar las siguientes funciones principales:

- Recibir y manipular mensajes XML que podrán ser reenviados a otros servicios.
- Recibir mensajes XML de los servicios remotos de forma asíncrona.
- Gestionar eventos y excepciones.
- Permitir el uso de variables y relaciones entre mensajes.
- Crear flujos de ejecución secuenciales y paralelos.
- Permitir deshacer modificaciones cuando se produce una excepción.

Además de estas funcionalidades, BPEL4WS ha sido construido como un lenguaje compatible con los lenguajes en uso actualmente (SOAP y WSDL), de tal forma que puede interactuar con servicios web estándar.

BPEL4WS cubre dos aspectos de la definición de procesos de negocio (figura 2.23). Por un lado, la "coreografía", entendida como la descripción de un protocolo de negocio, y por otro la "orquestación", que es la descripción de los procesos ejecutables de negocio. Es decir, la coreografía hace

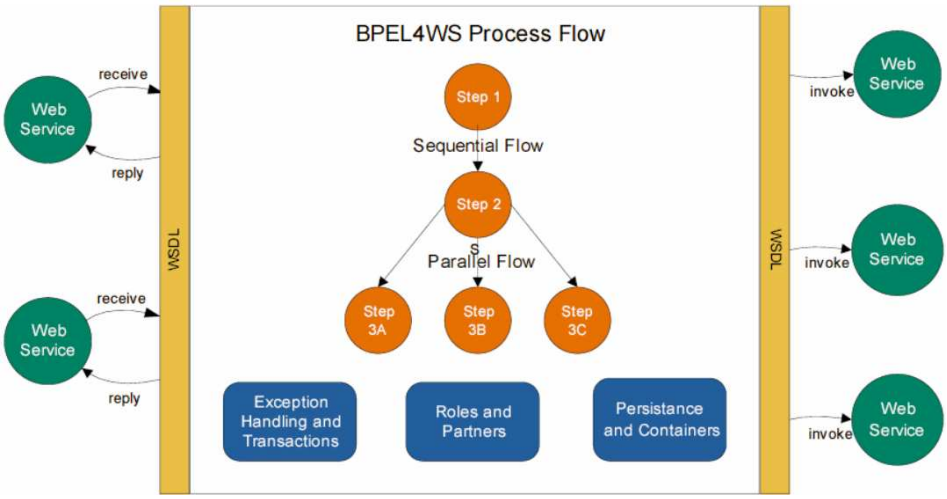

Figura 2.23. Flujo de procesos con BPEL4WS.

referencia a las interacciones que el proceso de negocio tiene con elementos externos, mientras que la orquestación se refiere a como se implementa ese proceso de negocio desde el punto de vista de la interacción entre los distintos componentes de negocio.

Otra propuesta en el campo de la composición de servicios es la que representa el WSCI [46] (*Web Services Choreography Interface*). El WSCI, impulsado fundamentalmente por Sun, BEA y SAP se centra únicamente en el aspecto coreográfico de la composición, es decir, en las interacciones entre un servicio y el exterior.

La coreografía es también el objetivo de la propuesta del consorcio: WS-Choreography [33]. Esta propuesta, que se encuentra todavía en sus etapas iniciales de desarrollo, tiene como fin el desarrollo de un lenguaje que permita describir las interacciones entre los servicios y sus usuarios (otros servicios, seres humanos, aplicaciones, etc.).

La *Business Process Management Iniciative* [3] (BPMI) es una entidad independiente encabezada por Sun e Intalio, entre otros, que desarrolla sus actividades en el entorno de la estandarización de la gestión de los procesos de negocio. De la BPMI nace el lenguaje BPML [45] (*Business Process Management Language*), un estándar abierto que se utiliza para definir procesos ejecutables de negocio. BPML se centra en la orquestación de los procesos de negocio.

Las cuatro iniciativas que acabamos de ver cubren distintos aspectos del problema de la composición. Por una parte, el BPEL4WS ataca los problemas de coreografía y orquestación, mientras el BPML se centra en la orquestación, WSCI y WS-Choreography se preocupan por la coreografía,

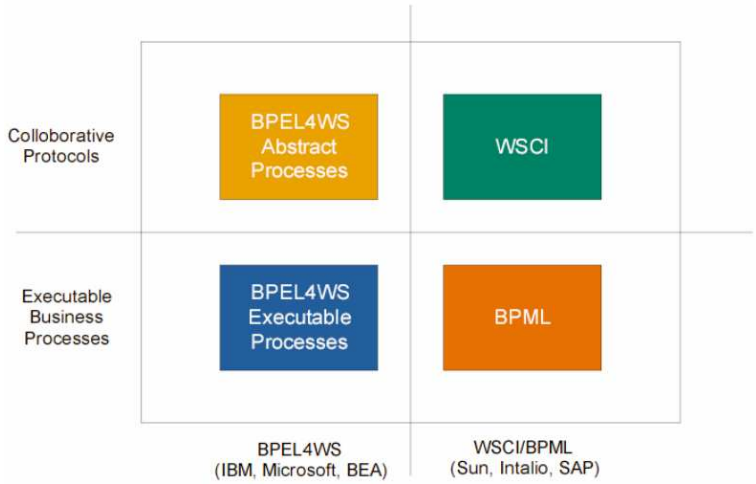

Figura 2.24. Lenguajes en la composición de servicios.

aunque es de esperar que BPMI, con WSCI y el estándar del consorcio formen un único frente de cara a la composición de servicios. En la figura 2.24 se muestra un cuadro comparativo de las tecnologías y sus ámbitos de aplicación.

El área de la composición de servicios por agregación permitirá aplicar la utilización de técnicas de ingeniería de software al desarrollo de aplicaciones basado en servicios web, ya que será posible construir aplicaciones en base a la utilización de componentes de software (servicios web) encapsulados de manera transparente y estandarizada. La agregación será pues el proceso responsable de permitir la simplificación del desarrollo de aplicaciones complejas.

Nuestra propuesta en el campo de la agregación ofrece dos posibilidades bien diferenciadas. Por un lado, la integración de nuestra solución con BPEL4WS abrirá una vía en la publicación de *workflows* como servicios web virtuales. Por otro, nuestra propuesta basada en virtualización permitirá el desarrollo de un lenguaje de programación para servicios web, de tal forma que utilizando técnicas de programación tradicionales se podrán construir servicios web complejos a partir de servicios simples.

La propuesta se basa en permitir el desarrollo de flujos de control apoyándose en la utilización de las estructuras de programación básicas (secuencial, condicional, iterativa, etc.). Estos flujos permitirán desarrollar lo que podríamos denominar un "programa" cuyas instrucciones son llamadas a servicios web.

2.9. Acuerdos de Nivel de Servicio

Un acuerdo de nivel de servicio (SLA, *Service Level Agreement*) es un contrato entre un proveedor de un servicio y un cliente que lo utiliza. Este contrato se redacta en términos de unidades que puedan ser medidas, de tal forma que ambas partes pueden saber en todo momento si se están cumpliendo las condiciones del contrato. El término SLA no es un término específicamente informático, y menos aún de los servicios web. En realidad, los acuerdos de nivel de servicio existen hace mucho tiempo, y se firman entre empresas cuando una de ellas (cliente) encarga la ejecución de un servicio a la otra (proveedor). También es posible encontrar acuerdos de nivel de servicio entre distintos departamentos de la misma entidad.

Existe una gran relación entre la calidad de servicio y los acuerdos de nivel de servicio (QoS y SLA), ya que los contratos que utilizan clientes y proveedores para firmar acuerdos (SLAs) se redactan utilizando métricas que definen el servicio. Normalmente, estas métricas son las mismas que se utilizan para medir la calidad de servicio.

El ciclo de vida de una SLA es, a grandes rasgos, el siguiente:

1. **Definición de la SLA**, que incluye las siguientes actividades:
   - Definir SLO (*Service Level Objectives*), que serán los objetivos a cumplir para satisfacer las condiciones de la SLA.
   - Definir alcance de la SLA, en términos de período, elementos implicados,...
   - Definir penalizaciones por incumplimiento de alguno de los objetivos detallados en la SLA así como las acciones a tomar.
2. **Control de métricas**. En tiempo real (o lo más cercano a él que sea posible y/o razonable) se debe evaluar todas las métricas implicadas en cada SLA.
3. **Control de los objetivos**. Evaluar si las métricas recogidas cumplen los SLO.
4. **Gestión de incidencias/penalizaciones**. Si se detecta algún incumplimiento, realizar las acciones acordadas en la SLA.
5. **Eliminación de una SLA**. Dependiendo de los periodos de validez que se hayan especificado (y de otros posibles acuerdos de extinción de la SLA), será necesario comprobar en todo momento la validez del contrato y dejar de aplicarlo cuando corresponda.

A la vista de este ciclo de vida, dos aspectos importantes deben ser tratados con especial atención dentro del proceso de gestión de una SLA. En primer lugar la **toma de medidas en tiempo real** y, en segundo lugar, las **acciones a llevar a cabo** en caso de incumplimiento de los objetivos.

En el entrono de los servicios web, la toma de medidas se traduce en recoger datos reales de las invocaciones que se están realizando en cada momento. Para ello, el proveedor debe tomar datos de sus sistemas, y las aplicaciones cliente harán lo propio con los servicios que utilicen. Esta forma de trabajar puede plantear algunos problemas de confianza entre cliente y proveedor, y de disparidad en la información recogida. La mayor parte de estos problemas suele deberse al tiempo de red, ya que el transporte de los datos queda fuera del alcance de las SLA. Para tener control sobre los tiempos de red sería necesario utilizar mecanismos de gestión de la calidad de servicio y SLA sobre la red, lo que nos llevaría a la necesidad de gestionar dos SLA de distinta naturaleza para el mismo servicio.

Para la gestión de los objetivos, y las penalizaciones asociadas, es necesaria la utilización de algún mecanismo de tratamiento de excepciones, que permita tomar acciones automáticamente al detectar el incumplimiento de alguno de los objetivos de la SLA. Dichas acciones pueden ser de dos tipos básicos: las que conducen a un cambio automático en el sistema, o las que producen un evento que ha de ser resuelto desde el exterior del sistema.

En una acción automática, el cliente podría, por ejemplo, cambiar de proveedor, lo que implicaría enlazar el componente de gestión de incidencias y penalizaciones con los *proxies* utilizados por las aplicaciones cliente. En una acción externa, un cliente podría, por ejemplo, percibir indemnizaciones económicas por parte del proveedor, para lo que sería necesario habilitar sistemas de notificación de incidencias a las partes.

No sólo los clientes deberán ejecutar acciones al detectar el incumplimiento de algún SLO; también los proveedores necesitan estar informados de las eventualidades que se produzcan en los servicios que están ofreciendo, con el fin de subsanar cualquier posible problema en sus sistemas y poder ofrecer a sus clientes los servicios según los acuerdos preestablecidos.

Un buen ejemplo de una arquitectura que soporta las necesidades que acabamos de describir lo encontramos en [70] (ver figura 2.25). En ella, la gestión de las SLA, una vez creadas, empieza por los servicios de medida (*Measurement Services*), que son puntos de toma de medidas que pueden establecerse en distintos puntos del sistema. Las medidas recogidas son enviadas al servicio de evaluación de condiciones (*Condition Evaluation Service*), que se encarga de comparar dichas medidas con los SLO. En función de los resultados de dichas evaluaciones, se envían notificaciones al servicio de gestión (*Management Service*). Por último, dicho servicio podrá conducir a la ejecución de acciones sobre las aplicaciones cliente y sobre las aplicaciones

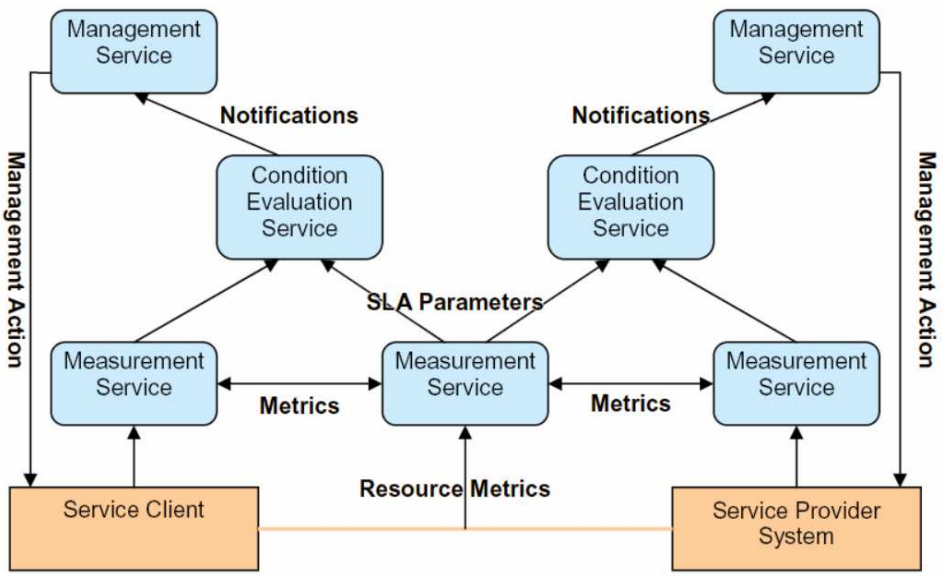

Figura 2.25. Entidades y relaciones en una arquitectura de SLA.

proveedoras.

El elemento clave de esta arquitectura lo constituye, sin duda alguna, el servicio de medida (*Measurement Service*), responsable de detectar los posibles incumplimientos de acuerdo que se puedan producir. En la solución de [70] no se propone ninguna implementación para este componente. En realidad, la arquitectura propuesta es teórica en su conjunto; es simplemente la arquitectura que los autores consideran que debería tener un sistema para control y gestión de SLA's.

En [106] se define un lenguaje que permite construir SLA utilizando la arquitectura que acabamos de describir: el WSLA (*Web Service Level Agreement*). Dicho leguaje se basa en tres conceptos principales: las partes, la definición de servicio y las obligaciones.

Una **parte** es cada una de las entidades que intervienen en la SLA que se describe. Existen dos tipos de partes, las firmantes (*Signatory Parties*) que son el cliente y el proveedor, y las partes de soporte (*Supporting Parties*), que son las que pueden actuar en representación de alguna de las partes firmantes para realizar alguna labor en el proceso de gestión de las SLA, como puede ser la toma de medidas.

La **definición de servicio** representa los servicios web que se quieren gestionar con la SLA que se describe. Y las **obligaciones** son en realidad los SLO y las acciones a llevar a cabo en caso de incumplimiento. El lenguaje también da soporte a otros elementos no menos importantes como especificación de períodos, *triggers* (disparadores), parámetros o funciones,

que no trataremos aquí.

De acuerdo a las propuestas de [106], los acuerdos de nivel de servicio deben ser redactados utilizando el lenguaje WSLA, produciendo de esta forma un documento que refleja los acuerdos alcanzados por las dos partes intervinientes: cliente y proveedor. Este documento ha de ser distribuido (*deployed*) al sistema (figura 2.26), para que cliente y proveedor (las partes firmantes), junto con las terceras partes en las que se haya delegado alguno de los servicios de control y gestión (las partes de soporte) tengan toda la información que necesitan.

Las partes de soporte podrán realizar alguna o varias de las siguientes funciones:

- Implementación del servicio de medida.
- Implementación del servicio de evaluación de condiciones.
- Implementación del servicio de gestión de acciones.

En ningún momento se apunta dentro de la solución propuesta, como ha de llevarse a cabo la implementación de la arquitectura. La verdadera aportación de este trabajo es la definición de una estructura bastante clara de cómo han de construirse los documentos que representan los acuerdos de nivel de servicio

En [100] podemos encontrar un buen estudio teórico sobre los problemas de la gestión de las SLA, tratando asuntos relacionados con la automatización de la gestión de las SLA. Se apunta la necesidad de que los proveedores construyan SLA's no firmadas (por los clientes), que serán publicadas. Constituyen en realidad esqueletos o modelos prefijados (*templates*) de SLA, entre los que los clientes han de escoger uno y firmar el acuerdo con la contraparte. De esta manera, los clientes podrán buscar, dentro de una lista de proveedores que puedan realizar un mismo tipo de servicio, aquellos que mejor se adapten a las condiciones de servicio que requiera. Esto permitiría automatizar el proceso de firma y distribución de los documentos, ya que los clientes, mediante una serie de directrices, serían capaces de buscar los servicios más apropiados.

No debemos olvidar que la localización automática de proveedores y de servicios web [108] es un tema complejo, completamente independiente del que tratamos aquí, pero necesario como paso previo a la selección automática de servicios según parámetros definidos dentro de un documento de acuerdo de nivel de servicio.

En todos los trabajos en los que se propone una arquitectura para la implementación de SLAs, se habla siempre de un servicio de medida o elemento de medida. Éste es el responsable de analizar cada invocación que se

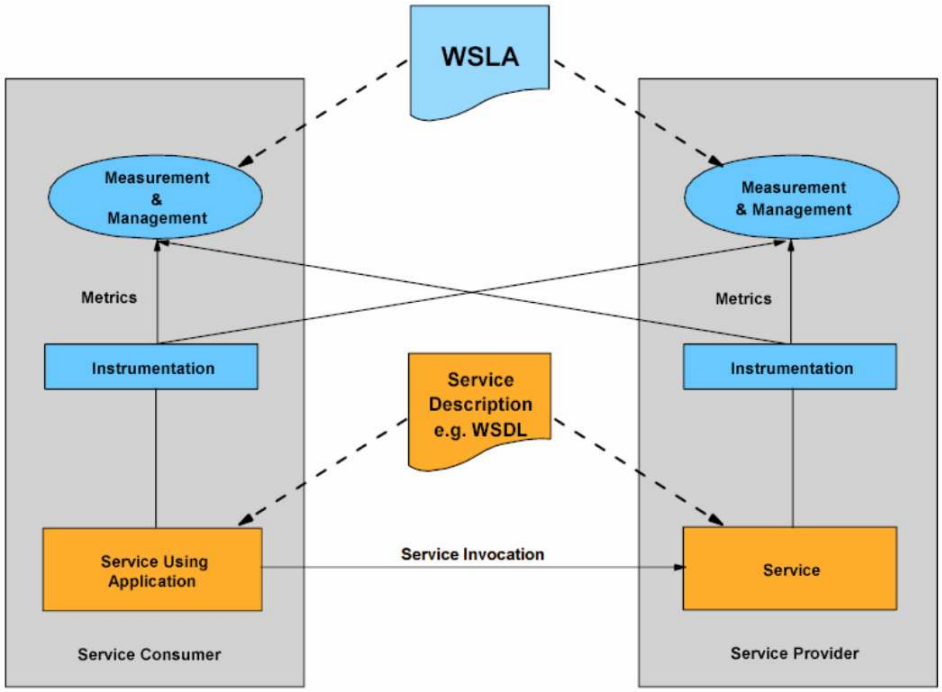

Figura 2.26. Funcionamiento de WSLA.

produzca entre un cliente y un proveedor y verificar si se han cumplido o no los términos especificados en una SLA. El problema surge a la hora de implementar este servicio de medida.

Nuestra propuesta de arquitectura con invocación indirecta ofrece una buena oportunidad para implementar los servicios de medida, ya que todas las invocaciones en nuestra propuesta viajan a través de componentes intermedios, responsables de canalizar los datos de las invocaciones hacia los proveedores y los resultados de las mismas hacia los clientes.

Así pues, la forma en la que se pueden llevar a cabo las mediciones se basará, según nuestra solución, en examinar la información de cada invocación (*online* o en tiempo diferido) y verificar el cumplimiento de los SLO. El componente intermedio será el responsable de desencadenar la ejecución de las acciones pertinentes en caso de que no se hayan cumplido dichos objetivos.

2.10. Esquemas de intercambio de mensajes

La arquitectura actual de los servicios web se apoya en los estándares existentes en la red. El lenguaje SOAP y el protocolo HTTP son la base

tecnológica de la mensajería y del transporte, respectivamente. Es por ello que los servicios web se definen a menudo como **componentes síncronos sin estado**. Se dice que son síncronos porque en este modo de funcionamiento, cuando un cliente realiza una invocación se envía un mensaje de petición; a continuación el cliente espera por la respuesta, lo que se traduce en que el proveedor debe enviar un mensaje de respuesta.

Se dice que los servicios web no tienen estado porque no existe relación alguna entre dos invocaciones consecutivas del mismo cliente al mismo proveedor. Es decir, cada vez que se realiza una invocación, en la parte servidor se instancia un componente servidor que recibirá la petición, la procesará (ejecutando algún tipo de proceso o acción) y devolverá un mensaje de respuesta. El ciclo de vida de esa invocación acaba ahí. La siguiente invocación que se realice no tendrá relación alguna con la anterior, ya que se ejecutará una instancia nueva del servicio proveedor.

El hecho de ser síncronos (como ya hemos comentado en secciones anteriores) y el de no tener estados plantea una serie de problemas de cara a la explotación total de la tecnología. El sincronismo y la ausencia de estados añaden un conjunto de limitaciones que no permiten aplicar la tecnología en situaciones en las que sería deseable.

Un ejemplo de ello son las transacciones de larga duración. Se trata de invocar un proceso de negocio cuyo tiempo de ejecución excede lo razonable para una aplicación *online*. Normalmente este tipo de procesos se relaciona directamente con procesos de negocio, como por ejemplo, la tramitación de un préstamo, o la confección de un paquete de viaje (avión, hotel y coche de alquiler). Puede incluso ocurrir, que incluyan procesos manuales dentro de su flujo de ejecución, donde la intervención humana forme parte de dicho flujo, pudiendo de esta forma prolongarse la ejecución por un tiempo indefinido.

Para poder implementar transacciones de larga duración dentro de la arquitectura de servicios web, será necesario disponer de un modelo de invocación en el que el cliente no espere por la respuesta del proveedor, sino que sea éste quien, al final del proceso (cuya ejecución puede finalizar por múltiples causas, no sólo por alcanzar el final del flujo), notifique al cliente tal evento. Esto nos lleva a utilizar un esquema asíncrono de intercambio de mensajes.

El asincronismo presenta ciertas peculiaridades que deben ser tratadas:

1. Los mensajes deben ser unívocamente identificados, ya que las peticiones y las respuestas van a ser entregadas, sin ningún orden, a los mismos proveedores y clientes. Al movernos en un entorno asíncrono, un mismo cliente puede lanzar varias peticiones a un proveedor y recibir las respuestas en un orden distinto, ya que el canal de ida y el

de vuelta son distintos (típicamente distintas conexiones HTTP). Por ello es necesario identificar de forma unívoca cada mensaje que se envía, para poder relacionarlo en un momento posterior con un mensaje de respuesta que deberá contener el identificador de la petición que originó dicho mensaje. Así se podrá relacionar las peticiones con las respuestas de forma inequívoca, de tal forma que cuando un cliente reciba una respuesta pueda saber a qué petición corresponde. En realidad, lo que se necesita es una forma de identificar todos los mensajes que formen parte de un mismo diálogo, entendiendo éste como un conjunto de invocaciones. Nos estamos refiriendo a lo que en BPEL4WS se denominan *correlation sets*.

2. Los mensajes de respuesta pueden ser entregados a los clientes (con la identificación pertinente) en cualquier momento. Es decir, el cliente manda la petición y no espera por la respuesta, con lo que está podrá producirse en cualquier momento. Recoger esta respuesta no es un proceso obvio, ya que implica la existencia de algún mecanismo para procesar los mensajes de forma desatendida. Básicamente, existen dos modelos que permitan la entrega asíncrona de mensajes: *polling* y *callback*.

Utilizando **polling** (sondeo) los clientes lanzan un mensaje de petición, es decir, hacen una invocación a un servicio. Esta invocación finaliza sin que se haya completado la ejecución del servicio solicitado por la entidad cliente. Debido a esto, como respuesta a la invocación se recibe algún *token* (ticket, testigo, identificador) que permitirá al cliente obtener la respuesta a la invocación del servicio en un momento posterior. Así, periódicamente, el cliente interroga al servicio web del proveedor para saber si ya está disponible la respuesta. En el momento que la respuesta al *poll* sea positiva, el cliente invocará de nuevo al servicio web para recibir la respuesta al primer mensaje. Este esquema presenta el inconveniente de que el cliente tiene que estar continuamente interrogando al servicio web a la espera de que se produzca la respuesta. Esto se traduce en dos problemas:

- Si se ha invocado un servicio web que está implementado como un flujo de trabajo, lo más probable es que el proceso sea largo. ¿Cada cuanto tiempo es conveniente realizar el *poll*? Si los intervalos de *polling* son muy largos, existe la posibilidad de que la respuesta se produzca en un instante dado dentro de un intervalo, y el cliente no la recoja hasta que finalice dicho intervalo, con lo que el cliente habrá esperado más tiempo del necesario. Si por el contrario, los intervalos son demasiado cortos, estaremos haciendo demasiadas consultas innecesarias.

- Consumo innecesario de recursos.

Por otra parte, utilizar una técnica de *polling* ofrece una ventaja muy importante: los clientes no necesitan implementar mecanismos para actuar como servidores en los que recibir los mensajes. Es decir, como el servicio web no va a notificar el resultado de una invocación al cliente que la originó, no es necesario que éste se ponga en "modo escucha", o sea, no tiene que actuar como servidor. Esto se traduce en que el servicio web no tiene que iniciar ningún tipo de conexión hacia el cliente, con lo que no habrá problemas de *firewalls* (el cliente podrá realizar invocaciones desde la red privada sin necesidad de permitirle al servidor en el que se ejecuta el servicio web acceder a dicha red); y no será necesario desarrollar lógica de servidor para una aplicación cliente (evitando la necesidad de desplegar componentes a un servidor de aplicaciones que se comuniquen con la aplicación cliente).

Utilizando **callback** (retrollamada) el cliente realiza una invocación que finaliza con una respuesta que indica la aceptación de dicha invocación, así como la recepción exitosa del mensaje correspondiente. Cuando el proceso del mensaje finaliza en el lado del proveedor (es decir, se completa la realización del servicio solicitado), el servidor realizará una invocación al cliente, a un punto (*endpoint*) que el cliente habrá especificado previamente.

Esto representa un avance considerable respecto a las técnicas de *polling*, ya que el cliente recibe la respuesta en el momento preciso en que ésta se produce. Por el contrario, la complejidad de los clientes aumenta, al hacerse necesaria la construcción de mecanismos que permita recoger los mensajes atravesando los sistemas de seguridad (*proxies*, *firewalls*, etc.).

La utilización de algún mecanismo que habilite el uso de invocaciones asíncronas permitirá la construcción de un tipo de proceso especial: el **proceso desatendido**. Los procesos desatendidos son procesos que se inician en un momento dado a partir de una intervención humana, o de algún evento determinado; que poseen una larga duración (la suficiente como para que no sea adecuado que el iniciador del proceso espere por la respuesta); y que son gestionados por algún motor de flujo que se encarga de controlar el estado en el que se encuentra cada proceso.

Tomemos como ejemplo una aplicación que actualiza una base de datos en la que se guardan las cotizaciones de cierre de los mercados bursátiles. Dicho proceso debe ser realizado diariamente o, incluso, a intervalos de tiempo inferiores a un día si se consideran mercados internacionales. Una forma de implementar este proceso consiste en utilizar algún sistema de *workflow* que permita construir un flujo con invocaciones a servicios web de cotizaciones de las distintas bolsas. En este flujo, se realizarán tantas llamadas como mercados bursátiles haya, se guardarán los resultados, y se esperará hasta el día siguiente, o hasta el intervalo siguiente, para repetir el proceso. Este proceso desatendido será iniciado invocando un servicio web, que causará el inicio de la ejecución del flujo, que se prolongará indefinidamente. No se

espera una respuesta, tan sólo que se ejecute día tras día.

En realidad los modelos de invocación asíncrona pueden dividirse en cuatro categorías ([40], [41]) en función del intercambio de mensajes que se produzca:

- **Sólo envío**: un cliente envía un mensaje y no espera respuesta (es el esquema utilizado para arrancar un proceso).
- **Envío y recepción**: un cliente envía un mensaje y espera una respuesta.
- **Recepción y envío**: se recibe un mensaje y se envía una respuesta (este es el punto de vista de un proveedor de servicio).
- **Solo recepción**: Se recibe un mensaje y no se envía respuesta (este esquema corresponde a los procesos desatendidos que acabamos de describir).

Estos cuatro patrones de intercambio de mensajes son en realidad dos, pero con dos distintos puntos de vista: aplicación cliente y proveedor de servicio web.

Existen diversas iniciativas en cuanto a nuevos esquemas de mensajería se refiere, pero todas en fase de estandarización. El BPEL4WS, que hemos mencionado anteriormente, soporta asincronismo a través del uso de un sistema de *polling*. Los clientes pueden invocar un servicio web y, periódicamente, pueden consultar el estado de ejecución en el que se encuentra el proceso que se inició con la invocación inicial.

Existen otras iniciativas como WS-Callback [85], específicamente diseñadas para solventar algunos de los problemas que acabamos de comentar. Este lenguaje (diseñado y construido como una extensión de SOAP y WSDL), permite especificar la URL a la que ha de enviarse un mensaje asíncrono de repuesta a una petición SOAP previa. Permite enviar en las cabeceras (*header*) de los mensajes SOAP la dirección de respuesta, que podrá ser modificada dinámicamente según los intereses de la entidad que emitió la petición.

De manera muy similar a la que trabaja WS-Callback, SOAP Conversation Protocol (SCP) [52] permite construir diálogos basados en SOAP y WSDL. Para ello, SCP permite especificar en las cabeceras SOAP las direcciones de respuesta y unos identificadores de conversación, lo que permite a cada una de las partes reconocer los mensajes que le llegan y responderlos de acuerdo al estado de la conversación en la que se encuentran.

La invocación asíncrona ha sido recientemente tratada de una forma estándar en la especificación WS-Addressing [54]. Esta pequeña especificación

permite a los creadores de servicios web detallar la forma en la que han de enviarse y recibirse los mensajes: síncronamente o asíncronamente. Adicionalmente, el hecho de poder especificar distintas direcciones para emisor, remitente y dirección de respuesta, abre otra puerta al mundo del proceso de mensajería SOAP mediante nodos intermedios, de una forma sencilla y estandarizada.

Una forma sencilla de realizar invocaciones asíncronas se basa en utilizar un protocolo asíncrono en lugar de uno síncrono. Es decir, podemos dejar de utilizar HTTP (protocolo síncrono) para utilizar SMTP, que es un protocolo asíncrono desde el punto de vista de la entrega de mensajes.

Según este esquema, cuando un cliente quiere realizar una invocación, lo único que ha de hacer es enviar por correo electrónico el mensaje SOAP que la describe. El protocolo SMTP reencaminará el mensaje hasta llegar al servidor adecuado, donde será entregado a un servicio web que lo examinará y ejecutará algún tipo de acción. El servicio web devolverá el mensaje por SMTP, que llegará al origen donde será entregado a alguna aplicación cliente. La mensajería de todo este proceso es asíncrona, es decir, el mensaje de petición y el de respuesta no tienen relación alguna desde el punto de vista del canal de transmisión.

El uso de SMTP como medio de transporte de SOAP no está muy extendido. Algunos servidores de aplicaciones basados en Axis [1] lo admiten, ya que el soporte del protocolo está implementado en el propio enrutador SOAP.

Un punto de vista muy diferente es el que percibe el cliente, ya que dependiendo de la forma que tenga de invocar el *proxy*, la llamada puede ser síncrona o asíncrona. Es decir, si la aplicación cliente o el *proxy* esperan por la respuesta, aunque ésta llegue en un mensaje distinto que el de la petición, el comportamiento, tal como lo percibe la aplicación cliente, será síncrono, ya que ésta quedará bloqueada hasta que llegue la respuesta.

De lo que acabamos de comentar, deducimos que la construcción de servicios web asíncronos depende en gran medida de la plataforma y de los protocolos sobre los que sean desarrollados. Por otra parte, las aplicaciones cliente deberán estar específicamente preparadas para invocar un servicio web de forma asíncrona. No debemos confundir la asincronía de un servicio web con la asincronía en la invocación dentro de la aplicación cliente. Es posible realizar invocaciones asíncronas dentro de una aplicación (mediante, por ejemplo, el uso de distintos hilos de ejecución). Por el contrario, la asincronía de un servicio web permite que la respuesta a una invocación sea recibida por un proceso completamente distinto del que dio origen a la invocación.

Nosotros ofreceremos la posibilidad de convertir en asíncronos servicios

web que han sido construidos como síncronos. Esto permitirá, por una parte, sin necesidad de desarrollar ningún código fuente, convertir en asíncrono un servicio web que esté funcionando. Por otra, la construcción de servicios web asíncronos será más sencilla, ya que los desarrolladores podrán construir los servicios web como si fueran a ejecutarse de forma síncrona, sin preocuparse además de si la infraestructura tecnológica (protocolos y plataforma software) soporta el uso de servicios web asíncronos.

Nuestra propuesta se basa en el uso de un componente intermedio responsable de gestionar el asincronismo, y dialogar de forma asíncrona con las aplicaciones cliente y de forma síncrona con los servicios web.

2.11. Rendimiento

Cuando hablamos de utilizar Internet como vehículo para la navegación en la web, solemos referirnos siempre al uso de información fundamentalmente estática, como veíamos en la introducción. En determinadas aplicaciones web, parte de la navegación se realiza utilizando información dinámica, es decir, páginas web cuyo contenido varia a medida que son solicitadas. En general, todas aquellas aplicaciones construidas para obtener información del usuario (flujo de información desde el cliente hacia el servidor) utilizarán información dinámica.

La información dinámica, por su propia naturaleza, es información que ha de ser producida en el instante en que es solicitada. Sin embargo, y como mostraremos a lo largo de este trabajo, en determinadas circunstancias puede ser recomendable utilizar técnicas de mejora de rendimiento basadas en políticas de *caching*. Es decir, al igual que los navegadores de web son capaces de mejorar la experiencia del usuario en base a guardar copias locales de archivos que sufren pocas o ninguna modificación (*caching* de páginas web), es posible hacer que las invocaciones a servicios web se beneficien de técnicas similares.

Podría parecer un oxímoron la aplicación de técnicas de *caching* a información eminentemente dinámica, como puede ser una llamada a un servicio web. Sin embargo, no carece de sentido su aplicación en determinados entornos o bajo ciertas circunstancias, como por ejemplo:

- Invocaciones de servicios cuyo tiempo de proceso es muy elevado.
- Invocaciones sobre una infraestructura de comunicaciones pobre o con una elevada latencia.
- Invocaciones a servicios que devuelven una gran cantidad de datos. Esto puede ser aplicable, por ejemplo, a servicios web que utilizan técnicas de *streaming*.

Cualquier invocación dentro de alguno de los supuestos anteriores, será susceptible de ser almacenada en una memoria *cache*. Si esto se produce, cualquier invocación de un mismo servicio web con los mismos datos podrá ser respondida recuperando los datos alojados en una memoria *cache*, evitando de esta forma la invocación hacia el servicio web, y obteniendo así la mencionada mejora de rendimiento y ahorro de recursos.

Las soluciones de *caching* en la actualidad se basan en utilizar recursos de programación ajenos a la tecnología de servicios web. Por ejemplo, en [111] se propone controlar desde la lógica de implementación del servicio la forma en que se generan las cabeceras HTTP de la respuesta a la invocación. En concreto, utilizando mecanismos de programación se fija el valor de la cabecera que indica el tiempo de validez del mensaje en la *cache* del cliente. El problema de este tipo de mecanismos es la dependencia que tienen de la capa de transporte. Esto obliga a que un servicio web que quiera utilizar mecanismos de *caching* como los descritos en [111] deba ser siempre accedido utilizando protocolo HTTP.

En [86] se propone una estructura de control, implementada como un conjunto de clases Java, que ha de ser invocada desde la lógica de implementación del servicio. Dicha estructura se encarga de almacenar mensajes en una *cache* durante unos periodos de validez. Este sistema presenta un problema de rendimiento, ya que las invocaciones han de llegar siempre hasta la lógica de negocio para poder obtener una respuesta almacenada en la memoria *cache*, es decir, los clientes no evitan la invocación y además las respuestas seguirán dependiendo del tiempo de transporte por la red.

Nuestra propuesta para la mejora de rendimiento se basa en la aplicación de dos técnicas distintas para la utilización de memoria *cache*. Por un lado, el uso de intermediarios que nosotros proponemos permitirá realizar *caching* en el sentido tradicional de la web, de forma análoga a la utilizada por los servidores *proxy*. El elemento intermedio podrá analizar cada una de las peticiones de invocación que reciba y, eventualmente, recuperar datos de su memoria *cache* y responder a los clientes evitando realizar la invocación hacia el servicio web original.

Por otra parte, la utilización de nuestra propuesta, la cual incorpora un lenguaje que permite el dialogo de control entre aplicaciones cliente y servicios web, permitirá que las propias aplicaciones cliente realicen *caching* de los mensajes que envían. Esto permitirá obtener mejoras de rendimiento sin necesidad de utilizar elementos intermedios.

Además, la utilización de mecanismos de *caching* podrá ser llevada a cabo con total fiabilidad, gracias a la utilización de técnicas que garantizan la consistencia de los mensajes almacenados en *cache*.

PARTE II

Virtualización

CAPÍTULO 3

Virtualización

3.1. Primera propuesta: Virtualización

En este trabajo mostraremos soluciones para algunos de los problemas pendientes (ver capítulo 2) de la arquitectura de servicios web. Con este objetivo, proponemos el uso de una técnica que ya ha sido utilizada con éxito en otros entornos como las redes IP [93, 136] o el hardware [14, 98, 28]: la virtualización.

La razón principal que nos ha llevado a considerar esta idea parte de la búsqueda de una forma de separar, de una manera estandarizada, el interfaz de los servicios web de su implementación.

Como hemos visto hasta ahora, muchos de los problemas achacables a los servicios web derivan de la estrecha relación que existe entre servicio web y proveedor de servicio web, llegando a confundirse en muchas ocasiones. Esto se debe a que el interfaz de un servicio web, que independiza a los clientes de la implementación, está íntimamente ligado a ambos.

En la parte del cliente existe una dependencia muy elevada debido a los procesos de *binding*, que se realizan generalmente de forma estática y hacen depender las aplicaciones cliente de las estructuras, tipos y nombres fijados por el proveedor.

En el lado de la aplicación servidora, la dependencia es similar, ya que los parámetros de entrada y salida al servicio web deben ser correctamente asignados a las variables o campos que espera recibir dicha aplicación (el componente de negocio).

Tanto en un lado como en el otro, la dependencia se detecta a dos niveles: por un lado, como acabamos de comentar, a nivel de aplicación (cliente o proveedora respectivamente) y por otro, a nivel de *proxy* y *wrapper*, que son

los componentes que hacen de interfaz entre la red y las aplicaciones (cliente o proveedora), por lo que la dependencia en estos componentes es doble: dependen de la mensajería intercambiada por la red y de las estructuras de datos a través de las cuales se comunican con las aplicaciones.

Aunque uno de los valores más admirados de los servicios web es precisamente la independencia entre cliente y servidor, gracias a toda la infraestructura de transporte basada en HTTP y SOAP, vemos que esa independencia puede nos ser tan efectiva como podría esperarse.

A modo de ejemplo, en el ejemplo 3.1 podemos ver un programa servidor que realiza una división entre dos números. En dicho programa se reciben dos parámetros de entrada (`a` y `b`), que representan dos variables enteras. Nuestro servicio de división devuelve además un parámetro de retorno (`cociente`) que es también de tipo entero.

```
  public int divide(int a, int b) {
    int cociente=a / b;
    return cociente;
  }
```

Ejemplo 3.1: Servicio web de división entera.

```
static void Main(string[] args) {
  servicio.Calculadora ws=new servicio.Calculadora();
  int resultado=ws.divide(3,5);
}
```

Ejemplo 3.2: Aplicación consumidora del servicio web de división entera.

En el ejemplo 3.2 podemos ver el listado de un pequeño programa cliente que invoca nuestro servicio web de división (llamado `calculadora`). Para realizar una división, la aplicación cliente rellena los parámetros (`a` y `b`), llama al método `divide` y recibe el resultado de la invocación.

Para publicar el servicio web, y para construir la aplicación cliente, después de haber desarrollado el código del servicio, deben realizarse además unos pasos adicionales:

1. Utilizar un generador de WSDL para construir una descripción de nuestro servicio web que pueda ser utilizada por los clientes. También es posible escribirlo "a mano", aunque es menos habitual.

2. Publicar dicha descripción en algún sitio accesible a los desarrolladores de aplicaciones cliente.

3. Instalar (*deploy*) nuestro servicio en algún servidor web o de aplicaciones que permita su invocación.

4. Utilizar una herramienta que nos permita, a partir de un documento WSDL, generar las clases necesarias (clases *proxy*) para realizar una invocación de nuestro servicio `divide` desde una aplicación cliente (proceso de *binding*).

5. Adaptar nuestra aplicación cliente para que invoque nuestro servicio web.

En los puntos 1 y 4 hablamos de utilizar una herramienta o de escribir "a mano". La utilización de herramientas generadoras de código es algo muy habitual, ya que nos evita tener que realizar procesos repetitivos y ya estandarizados. Si realizamos un cambio en el código fuente de nuestro servicio, deberemos volver a ejecutar la herramienta de generación de WSDL y *wrappers*, y a continuación, en el lado cliente, ejecutar la herramienta de generación de código para reflejar los cambios en el programa cliente, reconstruyendo las clases *proxy*. Por último, adaptar el programa cliente para que funcione correctamente con las nuevas clases *proxy*.

Siguiendo con nuestro ejemplo, supongamos ahora que modificamos nuestro servicio web para que contemple dos requisitos:

- Que las variables de entrada, `a` y `b`, pasen a llamarse `dividendo` y `divisor`, con el fin de facilitar a los desarrolladores de software la utilización de nuestro servicio.

- Para dotar a nuestro servicio de mayor precisión, la variable de retorno (`cociente`) pasará a ser una variable de tipo decimal (`float`).

```
public float divide(int dividendo, int divisor) {
  float cociente=dividendo / divisor;
  return cociente;
}
```

Ejemplo 3.3: Servicio web de division modificado.

```
static void Main(string[] args) {
  servicio.Calculadora ws=new servicio.Calculadora();
  float resultado=ws.divide(3,5);
}
```

Ejemplo 3.4: Aplicación cliente modificada.

En el listado del ejemplo 3.3 podemos ver como ha quedado el programa servidor después de las modificaciones, mientras en el ejemplo 3.4 podemos comprobar el estado del programa cliente.

El cambio en los nombres de las variables afecta fundamentalmente a los mensajes SOAP utilizados para realizar la invocación (ejemplos 3.5 y 3.6), ya que el mensaje SOAP que genere el cliente (serializando las variables del programa) debe coincidir con el formato que espera recibir el servidor (para deserializar el mensaje y "cargar" los datos en las variables del programa servidor). Si los nombres de las variables contenidas en el mensaje SOAP que envía el cliente no coinciden con los que espera recibir el servidor, se producirá un error. Esta situación se produciría, por ejemplo, si modificamos el servidor pero no regeneramos las clases *proxy* del cliente.

El cambio de tipo de dato del parámetro de retorno afecta también a la mensajería SOAP, ya que si modificamos el servidor, y no modificamos el programa cliente (y sus clases *proxy*), obtendremos un error en tiempo de ejecución al no coincidir el tipo de dato recibido (decimal) con el esperado (entero).

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Body>
    <divide xmlns="http://librosa.org/">
      <dividendo xsi:type="xsd:int">5</dividendo>
      <divisor xsi:type="xsd:int">3</divisor>
    </divide>
  </soap:Body>
</soap:Envelope>
```

Ejemplo 3.5: Petición de división escrita en lenguaje SOAP.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

  <soap:Body>
    <divideResponse xmlns="http://librosa.org/">
      <cociente xsi:type="xsd:float">1</cociente>
    </divideResponse>
  </soap:Body>
</soap:Envelope>
```

Ejemplo 3.6: Respuesta a una división escrita en lenguaje SOAP.

A la vista de estos resultados, y a pesar de la independencia que existe

entre cliente y servidor, nos hemos visto obligados a modificar la aplicación cliente debido a los cambios realizados en el servidor.

Supongamos ahora la existencia de un cliente que está utilizando el servicio web de división. Si, por algún motivo, el servicio deja de estar disponible (problemas en la red, con los *routers*, disponibilidad de servidores, gestores de bases de datos, etc.), la aplicación cliente no podrá operar, aún sabiendo que pueden existir en el mercado otros proveedores que hacen la misma operación de división. Para poder utilizarlos habrá que modificar la aplicación cliente, para que contemple reintentos y soporte más de un proveedor.

Supongamos ahora que un cliente que está utilizando el servicio web de división de números decide añadir a sus aplicaciones un segundo proveedor. Para ello, siguiendo la forma estándar de trabajar en la arquitectura tradicional, el cliente deberá modificar sus aplicaciones para que soporten este segundo proveedor (con sus nombres de variables particulares y sus tipos de datos). Además, deberá incorporar a las aplicaciones lógica adicional que permita decidir qué proveedor utilizar en cada momento. En general, cada vez que se añada un proveedor nuevo, se modifique uno que está en uso, o deje de utilizarse un proveedor, habrá que modificar la aplicación cliente.

Como vemos, muchos problemas que pueden surgir con la estructura actual son achacables a la relación tan directa que existe entre cliente y proveedor y, finalmente, entre las interfaces (*proxies* y *wrappers*) y las implementaciones (aplicación cliente y aplicación servidora respectivamente).

Una forma razonable de separar interfaz de implementación, disminuyendo por tanto el acoplamiento entre cliente y servidor, consiste en introducir una capa intermedia que permita independizar uno de otro. Como hemos visto, el cliente se ve obligado a respetar la estructura de la mensajería SOAP, cuyo esquema se extrae del WSDL que describe el servicio. Por su parte, el programa servidor se ve obligado a respetar las variables y tipos de datos que ha publicado, para no afectar a las aplicaciones cliente.

Aquí es donde entra en juego la **virtualización**. Lo que haremos será introducir una capa que virtualice los servicios web que ofrecen los proveedores de servicio. Al mismo tiempo, los clientes de esos servicios dejarán de utilizarlos y pasarán a utilizar las "vistas virtuales" que se construirán sobre los servicios reales ofrecidos por los proveedores.

Esta capa de virtualización debe ser vista como un elemento no intrusivo de la arquitectura, es decir, su introducción debe hacerse respetando, al menos, las siguientes directivas:

- No debe alterar la infraestructura actual, es decir, debe convivir con los lenguajes y protocolos en uso actualmente.

- No debe afectar a los servicios web sobre los que se construya la capa virtual. No debe ser necesario modificar un servicio web para adaptarlo a la nueva arquitectura, sino que ha de ser ésta la que se adapte a los servicios existentes en la arquitectura actual.
- No debe afectar a las aplicaciones cliente. Las vistas virtuales de los servicios utilizados por las aplicaciones cliente deben poder ser utilizadas de la misma manera que los servicios web reales.
- No debe alterar el funcionamiento actual de los proveedores, debiendo permitir que un servicio web forme parte de la arquitectura estándar y de la nueva arquitectura al mismo tiempo.

La aparición de esta capa intermedia se corresponde en cierto modo con una pieza de la arquitectura propuesta por el consorcio: el intermediario. Según se define en el documento de arquitectura del consorcio, un intermediario "es un nodo que procesa mensajes pero que no es necesariamente el destinatario de dichos mensajes, sino que simplemente procesa algunas partes de los mensajes". A los intermediarios se les atribuyen, típicamente, labores relacionadas con la seguridad, como los ya mencionados WS-DBC y WS-Gateway. De alguna manera, los intermediarios especializados en seguridad simplifican la labor de los servidores de servicios web, ya que éstos pueden asumir que trabajan en un entorno seguro y validado. Los intermediarios en la infraestructura de mensajería también pueden realizar otras labores como enrutamiento, proceso de mensajes, proceso de transacciones o envío de confirmaciones.

Nuestra propuesta basada en la virtualización de los servicios web utiliza un intermediario para realizar labores de enrutamiento y proceso de mensajes.

3.2. Segunda propuesta: Nuevos roles

Nuestra propuesta de virtualización conlleva la utilización de una nueva arquitectura, que encaja perfectamente dentro de la arquitectura actual. En esta nueva arquitectura encontramos los siguientes componentes:

- **Cliente**. Al igual que ocurre en la arquitectura estándar, el cliente representa a la entidad que necesita e invoca un servicio. Este cliente se debe entender como un cliente de una estructura SOA, tanto si utiliza servicios web como si no, ya sea software o simplemente organizacional.
- **Cliente Delegado**. El cliente delegado es el agente en el cual el cliente ha delegado la responsabilidad de obtener un servicio. Este es

un componente software que realizará una invocación a un servicio web. Representa, en general, a las aplicaciones cliente.

- **Proveedor**. El proveedor es la entidad que ofrece un servicio. Al igual que el cliente, el proveedor debe ser entendido dentro de una arquitectura SOA, no dentro de una arquitectura estrictamente tecnológica.

- **Proveedor Delegado**. Es la entidad pareja al cliente delegado. Los proveedores de servicio delegan la responsabilidad de ofrecer un servicio en los proveedores delegados.

- **Motor o Intermediario**. Este es el componente más novedoso de la arquitectura, y es el encargado de poner en contacto clientes delegados y proveedores delegados. El motor realiza procesos de comunicación entre ambos componentes siguiendo distintos algoritmos y con objetivos muy distintos dependiendo del entorno en el que sea utilizado.

La razón más importante para extender los componentes de la arquitectura de dos (cliente y proveedor) a cinco, reside fundamentalmente en la separación de roles entre "proveedor de servicio" y "proveedor de servicio web", así como los de "cliente" y "cliente de servicio web". Esta separación tiene su analogía en las arquitecturas orientadas a servicios.

Tomemos como ejemplo una empresa de venta de libros (llamémosla LIBROSA) que dispone de una tienda en la que se atiende al público en general. Supongamos además que la empresa dispone de página web y de un servicio de venta telefónica. Desde el punto de vista de los clientes de LIBROSA el proveedor es siempre el mismo, mientras que los proveedores delegados son: la tienda de venta al público, la venta telefónica y la venta por Internet.

También puede entenderse esta estructura como un único proveedor con más de un canal: es la arquitectura multicanal [39, 97, 127], muy de moda hoy en día entre las empresas con el auge de la web. Según esta forma de entender el problema, el proveedor es uno y único, LIBROSA, y dispone de varios canales por los que puede realizar la venta. Aunque esta aproximación puede ser correcta, a nuestro modo de ver el canal no deja de ser un simple medio de transmisión de información, o incluso de realización de transacciones, pero nada más. La diferenciación entre canales debe ser llevada más allá.

En nuestra propuesta, los proveedores delegan su trabajo en los proveedores delegados, que pueden tomar decisiones, actuar como un "subproveedor". En la arquitectura multicanal los canales no tienen capacidad de decisión, son solamente responsables de dar soporte a múltiples vías de comunicación con los clientes, pero en nuestra propuesta de delegación, los proveedores delegados son algo más que meros canales.

Incluso, cuando hablamos de proveedores delegados dentro del entorno de las aplicaciones software, podremos encontrar más de un proveedor delegado. Siguiendo con nuestro ejemplo, uno de ellos sería un servicio web, y otro posible delegado sería una aplicación web, con la que los clientes realizan directamente las operaciones.

Cada uno de los componentes que acabamos de ver (figura 3.1) juega un rol determinado dentro de la propuesta:

- **Cliente**. Es la entidad que necesita un servicio. Dentro de nuestra arquitectura software el cliente será normalmente una empresa que dispone de una aplicación software que invocará un servicio web.

- **Cliente Delegado**. Es la aplicación del cliente encargada de invocar al servicio web que el cliente necesita para el desarrollo de sus actividades. El "cliente" delega en el "cliente delegado" la responsabilidad de realizar o ejecutar ese servicio. Normalmente el cliente delegado será una aplicación (no importa la plataforma o el lenguaje), en la que se necesita incorporar un servicio externo que será solicitado a través de la invocación de un servicio web. Para ello, el cliente debe realizar una búsqueda de proveedor y, posteriormente, enlazar (*bind*) su cliente delegado con el proveedor delegado que haya encontrado.

- **Motor**. Es el punto central de la arquitectura. Normalmente, será un servidor en el que se publican servicios web que invocarán los clientes delegados, y será el responsable de invocar, a su vez, a los proveedores delegados. Es importante no confundir el papel que el motor juega con el que juega un directorio UDDI o una agencia de descubrimiento (*discovery agency*) [53]. Este último es simplemente un registro en el que los clientes buscan proveedores para sus delegados. Esta búsqueda se realiza de forma estática, en tiempo de desarrollo de las aplicaciones, o incluso antes. El motor tal como nosotros lo entendemos juega un papel dinámico, y entra a formar parte de la arquitectura en tiempo de ejecución.

- **Proveedor Delegado**. Es una pieza de software que se encarga de recibir peticiones de clientes y realizar un servicio que le ha sido delegado por un proveedor. El proveedor delegado es, en nuestro caso, un servicio web. Los clientes que mandan peticiones a los proveedores pueden ser de dos tipos: clientes delegados y motores. Cuando un cliente delegado realiza una invocación a un proveedor delegado decimos que nos encontramos ante la arquitectura estándar. Si quien realiza las peticiones al proveedor delegado es un motor, entonces estaremos hablando de la nueva arquitectura que nosotros proponemos.

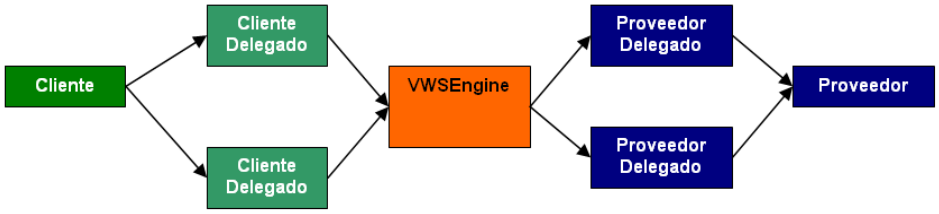

Figura 3.1. Arquitectura virtual.

- **Proveedor**. Es la entidad que quiere ofrecer un servicio y lo hace en base a algún tipo de delegación. Ésta se realiza en un servicio web, es decir, un tipo determinado de proveedor delegado que es el que nosotros vamos a tratar.

A lo largo de la exposición de nuestra propuesta nos centraremos en tres elementos principales: cliente delegado, motor y proveedor delegado, si bien, haremos continuas referencias al cliente y proveedor.

3.3. Tercera propuesta: Arquitectura en red

La arquitectura estándar de servicios web define dos componentes fundamentales: el cliente y el proveedor. Además, un tercer componente es tenido en cuenta, aunque sólo intervenga en tiempo de desarrollo, en el proceso de *binding*, que es el registro, directorio o agencia de descubrimiento.

Como hemos visto anteriormente, en la arquitectura estándar, y en tiempo de desarrollo, el proveedor desarrolla un servicio, construye el WSDL y lo publica en un directorio. A continuación, un cliente que necesita un servicio, lo busca en un directorio y lo agrega a sus aplicaciones. En tiempo de ejecución, la aplicación cliente invoca directamente a la aplicación servidora (figura 3.2)

En nuestra propuesta, que denominaremos **arquitectura en red**, los procesos que tienen lugar en tiempo de desarrollo son muy similares a los de la arquitectura estándar (que podemos denominar **arquitectura punto a punto**). De hecho ambas arquitecturas pueden y deben convivir, es decir, la arquitectura en red que nosotros proponemos puede establecerse sobre componentes que forman parte de la arquitectura punto a punto. Más aún, un proveedor de la arquitectura punto a punto puede, al mismo tiempo, ser proveedor delegado de nuestra arquitectura en red. Por tanto, cualquier aplicación cliente puede invocar proveedores de ambas arquitecturas indistintamente.

En la arquitectura en red, en tiempo de desarrollo, los servicios que

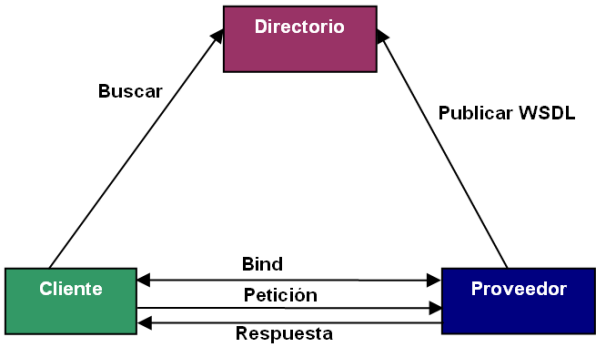

Figura 3.2. Procesos en la arquitectura estándar.

se publican en los directorios son aquellos que se hacen públicos a través del motor, y no los que ofrecen los proveedores delegados. Es importante no confundir el papel del directorio con el del motor. El motor actúa aquí como un punto de invocación de servicios web, un *endpoint* para las invocaciones y, por lo tanto, los servicios que se publiquen en relación con este motor, llevarán un apuntador a la URL del motor. Desde la óptica del cliente, los servicios son buscados igualmente en un directorio y el proceso de *binding* es el mismo. La diferencia más clara entre directorio y motor radica en que el directorio es un componente que es utilizado en tiempo de desarrollo, mientras que el motor interviene en el tiempo de ejecución.

En el tiempo de ejecución es donde encontramos las mayores diferencias entre ambas arquitecturas. En la arquitectura punto a punto, los clientes delegados realizan una invocación que se traduce en el envío de un mensaje SOAP que viaja directamente del *proxy* al *wrapper*. Cuando dicho mensaje es procesado en el proveedor delegado, otro mensaje SOAP viaja desde el proveedor delegado al cliente delegado. Es lo que denominamos una invocación directa.

En la arquitectura en red, los clientes delegados realizan las invocaciones hacia el motor y este, a partir de una configuración determinada que ha recibido previamente, realizará una nueva invocación hacia un proveedor delegado. Esta nueva invocación, de motor a proveedor, generará una respuesta, que será reenviada hacia el cliente delegado desde el motor. Es decir, es lo que podríamos denominar una invocación indirecta, a través de un motor (figura 3.3). Esta invocación indirecta se corresponde en realidad con una invocación directa de un **servicio web virtual**. Los proveedores publican servicios web reales que forman la base para construir servicios web virtuales, que a su vez son los que publican los motores y son los que utilizan los clientes.

El motor de la arquitectura en red puede ser implementado de tres po-

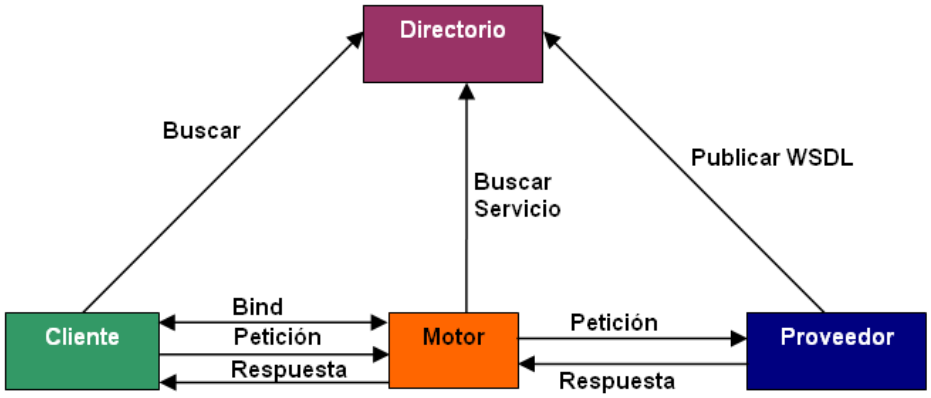

Figura 3.3. Procesos en la arquitectura en red.

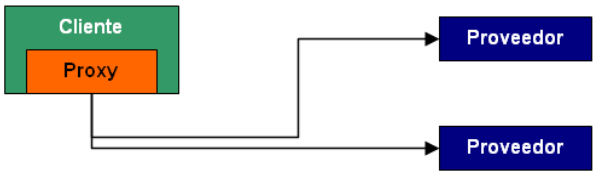

Figura 3.4. Motor simple.

sibles maneras. Por una parte puede ser considerado como una evolución de los *proxies* que utilizan los clientes delegados para realizar las invocaciones. Es lo que denominamos **motor simple** (figura 3.4), y se basa en aplicar más capacidad de decisión a los procesos de invocación realizados por dichas clases *proxy*. Tradicionalmente, la única responsabilidad que tienen estos componentes es la gestión de errores. El resto de sus funciones consiste en ensamblar y desensamblar (serializar y deserializar) mensajes para que sean entendidos por las aplicaciones servidoras.

La segunda posibilidad la constituye lo que denominamos **motor independiente** (figura 3.5). El motor independiente debiera ser implementado típicamente como un servidor independiente o, al menos, un componente de software independiente del cliente y del servidor. El motor independiente actúa como un intermediario entre clientes delegados y proveedores delegados, recibiendo las peticiones de los clientes delegados, que son analizadas para dar lugar a la construcción de una nueva petición que será enrutada hacia el proveedor delegado que corresponda en cada momento. Las respuestas del proveedor siguen el camino inverso.

La forma en que clientes y proveedores ven el comportamiento del motor es completamente distinta. Al contrario de lo que ocurre con los servidores de directorio (que son vistos de igual manera por clientes y proveedores, a

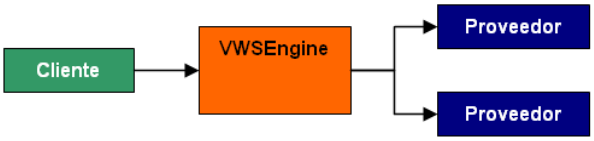

Figura 3.5. Motor independiente.

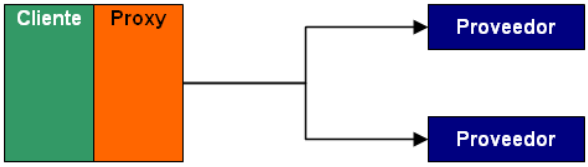

Figura 3.6. Motor integrado.

pesar de realizar acciones bien diferentes sobre ellos), el motor independiente es percibido de manera contrapuesta por clientes y proveedores. Podemos decir que el motor se comporta como un proveedor para los clientes, y a la vez actúa como un cliente para los proveedores.

Una tercera forma de implementar el motor de la arquitectura en red se basa en una combinación de las técnicas utilizadas en los motores simple e independiente. Es lo que denominamos **motor integrado** (figura 3.6). El objetivo principal de esta combinación es aportar las ventajas de cada una de las otras dos soluciones a cambio de perder cierta versatilidad y funcionalidades del motor.

3.4. Motor Simple

Como hemos mencionado anteriormente, el motor simple se basa en la construcción de *proxies* para las aplicaciones cliente que tengan ciertas capacidades de decisión (*proxy* inteligente). Dentro de estas capacidades, y dependiendo del grado de automatización que se desee conseguir, un *proxy* que implemente un motor simple debiera presentar al menos las siguientes características:

- **Detección y gestión de errores** en las llamadas a los proveedores delegados. Esto significa que el *proxy* ha de ser capaz de detectar las situaciones de error y decidir qué hacer a continuación, sin devolver control a la aplicación cliente. Sistemas de reintento deben ser implementados en el *proxy* con el objetivo de mejorar la disponibilidad de las aplicaciones cliente sin aumentar la complejidad de dichas aplicaciones.

- **Selección de proveedor delegado**. Las clases *proxy* han de ser capaces de seleccionar el proveedor delegado que mejor se adecúe a cada petición en cada momento. Múltiples factores pueden afectar esta decisión: tiempo de respuesta, disponibilidad, coste, etc.
- **Modificación de un enlace (*binding*)**. El *proxy* debe permitir, sin necesidad de tener que desarrollar código, modificar el enlace con los proveedores delegados sin modificar las aplicaciones cliente, es decir, éstas siguen utilizando la clase *proxy* aunque hayan cambiado las especificaciones del proveedor delegado. Obviamente, un cambio en las funcionalidades del proveedor delegado o del cliente delegado implicará algún tipo de modificación en las aplicaciones cliente y/o proveedora.
- **Modificación de los esquemas de los parámetros**. Debe permitir modificar los esquemas que siguen los parámetros de ida y de retorno para que coincidan con los esperados por los proveedores (sus tipos, sus nombres, etc.) Al mismo tiempo, el *proxy* debe mantener la compatibilidad de tipos con las aplicaciones cliente.
- **Información de funcionamiento**. Debe generar información que sirva para realimentar el propio *proxy*, de tal forma que automática o manualmente se mejore su funcionamiento.

Lo que se persigue con todas estas características es habilitar un sistema de toma de decisiones de control fuera de la lógica de negocio, es decir, aislar todavía más el desarrollo de las aplicaciones de negocio del sustrato tecnológico sobre el que se asientan, que en este caso son los servicios web.

Otro punto muy importante a tener en cuenta en la construcción de un *proxy* inteligente es la configuración de dicho *proxy*. Si queremos construir un sistema que nos evite modificaciones a las aplicaciones cuando se producen cambios en el servidor o en las directrices del negocio (cambio de proveedores, por ejemplo), necesitaremos disponer de un sistema que permita la configuración del *proxy* desde el exterior, ya sea a través de ficheros de configuración, de una aplicación, o de la forma que sea. Por ejemplo, las políticas de recuperación de errores pueden cambiar en cualquier momento (modificar el número de reintentos de invocación, por ejemplo). O puede ocurrir que queramos añadir proveedores nuevos, realizar modificaciones sobre los existentes, etc.

Todas estas modificaciones deben ser llevadas a cabo desde el exterior de las aplicaciones. Es posible que un cambio en las funcionalidades de las clases *proxy* requiera modificación de dichas clases. Pero cualquier cambio en las funcionalidades de la aplicación cliente (respecto de los servicios web que utilice) debe ir parejo a un cambio en la configuración del *proxy*. La figura 3.7 muestra un esquema de este tipo de motor.

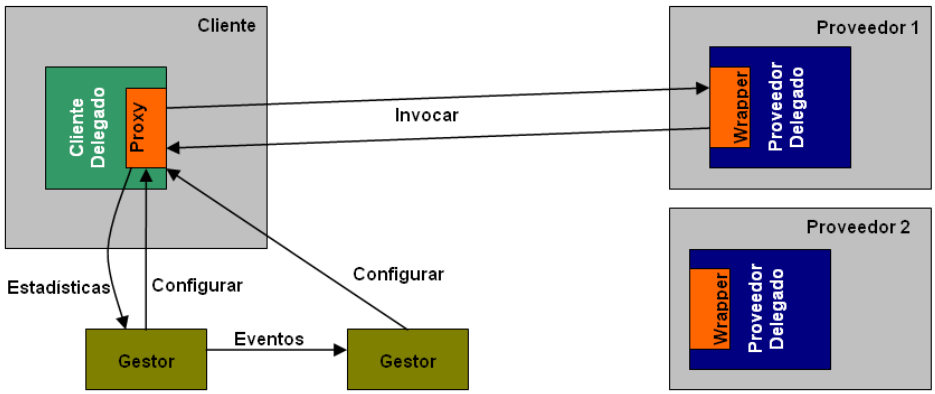

Figura 3.7. Entidades y procesos con el motor simple.

3.5. Motor Independiente

Este tipo de motor reside en un servidor independiente, no estando ligado al proveedor ni al cliente. Podríamos decir que realiza un papel de *broker* entre clientes delegados y proveedores delegados. Este tipo de motor debe incorporar las siguientes características principales:

- Gestión automática de errores.
- Acoplamiento débil entre cliente delegado y motor, y entre motor y proveedor delegado.
- Compatibilidad con la arquitectura actual.

Un punto muy importante a tener en cuenta a la hora de utilizar un motor independiente es la ubicación de dicho motor. Básicamente podemos diferenciar dos ubicaciones: en **red pública** o en **red privada**. La razón de esta diferenciación reside en las implicaciones que ello tiene para la seguridad, y los usos que se hagan del servidor dependiendo de si éste se encuentra en una zona segura (una Intranet, por ejemplo), o en una zona insegura (Internet).

3.5.1. Motor Independiente Público

Hablamos de un motor público cuando nos referimos a un motor independiente situado en Internet, y tanto el cliente delegado como el proveedor delegado acceden a él a través de la red pública (figura 3.8).

Evidentemente esto plantea una serie de problemas relacionados con la seguridad, pero no dejan de ser iguales a los que encontramos en la

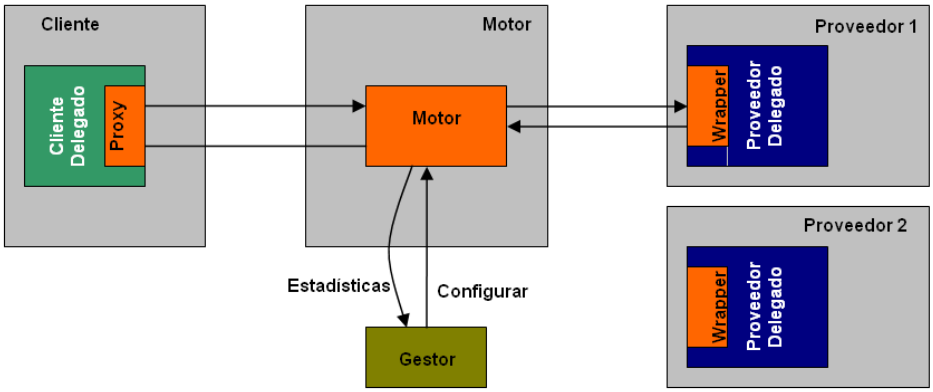

Figura 3.8. Entidades y procesos con el motor independiente público.

arquitectura punto a punto cuando el cliente y el proveedor se conectan utilizando infraestructura de comunicaciones pública.

Un nuevo tipo de problemas aparece a la hora de utilizar este tipo de motores: los problemas de confianza. El mayor interés de este tipo de motores instalados en la red pública es su utilización como *broker* de servicios. De acuerdo a esto, el cliente realiza una invocación al motor, y éste, dependiendo de la configuración que tenga y de los proveedores que conozca, repetirá esta invocación hacia uno de los proveedores. Las decisiones que tome el motor representan un asunto de confianza entre cliente y motor. En comparación con el motor simple, donde las decisiones se toman dentro del proveedor delegado (más concretamente dentro de las clases *proxy*), al utilizar un motor independiente, el cliente delegado debe, a su vez, delegar la responsabilidad de la toma de ciertas decisiones en una entidad distinta de si misma, que será la encargada de gestionar la configuración del motor independiente.

Estas cuestiones de confianza se traducen en problemas a la hora de utilizar un motor u otro, ya que los clientes, aunque no necesitan decidir que proveedor utilizarán, sí deberán decidir que motor utilizarán.

3.5.2. Motor Independiente Privado

Hablamos de motor privado cuando el motor independiente se ejecuta dentro de una red privada, es decir, una Intranet (figura 3.9). En este caso, las funcionalidades que el motor pueda ofrecer dependerán directamente de la parte de la arquitectura en red en la que esté instalado. Podrá instalarse un motor privado en la parte cliente o en la parte del proveedor (o en ambas, por supuesto).

Independientemente de cual de los dos sea el escenario que nos plan-

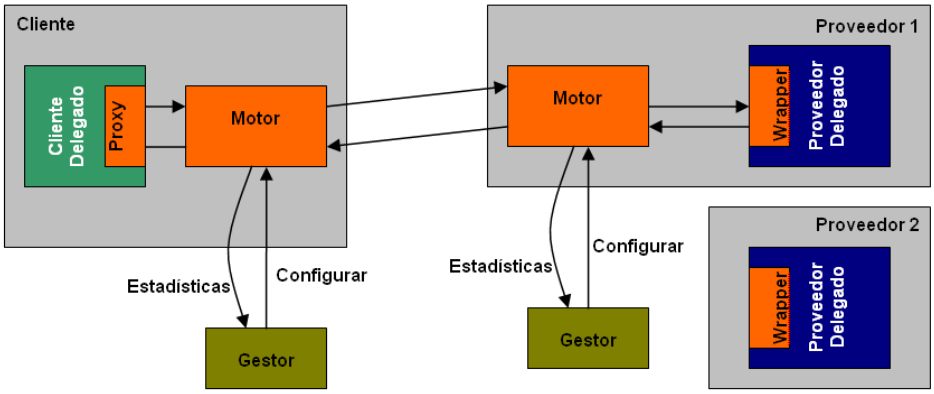

Figura 3.9. Entidades y procesos con el motor independiente privado.

teemos, un motor instalado en una zona privada siempre podrá realizar actividades relacionadas con la seguridad y el acceso a la red pública, ya que tanto en el lado del cliente como en el lado del proveedor, las invocaciones se realizan (o son recibidas) en primera instancia dentro de la red privada, y es el motor el que tiene acceso a la red pública, no los clientes delegados ni los proveedores delegados. Además, dependiendo de la parte en la que se instale, podemos obtener las siguientes funcionalidades:

1. *Instalación en la parte cliente.* En este caso el motor actúa como un *proxy* de servicios web (no confundir ahora con las clases *proxy*). Es decir, actúa de manera análoga a los *proxies* de web, que se encargan de mantener a los clientes en una zona segura mientras ellos realizan las conexiones hacia el exterior de la red privada. Además de realizar ciertas labores de seguridad, un motor de este tipo actúa como un repositorio de servicios invocables, asumiendo que el motor será el único punto de salida al exterior.

2. *Intalación en la parte proveedor.* En este caso el motor actúa como un repartidor de trabajo, ya que las peticiones de los clientes serán recibidas todas en el mismo sitio (en el motor), y este repartirá la carga entre los distintos proveedores delegados que puedan existir en la red privada. Esta funcionalidad ofrece muchas posibilidades en lo referente a la construcción de sistemas de alta disponibilidad (este tema será tratado más adelante en el capítulo 11).

3.6. Motor Integrado

El motor integrado (figura 3.10) representa una solución intermedia entre el motor simple (un *proxy* avanzado) y el motor independiente (un servidor de servicios web). La idea de utilizar un motor integrado parte de la confluencia de dos características de las otras dos soluciones propuestas previamente:

- Por una parte, el motor simple puede no ser suficiente para aplicaciones complejas en las que las necesidades de disponibilidad, de gestión de errores o de calidad de servicio sobrepasen a las ofertadas por los *proxies* inteligentes.
- Por otra, la utilización de un motor independiente, en un entorno de pocos clientes, puede añadir cierta complejidad innecesaria a la arquitectura de servicios web, además de más puntos de fallo respecto a la invocación directa.

Por estos motivos se puede producir un acercamiento entre las dos soluciones ya analizadas. Dicho acercamiento es el origen de una tercera solución en la que los *proxies* adquieren mayor versatilidad y los motores independientes se aproximan a los clientes que los utilizan.

Como resultado, obtenemos los motores integrados, que pueden ser vistos como una implementación en la que las aplicaciones cliente y el motor se ejecutan en la misma máquina. Esto permite:

- Seguir utilizando *proxies* simples, iguales que los que se generan con cualquier herramienta de desarrollo de servicios web.
- Disponer de un sistema fiable y flexible con el que realizar las invocaciones que nos aporta, además, ciertas características de gestión de errores y selección de proveedor.
- Que el acoplamiento entre las aplicaciones cliente y el motor encargado de gestionar las invocaciones sea leve desde el punto de vista del software, ya que se utiliza un interfaz de servicios web entre el cliente y el motor, y no uno basado en componentes (DCOM, EJB, etc.).
- Que a pesar del débil acoplamiento, la solución no presente problemas al aumentar el número de puntos de fallo, ya que la aplicación cliente y el motor dependerán del mismo hardware y software. Desde el punto de vista de las comunicaciones, el acoplamiento, por el contrario, es muy fuerte, al ejecutarse ambos componentes (*proxy* y motor) en la misma máquina.

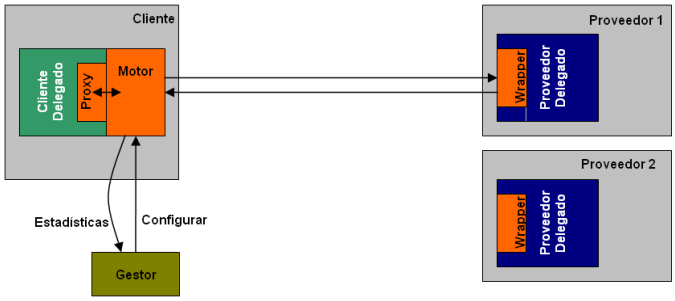

Figura 3.10. Entidades y procesos con el motor integrado.

- Disponer de un sistema seguro, porque las conexiones al exterior (desde la red privada) se realizan a través del motor integrado, nunca desde las aplicaciones.

En el cuadro 3.1 representamos un resumen de las funcionalidades principales de cada uno de los motores (y sus distintas variantes).

	Simple	Indepte. Público	Indepte. Privado Cliente	Indepte. Privado Prov.	Integrado
Gestión de errores	X	X	X	X	X
Selección proveedor	X	X	X	X	X
Enlace diná-mico	X	X	X	X	X
Acopl. débil		X	X	X	
Brokering		X			
Proxy	X	X	X		X
Firewall				X	
Alta disp.		X		X	
Caching		X	X		X
Asincronismo		X	X	X	

Cuadro 3.1. Funcionalidades disponibles según el tipo de motor.

CAPÍTULO 4

Motor Independiente

4.1. Motor Independiente

Como hemos visto anteriormente, nuestra propuesta para la implementación de las técnicas de virtualización en el campo de los servicios web se alcanza mediante el uso de motores. Hemos visto también tres formas de implementar esos motores: simple, integrado e independiente. En este capítulo nos centraremos en analizar la estructura y el funcionamiento del motor independiente.

Lo primero que necesitamos saber acerca de un motor independiente es cómo ha de ser su implementación hardware. Independientemente del sistema operativo utilizado y del resto de software de base, un motor independiente debiera ser implementado como un servidor independiente. Típicamente, una máquina con características de alta disponibilidad (subsistema de discos RAID, doble tarjeta de red, memoria *mirror*, etc.) sería el candidato ideal.

Desde el punto de vista del software, dicho servidor debe contener, al menos, un servidor web y un servidor de aplicaciones. Otras funcionalidades distintas del motor podrán requerir la existencia de un servidor de base de datos, que no tiene por que ser necesariamente local al servidor. La necesidad de un servidor web y uno de aplicaciones, que cada vez con mayor frecuencia encontramos formando una única unidad software, viene dada por la necesidad de instalar una aplicación que se encargue de convertir un servidor de aplicaciones en un motor independiente.

Dicha aplicación es la que realizará las tres labores básicas del motor:

- Ofrecer servicios web virtuales a las aplicaciones cliente.

- Recibir las peticiones de los clientes delegados y procesarlas de acuerdo a unas especificaciones que ha recibido previamente.

- Invocar los servicios reales encargados de implementar el servicio web virtual.

La aplicación que se encarga de realizar estas labores se nutre de documentos VWSDL (serán tratados en el capítulo 7). A partir de estos documentos, el motor generará el WSDL que se publicará para que los servicios puedan ser encontrados y accedidos por los clientes. Al mismo tiempo, los documentos VWSDL contienen la descripción de cómo ha de realizarse la ejecución de un servicio virtual. Dicha descripción se construye a partir de otros documentos WSDL de servicios web estándar (o generados a partir de otros servicios web virtuales).

En su estructura más básica la aplicación que implemente el motor debe incorporar al menos tres componentes (tal como se muestra en la figura 4.1):

- Un manejador de peticiones, que será el encargado de recibir las peticiones de las aplicaciones cliente y reencaminarlas hacia su destinatario.

- Un cliente de servicio web genérico, que será el encargado de reconstruir las peticiones que solicitan los clientes para que puedan ser entendidas por los proveedores delegados. Este cliente no se enlaza en tiempo de desarrollo a ningún proveedor, es genérico y construye las peticiones y extrae los mensajes de respuesta dinámicamente, en tiempo de ejecución.

- Un manejador SOAP, que se encargará de analizar y construir mensajes SOAP que serán intercambiados entre el motor y los servicios web reales que han de ser invocados, así como los mensajes intercambiados entre el motor y los clientes delegados. A menudo este componente suele ser definido como un enrutador SOAP (*SOAP router*).

En realidad, un motor independiente juega un doble rol dentro de la arquitectura de servicios web. Por una parte, el motor es un proveedor delegado para el cliente delegado, ya que dicho cliente realiza invocaciones de servicios hacia el motor (quien utiliza su manejador de peticiones). Por otra parte, el motor es, al mismo tiempo, un cliente delegado, ya que él mismo realiza invocaciones hacia servicios web reales (utilizando el cliente genérico).

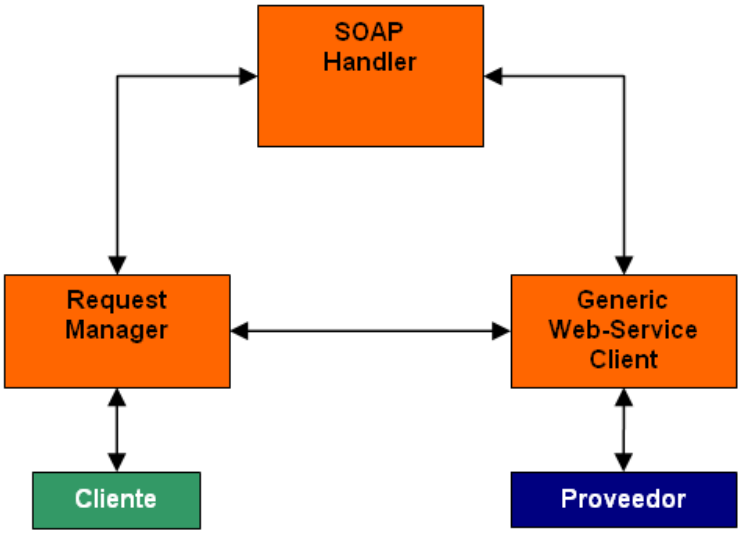

Figura 4.1. Estructura básica de un motor independiente

4.2. Escenarios de uso

Existen tres escenarios básicos en los que un motor independiente puede ser utilizado:

- Instalado en la Intranet del cliente de un servicio web, actuando como *proxy*.
- Instalado en la red privada de un proveedor de un servicio web, actuando como controlador de un *cluster*, es decir, como nodo central de un *cluster* de servidores de servicios web; o bien como **firewall**.
- Instalado en Internet, en una red pública, actuando como **broker** entre clientes y servidores.

4.2.1. El motor en la red privada del cliente

La utilización de un motor independiente como *proxy* de servicios web permite a las entidades utilizar múltiples aplicaciones cliente de servicio web que acceden a la red pública (y al proveedor de servicios web) a través de un punto único de conexión a dicha red. Según se muestra en la figura 4.2, el motor es el único elemento de la red que accede al exterior, en lo que a invocaciones de servicio salientes se refiere.

Utilizando el motor como *proxy* de servicios web obtenemos los siguientes beneficios:

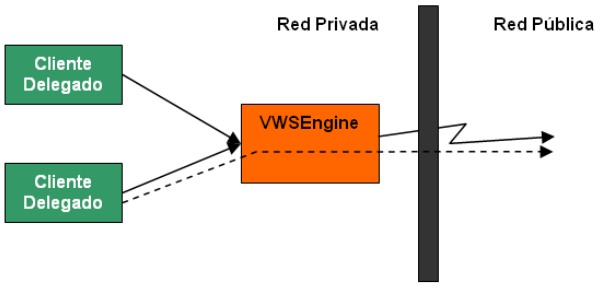

Figura 4.2. Motor independiente en la red privada del cliente.

- Punto único de salida hacia la red pública, simplificando las tareas de gestión y administración de los accesos a Internet.
- Gestión simplificada de la seguridad, ya que sólo es necesario controlar un único elemento de la red privada. Esto afecta tanto a los protocolos de bajo nivel (HTTPS, IPSec, etc.) como a otros niveles superiores (certificados digitales, autenticación HTTP, etc.).
- Posibilidad de realizar *caching* (bajo unas circunstancias concretas) mejorando sensiblemente los tiempos de respuesta.

4.2.2. El motor en la red privada del proveedor

Dentro de este epígrafe debemos tener en cuenta dos posibles formas de actuar: como nodo controlador de clúster y como *firewall* de servidores de servicios web.

Como *firewall*, la arquitectura expuesta en la figura 4.3 muestra un ejemplo, en el cual un motor hace de punto único de entrada a la red privada del proveedor, obteniendo beneficios muy similares a los que los *firewalls* ofrecen como protectores de una red privada corporativa. De acuerdo a esta forma de trabajar, las peticiones son recibidas en el motor, que es el nodo fronterizo de la Intranet, y son reencaminadas a los proveedores de servicios web que funcionan dentro de la red privada, sin acceso desde el exterior. Entre los beneficios más importantes que obtenemos con la función de *firewall* está la protección de la red privada, ya que utilizando un motor, solamente éste debe poder ser accedido desde Internet, mientras que los servidores de servicios web no podrán ser accedidos desde el exterior, al encontrarse instalados en la red privada.

Como controlador de un *cluster*, el motor que recibe las peticiones de los clientes puede repartir la carga según crea conveniente entre todos los nodos proveedores que estén disponibles. Para ello, dicho controlador debe ser dotado de los mecanismos necesarios para conocer el estado de carga

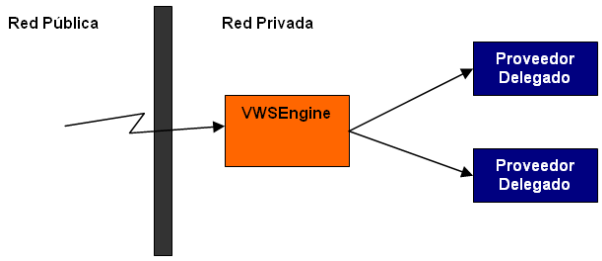

Figura 4.3. Motor independiente en la red privada del proveedor.

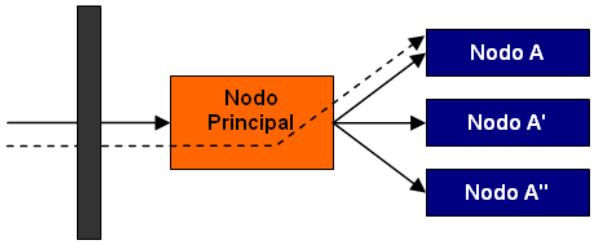

Figura 4.4. En la red privada como controlador de *cluster*

de trabajo del *cluster* y de la capacidad suficiente para poder decidir a qué nodo se le va a distribuir cada petición de invocación recibida. La utilización de un motor como controlador de un *cluster* nos aporta muchos beneficios. El más importante lo representa la capacidad para ofrecer servicios web de alta disponibilidad, donde la implementación del *cluster* se realiza a base de servidores de servicios web (figura 4.4).

Los distintos proveedores delegados ofrecen en realidad los mismos servicios web, que serán posteriormente publicados en el motor independiente, para que puedan ser accesibles desde el exterior del *cluster*.

4.2.3. El motor en Internet

Esta es una de las facetas más interesantes del motor, según la cual, un motor instalado en una red pública pone en contacto a clientes y proveedores en tiempo de ejecución, es decir, actuando como un intermediario independiente (*broker*). En la figura 4.5 podemos ver un esquema de esta posible implementación, donde cliente y proveedor residen en sendas redes privadas, mientras el motor está instalado en la red pública.

Aunque este rol del motor independiente será tratado más adelante, nos gustaría constatar las ventajas de introducir un intermediario entre cliente y proveedor. En primer lugar, la utilización de intermediarios permite a los proveedores llegar hasta muchos más clientes sin necesidad de conocerlos.

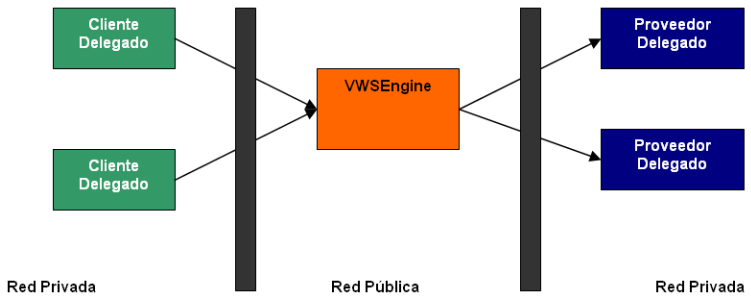

Figura 4.5. Motor independiente en la red pública.

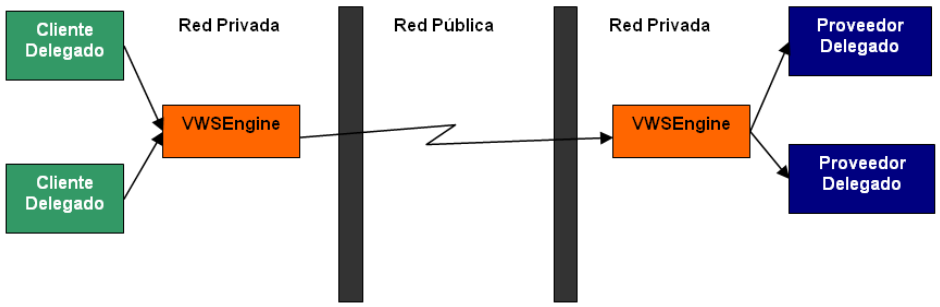

Figura 4.6. Motor independiente en ambas redes privadas.

Desde el punto de vista del cliente, una entidad cliente se beneficia del uso de intermediarios porque aprovecha el valor añadido que éste le ofrece. Dicho valor se puede materializar de muchas formas pero, fundamentalmente, un cliente podrá utilizar distintos proveedores al mismo tiempo sin necesidad de firmar contratos ni desarrollar aplicaciones ad-hoc para cada interfaz que le ofrezca cada proveedor. Las entidades clientes ven simplificado su trabajo al dialogar con un único proveedor: el intermediario.

4.3. Encadenamiento de motores

Las tres posibilidades de instalación que acabamos de describir, representan en realidad distintas funcionalidades obtenidas de la propuesta de nueva arquitectura. Pero estas funcionalidades no son las únicas. Combinando las distintas configuraciones que hemos visto anteriormente podremos crear arquitecturas cada vez más complejas. En la figura 4.6 se muestra un ejemplo de arquitectura que, gracias al nuevo modelo de invocación que proponemos, ofrece elevados niveles de seguridad en clientes y proveedores así como una elevada disponibilidad del proveedor.

Esta arquitectura mejorada se logra utilizando dos motores: uno pa-

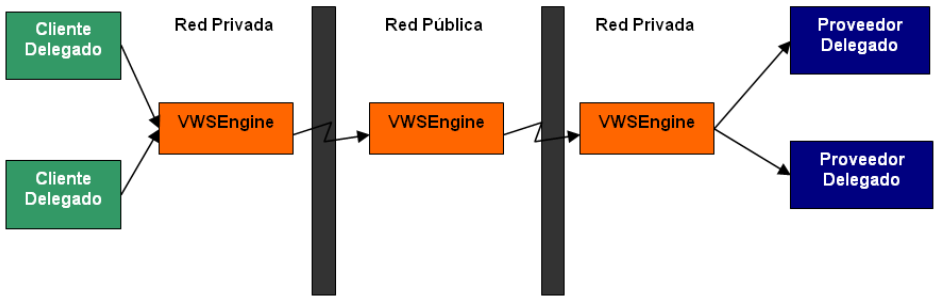

Figura 4.7. Motor independiente en red pública y redes privadas.

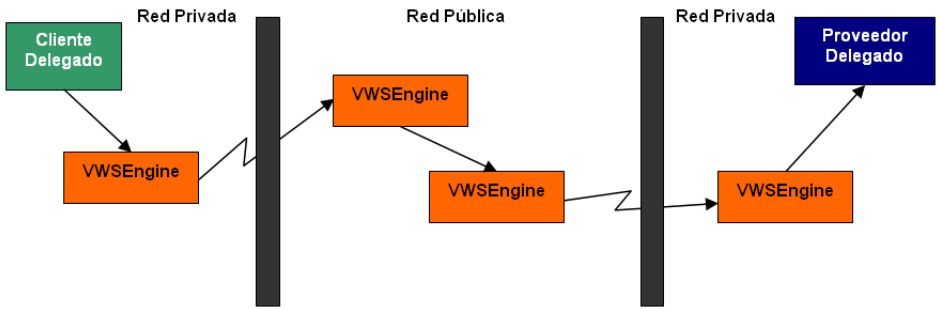

Figura 4.8. Encadenamiento de motores.

ra aislar las aplicaciones del cliente, y otro para aislar los proveedores de servicios web y ofrecer al mismo tiempo alta disponibilidad.

Otra posibilidad muy interesante puede ser obtenida instalando entre cliente y proveedor un tercer motor que actuaría como *broker* entre cliente y distintos proveedores. Esto aumentaría la disponibilidad todavía más, ya que si un proveedor no está disponible por algún motivo, el motor que actúa como *broker* puede en cualquier momento decidir seleccionar otro proveedor sin que el cliente (y sus aplicaciones) tengan que preocuparse de ello. En la figura 4.7 podemos ver un ejemplo de ello.

Teniendo en cuenta que los motores ofrecen servicios web virtuales como si fuesen servicios web normales y que, a su vez, los mismos motores construyen los servicios web virtuales a partir de servicios web estándar, podremos llegar a arquitecturas de servicios web tan complejas como queramos, como el ejemplo que se muestra en la figura 4.8, donde una jerarquía de motores permite construir redes de servidores de servicios web con la arquitectura que se desee.

La utilización de motores como *broker* añade una cuarta entidad a la arquitectura estándar. Además de cliente (y sus delegados), proveedor (y sus delegados), y registro o directorio, encontramos a la entidad *broker*. Dicha

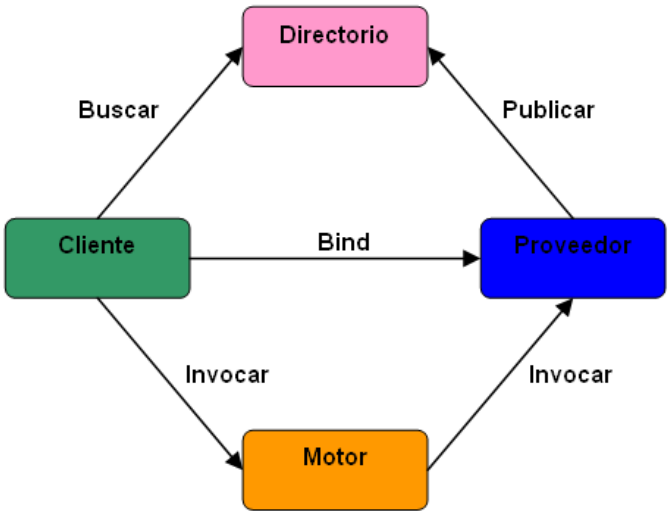

Figura 4.9. Procesos de la arquitectura en red con motor independiente.

entidad es la encargada de poner en contacto clientes con proveedores. La implementación de dicha entidad se realiza también a través de un delegado: el *broker* **delegado**, que es, a la postre, un motor independiente que pone en contacto (en tiempo de ejecución) un cliente delegado (una aplicación cliente) con un proveedor delegado (un servicio web). En la figura 4.9 podemos ver una estructura revisada del funcionamiento de la arquitectura estándar, que es la que se corresponde con nuestro modelo de invocación indirecta, es decir, la arquitectura en red.

CAPÍTULO 5

Ciclo de Vida

5.1. Ciclo de vida tradicional

Para analizar y comparar el ciclo de vida de un servicio tradicional con el de un servicio virtual debemos dividir este ciclo de vida en dos partes: ciclo de vida de desarrollo y ciclo de vida de producción, ya que los procesos que tienen lugar dentro de cada una de las partes son muy diferentes. Adicionalmente, trataremos el mantenimiento de los servicios web (reales y virtuales) en una sección aparte, aunque forme parte del mismo ciclo de vida del servicio.

5.1.1. Ciclo de Desarrollo (Proveedor)

El desarrollo de un servicio web tradicional comienza con la construcción del software de proveedor. Dicha construcción podrá realizarse de dos maneras. Si tenemos una aplicación funcionando y queremos adaptarla a la nueva tecnología de servicios web deberemos iniciar el proceso con la construcción de interfaces para la plataforma en la que se ejecute la aplicación actualmente. Si vamos a desarrollar software de proveedor partiendo de cero, el proceso será ligeramente más sencillo, siempre que utilicemos un lenguaje de programación y una plataforma de ejecución que soporte (de forma nativa o a través de herramientas de desarrollo) la construcción de clases y métodos que puedan convertirse fácilmente en servicios web y puertos respectivamente.

Una vez que tenemos la lógica de negocio debemos proceder a construir los *wrappers*. Un *wrapper* es un componente software que se encarga de recibir peticiones de ejecución desde la web y formatear los datos para que sean entendibles por las aplicaciones servidoras, así como de realizar

el proceso inverso, convirtiendo las estructuras de datos manejadas por la aplicación servidora en datos que serán enviados por la red. A menudo, estos procesos son conocidos como procesos de serialización (el segundo) y deserialización (el primero). Un *wrapper* debe ser un componente que aísle completamente la infraestructura de comunicaciones (TCP, HTTP, XML, SOAP...), permitiendo a los programas servidores ejecutar solamente código relacionado con la lógica de negocio. A parte de esta función de aislamiento, los *wrappers* hacen una función de interfaz, convirtiendo datos que están representados en algún formato interno de un programa escrito en el lenguaje que sea (Java, C++, C#, etc.) en datos representados en un lenguaje estándar (SOAP).

Una vez que disponemos de los *wrappers* debemos construir la descripción del servicio que vayamos a publicar. Para ello escribiremos un documento WSDL que describa los tipos de datos y los métodos que queremos hacer públicos.

Normalmente estos dos pasos (generación de *wrappers* y construcción de un documento WSDL) suelen ser realizados por herramientas automáticas que inspeccionan el código fuente de las aplicaciones servidoras y generan todo lo necesario para construir y desplegar (*deploy*) un servicio web.

Una vez que hemos instalado el servicio web en un servidor de aplicaciones, y disponemos del documento WSDL que describe el servicio, tendremos que publicar ese servicio, normalmente en un directorio UDDI. Este paso dependerá mucho del entorno en el que estemos desarrollando las aplicaciones. Si lo que estamos haciendo es EAI, podremos utilizar cualquier directorio corporativo que se esté utilizando como repositorio de software de la instalación. Si estamos desarrollando servicios web que queremos hacer públicos a través de Internet, entonces tendremos que utilizar un repositorio público, ya sea UDDI u otro cualquiera. Lo importante, sea cual sea el sistema de publicación, es que los clientes que requieran un servicio como el que nosotros queremos ofrecer, lo puedan encontrar de alguna forma.

En la figura 5.1 se muestra un diagrama con las operaciones que se debe seguir para construir un servicio web en la arquitectura estándar.

5.1.2. Ciclo de Desarrollo (Cliente)

El ciclo de vida de una aplicación cliente que utiliza un servicio web comienza en el momento en el que se detecta una necesidad dentro de una aplicación que conduce a la decisión de incorporar un servicio externo.

Lo primero que se debe hacer es buscar el servicio web que mejor se adapte a nuestras necesidades. Para ello, han de tomarse en cuenta muchos factores, no sólo técnicos (rendimiento, disponibilidad,...), sino también

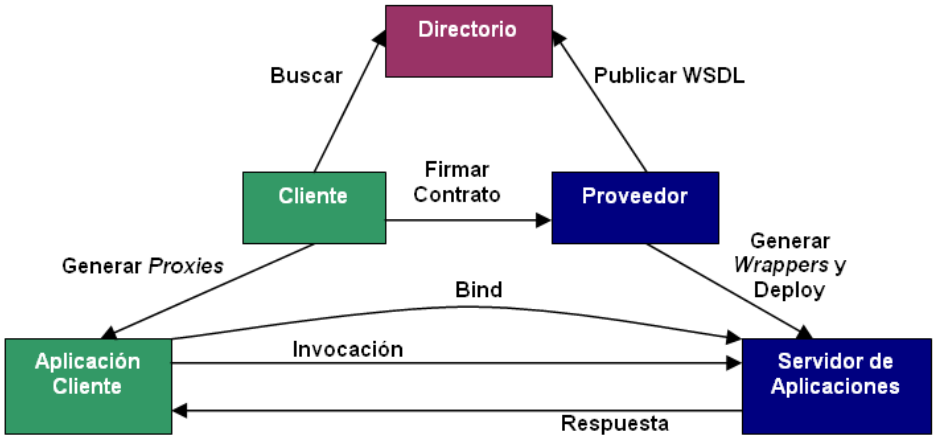

Figura 5.1. Ciclo de desarrollo de un servicio web.

comerciales (coste, calidad,...). Es decir, la búsqueda del servicio debe contemplar factores asociados al servicio web y factores asociados al proveedor de dicho servicio.

Una vez encontrado el servicio, debe iniciarse el proceso de *bind*. Para ello, y normalmente utilizando una herramienta, localizaremos el documento WSDL que describe el servicio que vamos a incorporar a nuestra aplicación. Desde la misma herramienta generaremos las clases *proxy*, que son las que aíslan la aplicación cliente de la infraestructura de comunicaciones.

Una vez que tenemos los *proxies*, ya sea porque hemos utilizado una herramienta o porque los hemos desarrollado nosotros mismos, debemos integrarlos en nuestra aplicación, que es el último paso del ciclo de desarrollo de la aplicación cliente, desde el punto de vista de la utilización de servicios web.

5.1.3. Ciclo de Producción

Este ciclo de vida comienza cuando, durante la ejecución de una aplicación cliente, ésta realiza una invocación a un servicio web.

El primer paso de la invocación se produce cuando la aplicación cliente ejecuta la clase *proxy* pasándole como parámetros los datos de entrada al servicio web. A continuación el *proxy* **serializa** la estructura de datos recibida y la envía por la red. Normalmente, la serialización consiste en la construcción de un mensaje SOAP a partir de los datos, que es enviado a continuación sobre protocolo HTTP hacia un servidor de aplicaciones en el que reside el servicio web en cuestión.

En el servidor de aplicaciones se recibe una petición web que debe ser

analizada y enrutada convenientemente hacia una aplicación web. En la entrada de la aplicación web se encuentra el *wrapper*, que recibe el mensaje SOAP y lo **deserializa**, es decir, reconstruye una estructura de datos que será pasada a la aplicación servidora en el formato en que ésta espera recibirlo.

En la aplicación servidora se ejecuta la lógica de negocio, que dará lugar a una respuesta que deberá ser enviada de vuelta al cliente. Para ello, la aplicación responderá al *wrapper* devolviéndole dicha estructura, y éste la serializará para enviársela posteriormente al cliente (un nuevo mensaje SOAP, esta vez conteniendo la respuesta a la invocación). En el lado del cliente, el *proxy* recibirá la respuesta que deberá ser deserializada para ser entregada a la aplicación cliente en el formato adecuado. Con esto finaliza el proceso de invocación.

Dentro del ciclo de vida de producción deben contemplarse también las cuestiones relacionados con el control y gestión de errores, de tal forma que las aplicaciones cliente sean capaces de detectar los errores y tomar las decisiones oportunas, con el fin de recuperar el estado de normalidad lo antes posible. Una forma muy adecuada de recuperar esa normalidad sería reintentar las invocaciones fallidas siguiendo algún criterio predeterminado, que ha de ser incorporado en el código de las aplicaciones. Por ejemplo, si queremos que la aplicación web invoque un segundo proveedor en caso de no recibir respuesta del proveedor principal (por el motivo que sea, ajeno a la aplicación cliente), deberemos incorporar a la aplicación la lógica necesaria para hacerlo. Además, cada vez que añadamos, modifiquemos y eliminemos un proveedor, deberemos retocar dicha lógica.

5.1.4. Mantenimiento

Lo que acabamos de describir es una simplificación del proceso, que contempla los hitos más importantes dentro de una invocación. Pero este proceso puede sufrir múltiples alteraciones debido al número de puntos de fallo que tiene el sistema. El primer problema que podemos encontrar es una caída del proveedor. Esto puede ser originado, entre otras, por las siguientes causas:

- Problemas en el servidor de aplicaciones (problemas de red, de configuración, de seguridad, etc.), debidos a problemas de disponibilidad en general.
- Problemas en los servidores de componentes (EJB, DCOM, DCE,...)
- Problemas en el gestor de bases de datos.

- Problemas en la aplicación servidora que implementa la lógica de negocio.

Además de estos (y otros) problemas de disponibilidad que puedan surgir, debemos tener en cuenta también un conjunto de circunstancias relacionadas con el propio mantenimiento de la infraestructura y de las aplicaciones. A modo de ejemplo enumeramos algunas posibles situaciones de error:

- Modificación del software del proveedor. Si por algún motivo, el proveedor decide realizar algún cambio en el proveedor delegado nos encontraremos con tres posibles situaciones: modificación de los parámetros (su nombre o su tipo), modificación de los nombres de los métodos (los *ports*), modificación de los parámetros de retorno (nombres y/o tipos). Si alguna de estas circunstancias ocurre, el proveedor generará unas nuevas clases *wrapper*, así como un nuevo WSDL, con lo que, si estas modificaciones no son notificadas a los clientes a tiempo, las invocaciones fallarán irremediablemente.
- Cambios de dirección. Por diferentes motivos, el proveedor puede verse obligado a cambiar el sitio desde el que se ofrece un servicio web, alterando de esta forma la URL en la que ha de ser invocado. Estos cambios de URL deben ser realizados también en los clientes para que las aplicaciones cliente sigan funcionando.
- Modificaciones en los parámetros que rigen la generación de los mensajes SOAP, como pueden ser los espacios de nombres (*namespaces*), los enlaces, etc.

En los ejemplos que acabamos de mencionar, las aplicaciones cliente dejarán de funcionar correctamente, ya que fallarán las invocaciones. Adicionalmente, las aplicaciones cliente deberán incorporar código dentro de los programas que les permita gestionar adecuadamente estas y otras situaciones de error.

Como vemos, existe una gran dependencia entre el ciclo de desarrollo y la buena marcha del ciclo de producción, debido principalmente al acoplamiento que existe en realidad entre clientes delegados y proveedores delegados.

5.2. Ciclo de vida de un servicio virtual

Es muy importante tener en cuenta, en primer lugar, que un servicio virtual no se implementa utilizando lógica de negocio, sino que la implementación se hace a base de otros servicios web (virtuales o no) que son

capaces de implementar una lógica de negocio concreta, que es la que el cliente del servicio virtual necesita. Por lo tanto, los ciclos de vida que aquí trataremos, no tienen relación alguna con los ciclos necesarios para el desarrollo de un servicio web.

5.2.1. Ciclo de Desarrollo

El ciclo de desarrollo engloba todos aquellos procesos relacionados con la creación de un servicio web virtual. Como hemos dicho, nada tiene que ver con la construcción de aplicaciones, sino que se construyen servicios web virtuales a base de servicios web existentes.

En primer lugar, el desarrollador de un servicio web virtual debe localizar todos aquellos servicios web estándar que vayan a ser utilizados como implementación del servicio virtual, que podrá ser solamente uno, o un conjunto de ellos. Para ello, el desarrollador debe localizar a los proveedores y/o proveedores delegados por el mecanismo que sea: diálogo directo con los proveedores, o utilizando algún sistema automatizado de búsqueda como el UDDI.

Una vez que disponemos de una lista de proveedores delegados, lo siguiente que debemos hacer es agregarlos a un nuevo servicio web virtual. Esta agregación puede ser realizada de múltiples formas; como veremos más adelante, puede ser algo tan simple como una lista de proveedores ordenada por orden de preferencia, o puede incorporar especificaciones adicionales en función de las necesidades del desarrollador del servicio virtual:

- Puede incorporarse ciertas características de alta disponibilidad, de calidad de servicio, etc.
- Es posible especificar SLA's que gobiernen el funcionamiento del servicio virtual a la hora de seleccionar proveedor.
- Se puede configurar el servicio virtual a medida del cliente, dependiendo de las funcionalidades que se le quieran dar a dicho servicio (como *proxy*, como *broker*, etc.)

La búsqueda y agregación de servicios forma parte de un proceso que podríamos definir como "implementación" del servicio web. Ahora tendremos que preocuparnos de la construcción del resto del servicio, que será la parte que las aplicaciones cliente vean a la hora de realizar un *bind* o invocar el servicio. Es la parte que podríamos denominar interfaz.

El primer paso en la construcción del interfaz consiste en definir los métodos. En esta etapa decidiremos que métodos queremos hacer públicos para

que puedan ser invocados por los clientes. Por cada método que publiquemos, debemos además especificar qué parámetros recibe y qué parámetros devuelve.

Una vez que hemos definido los métodos, debemos establecer la relación entre interfaz e implementación, es decir, debemos especificar cómo han de asignarse los parámetros de entrada y salida de los métodos que hemos publicado (los métodos virtuales), con los *ports* definidos en los servicios web que hemos seleccionado para la implementación. También es necesario definir los parámetros de entrada y salida a cada método, así como sus tipos de datos.

Los métodos y parámetros publicados en un servicio web virtual son completamente independientes de los que publican los proveedores delegados, por lo que se hace necesaria la existencia de mecanismos de transformación que permitan el paso de parámetros entre clientes delgados y proveedores delegados a través de un motor VWS

En el momento de crear un documento VWSDL, es necesario especificar cómo han de asignarse los parámetros virtuales que se reciban de los clientes delegados a los parámetros que esperan recibir los *ports* de los proveedores delegados (y lo mismo con los valores de retorno). Esta asignación (especificada en tiempo de diseño) será realizada en tiempo de ejecución mediante mecanismos de transformación cada vez que un cliente delegado realice una invocación a un servicio web virtual.

El paso siguiente es enviar la descripción que acabamos de crear (la que define nuestro servicio web) a un servidor de servicios virtuales (un motor). De esta forma, cuando el motor reciba una petición de invocación, sabrá a qué proveedor debe reenviarlo y cómo componer los mensajes SOAP necesarios.

Para hacer que nuestro servicio web virtual sea accesible por aplicaciones cliente que sólo entienden servicios web estándar, debemos generar un documento WSDL, que ha de ser publicado en algún directorio UDDI, al igual que ocurre con los servicios web estándar. Evidentemente, la generación del documento WSDL debería realizarse de forma automática, mediante el uso de hojas de estilo (XSL, *eXtensible Stylesheet Language* [42, 102]) o de cualquier otro mecanismo adecuado para ello. En la figura 5.2 podemos ver todo el proceso de manera esquemática.

5.2.2. Ciclo de Producción

El ciclo de producción es completamente diferente al utilizado en los servicios web estándar, debido al cambio de modelo de invocación que se introduce, pasando de utilizar el tradicional modelo de invocación directa

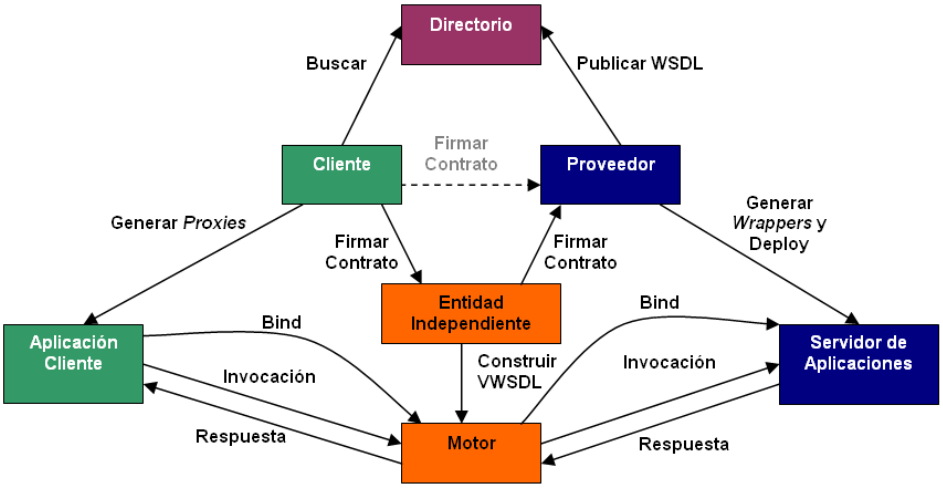

Figura 5.2. Ciclo de vida de un servicio web virtual.

al modelo de invocación indirecta.

El ciclo de producción se inicia cuando un cliente solicita la invocación de un servicio virtual. Para ello, el cliente realiza una invocación estándar, exactamente igual a la que se realiza para invocar un servicio web normal. De hecho, el cliente no sabe que el servicio web es virtual.

Cuando dicha petición es recibida en el motor, lo primero que éste debe hacer es extraer la información necesaria del mensaje SOAP recibido. Datos como la URL del servidor o la acción SOAP a ejecutar (cabecera SOAPAction dentro del protocolo HTTP) deben ser analizados con el objetivo de decidir qué servicio web virtual está siendo invocado. Una vez decidido, el motor debe buscar el documento VWSDL que describe dicho servicio virtual y examinarlo. En función de la información contenida en dicho documento, se seleccionará un proveedor de servicio web real.

El paso siguiente consiste en la creación de un nuevo mensaje SOAP que será enviado al proveedor de servicio que se encargará finalmente de la ejecución. La construcción de este nuevo mensaje se hace de acuerdo con la información contenida en el documento VWSDL que describe la relación entre el servicio virtual y el proveedor del servicio real. Es decir, nombres de parámetros, estructuras, tipos de datos, etc., se hacen coincidir para que la petición enviada al proveedor de servicio web pueda ejecutarse correctamente.

Si todo se ha ejecutado correctamente en el proveedor, el motor recibirá un mensaje SOAP del proveedor de servicio web, que ha de ser deserializado, obteniendo un objeto que contiene la respuesta recibida, y generando a partir de ésta un nuevo mensaje SOAP que contendrá esta vez la respuesta

que se enviará al cliente que solicitó la invocación en primera instancia. Igual que antes, la construcción de este mensaje SOAP se hará teniendo en cuenta las especificaciones contenidas en el documento VWSDL que describe el servicio.

El ciclo de producción se repite continuamente, es decir, durante la vida del motor, éste recibe peticiones que son reencaminadas hacia un proveedor para obtener una respuesta y poder entregársela al cliente que originó la invocación inicial.

5.2.3. Mantenimiento

Existen básicamente dos tipos de mantenimiento que pueden ser realizados sobre un servicio virtual: sobre la implementación y sobre el interfaz.

Las actualizaciones de la implementación se refieren en realidad a las modificaciones en los servicios web que proveen la implementación del servicio web virtual. Como hemos dicho anteriormente, en el mantenimiento de un servicio web estándar, distintos tipos de cambios pueden afectar a los parámetros, a los métodos, a la URL de servicio, etc. Todos estos cambios deben ser tenidos en cuenta a la hora de mantener el motor actualizado. Para ello, las modificaciones que se realicen en los servicios web deben acarrear modificaciones en los documentos VWSDL, pero no en las aplicaciones cliente. Es decir, debemos alterar las especificaciones de cómo se relaciona el servicio virtual con un servicio real que ha sido modificado.

Tomemos como ejemplo un cambio en el proveedor que representa la modificación de los parámetros de entrada a un *port*. Lo único que hemos de hacer en el motor es modificar el documento VWSDL (un documento XML al fin y al cabo), y reflejar la nueva forma de construir los mensajes SOAP que se han de enviar al proveedor, así como una nueva forma de extraer los mensajes que se reciben de éste. Desde el punto de vista del cliente no cambia nada.

Otro ejemplo muy típico de modificación de la implementación de un servicio web virtual lo tenemos en la actualización del grupo de proveedores. Supongamos que queremos añadir un proveedor nuevo, o eliminar uno existente. Lo único que hemos de hacer es, una vez más, modificar el documento VWSDL que describe el servicio virtual.

La actualización del interfaz es la que afecta más directamente a las aplicaciones cliente que utilizan los servicios virtuales. Pero en realidad, la necesidad de modificar el interfaz se minimiza enormemente, ya que la utilización de motores con servicios virtuales permite que los proveedores reales modifiquen los interfaces de sus servicios web mientras los clientes siguen viendo el mismo interfaz que les muestra el motor. Distintos tipos

de modificaciones que requieren cambios en el cliente pueden ser realizados sobre el interfaz del servicio virtual, como añadir un nuevo método a un servicio dado, o modificar los parámetros que espera enviar y recibir la aplicación cliente.

En el caso ya mencionado en que un proveedor de un servicio web estándar decida modificar un parámetro, su tipo, o su nombre, también es posible que el cliente no necesite realizar ninguna modificación, ya que el uso de servicios virtuales nos permitirá, por una parte, crear un nuevo método que contemple las modificaciones realizadas en el servicio web. Por otra parte, gracias a las especificaciones de los documentos VWSDL, podremos mantener publicado el método antiguo, simplemente modificando las reglas que definen la forma en la que se han de construir los mensajes SOAP que se mandan al proveedor. Lo que se consigue es hacer convivir los dos métodos (el antiguo y el nuevo), con el mismo nombre y distintos parámetros (de entrada o de salida), es decir, estamos utilizando una forma de polimorfismo.

5.3. Conclusiones

Como podemos ver, los procesos de desarrollo de un servicio web virtual y uno estándar son completamente distintos. La razón fundamental es que el primero de ellos se construye a base de servicios web reales y, por lo tanto, el desarrollo de la lógica de negocio no forma parte de la construcción de un servicio virtual. En el lado contrario, la construcción de un servicio web estándar requiere, como paso previo, el desarrollo de la lógica de negocio, que será su implementación.

Desde el punto de vista del desarrollo de las aplicaciones cliente, hay pocas diferencias entre el desarrollo utilizando servicios estándar y servicios virtuales. A la hora de programar una aplicación cliente, el desarrollador necesitará escribir menos código si los servicios web que utiliza son virtuales, ya que parte de la gestión de errores o de la gestión de múltiples proveedores, por ejemplo, se podrá hacer dentro del propio servicio web virtual. Por el contrario, las aplicaciones cliente desarrolladas utilizando servicios web estándar deberán incluir código que permita manejar situaciones como las que acabamos de mencionar.

Desde el punto de vista de la producción o explotación de servicios web, no hay diferencia alguna desde la óptica de las aplicaciones cliente, ya que éstas siempre invocan servicios web estándar (en lo que al interfaz se refiere). Sin embargo, la ejecución de un servicio web virtual requerirá siempre una etapa más respecto a la ejecución estándar. Ésta es la que se corresponde con la capa intermedia, la capa de virtualización.

Al igual que ocurre con el desarrollo, el mantenimiento de un servicio web virtual nada tiene que ver con el de un servicio web estándar. En este último, el mantenimiento se refiere a la modificación de la lógica de negocio, mientras que en el primero, el mantenimiento se limita a modificar su definición, agregando o eliminando proveedores y adaptando los interfaces (de entrada y de salida) a los cambios introducidos por los clientes delegados y/o los proveedores delegados.

PARTE III

Servicios Web Virtuales

CAPÍTULO 6

VWS

6.1. VWS

Como ya hemos comentado en capítulos anteriores, un servicio web virtual (VWS, *Virtual Web Service*) se compone de varios servicios web, aunque se ofrece a los clientes como un servicio web estándar. La forma en la que se construyen los servicios virtuales incluye dos actividades principales:

- **Selección**. El proceso de selección es un proceso que ha de ser realizado por humanos. Dicho proceso consiste en seleccionar los servicios web reales que van a implementar el servicio web virtual. Los procesos de búsqueda automatizada de servicios web, que aquí no contemplaremos, están directamente relacionados con lo que se denominan *Semantic Web Services* [108, 58, 59].

  Cuando decimos que uno o más servicios web **implementan** un servicio web virtual nos referimos a la forma en que funcionan y se construyen los servicios virtuales. Un servicio virtual, como ya hemos comentado, hace de interfaz entre uno o varios servicios reales y los clientes, pero en realidad, los servicios virtuales no contienen lógica de negocio, es decir, la implementación de un servicio virtual es un servicio real, mientras que la implementación de un servicio web estándar es, normalmente, un componente que ejecuta la lógica de negocio (figura 6.1)

- **Agrupamiento**. Una vez que disponemos de una serie de servicios web, tendremos que agruparlos para crear el servicio virtual. Distintas formas de agrupación nos llevarán a la obtención de funcionalidades distintas, pero siempre utilizando la misma tecnología de servicios web virtuales.

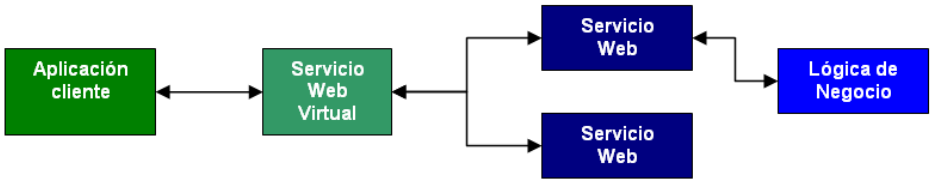

Figura 6.1. Estructura de un servicio web virtual.

El agrupamiento es un proceso que conduce a la creación de documentos que describen al servicio web virtual. Estos documentos han de ser escritos utilizando un lenguaje basado en XML, y cada uno de ellos se utiliza para describir un único servicio web virtual y todos los métodos que éste ofrezca. Distintos tipos de descripciones nos permitirán construir servicios virtuales que ofrezcan alta disponibilidad, que nos permitan realizar composición de servicios, gestión automática de errores o control de la calidad de servicio, por citar algunos ejemplos. La descripción de un servicio virtual nos permite incluso crear servicios que ofrezcan más de una de estas prestaciones al mismo tiempo.

El lenguaje utilizado para describir un servicio web virtual es el VWSDL, *Virtual Web Service Definition Language* y, como hemos dicho, es un lenguaje basado en XML, por lo que los documentos VWSDL han de respetar dicha sintaxis y seguir el esquema propio del lenguaje que nosotros proponemos.

Los documentos construidos utilizando VWSDL cumplen tres funciones principales:

- Sirven como descripción de la implementación del servicio web virtual, ya que estos documentos son los que relacionan lo que es real con lo que es virtual en cada servicio web.
- Sirven como descriptor del despliegue del servicio web (*deployment descriptor*), es decir, los servidores responsables de dar soporte a la explotación de estos servicios (los motores según nuestra arquitectura) conocen, gracias a los documentos VWSDL, la forma en la que han de convertir parámetros, la forma en la que han de seleccionar el servicio web real a utilizar en cada momento, etc.
- Son la base para la construcción de un documento WSDL estándar que contiene la descripción del servicio web virtual que utilizarán los clientes, quienes no distinguirán entre virtual y real, es decir, aquello que para el motor es un servicio virtual, para el cliente es un servicio real.

6.2. Lenguaje de descripción

El lenguaje que nosotros proponemos está basado en XML, al igual que todos los demás lenguajes utilizados dentro de la tecnología de servicios web (SOAP, WSDL, BPEL4WS, etc.). Como todo lenguaje XML, dispone de un esquema propio y los documentos escritos con VWSDL deben seguir este esquema. Dicho esquema es nuevo y ha sido creado dentro de nuestra propuesta de servicios virtuales, es decir, no ha sido concebido como una extensión al WSDL, como ha ocurrido con otros lenguajes dentro de la tecnología de servicios web, sino que es un lenguaje independiente (*stand-alone*) que ha sido diseñado específicamente para un propósito concreto: permitir la definición de servicios web virtuales.

Las principales razones para la creación de un lenguaje completamente nuevo son las siguientes:

- En primer lugar, debemos tener en cuenta la estructura completamente diferente que presenta el WSDL respecto a los servicios que nosotros queremos describir (servicios virtuales). Esto se debe a que los servicios virtuales necesitan muchas definiciones adicionales que nada tienen que ver con la forma en la que se definen los servicios estándar.

- Una de las diferencias más notables entre ambos lenguajes es el usuario final de los documentos. Mientras el usuario de un WSDL es una aplicación cliente, los documentos VWSDL son utilizados por los motores VWS, nunca por los clientes.

- Otra diferencia muy importante tiene que ver con los procesos que tienen lugar cuando un motor VWS procesa un documento VWSDL. El motor debe hacer uso de transformaciones, de variables, de expresiones, de control de alcance, de eventos, etc., que nada tienen que ver con una simple descripción como la que se utiliza en WSDL.

- El *binding* es otra diferencia importante entre ambos lenguajes. En VWSDL no existe, ya que lo motores VWS deben utilizar el *binding* de los servicios reales en un extremo, y ofrecer otro determinado en el extremo del cliente. El enlace ofrecido a los clientes dependerá de las funcionalidades del motor VWS, nunca de los documentos VWSDL.

Los documentos WSDL son utilizados para definir un único servicio web, y dentro de éste se definen puertos (que podemos ver como los métodos de una clase). De la misma forma, en los documentos VWSDL también se definen métodos, que pertenecerán a un único servicio virtual.

Los componentes principales de un documento VWSDL son los siguientes:

- Definición de **tipos de datos**. Dentro de todo documento VWSDL se debe definir los tipos de datos que serán utilizados por el servicio web, tanto parámetros de entrada como parámetros de salida. Es posible crear documentos que no contengan definiciones de tipos, ya que puede crearse un servicio web que contenga métodos que no reciban parámetros, que no devuelvan parámetros, o que utilicen solamente tipos de datos básicos (*string*, *float*, etc.)

- Definición de **proveedores** delegados que podrán ser utilizados en la implementación del servicio web. Es una enumeración de proveedores susceptibles de ser seleccionados para la ejecución de un servicio web.

- Definición de las **transformaciones**. Es un elemento muy importante de nuestra propuesta, que permite especificar las transformaciones a realizar sobre los mensajes SOAP de tal forma que clientes y proveedores puedan comunicarse (realizar invocaciones) utilizando diferentes esquemas de mensajería.

- La parte más importante de un documento VWSDL es la descripción de los **métodos** (puertos según la nomenclatura WSDL). En dicha descripción se detallan todos aquellos métodos que se harán públicos, es decir, los que los clientes utilizarán para comunicarse con el servicio web. La definición de un método es compleja, y podrá contener distintos apartados dependiendo de las funcionalidades que le queramos añadir a cada método de cada servicio virtual.

- Definición del **servicio**, datos globales comunes a todos los métodos del servicio.

CAPÍTULO 7

Lenguaje VWSDL

7.1. Estructura de un documento

Un documento VWSDL es creado con la intención de construir un nuevo servicio web, un servicio virtual, a partir de otros servicios web existentes. El uso primordial de un documento VWSDL es proporcionar a un motor independiente la información necesaria para poder ejecutar servicios web virtuales. Los clientes de servicios web virtuales **no utilizan estos documentos**. En su lugar, ellos utilizan documentos WSDL estándar que son derivados, utilizando procedimientos automáticos, de los documentos VWSDL que utilizan los motores (fig. 7.1).

A pesar de las diferencias ya mencionadas entre WSDL y VWSDL, el lenguaje de descripción de servicios web virtuales presenta ciertas similitudes con el lenguaje estándar de descripción de servicios (el WSDL) como puede ser el uso de las secciones `types` y `service`. El elemento raíz de cualquier documento VWSDL es el elemento `definition`. Dicho elemento

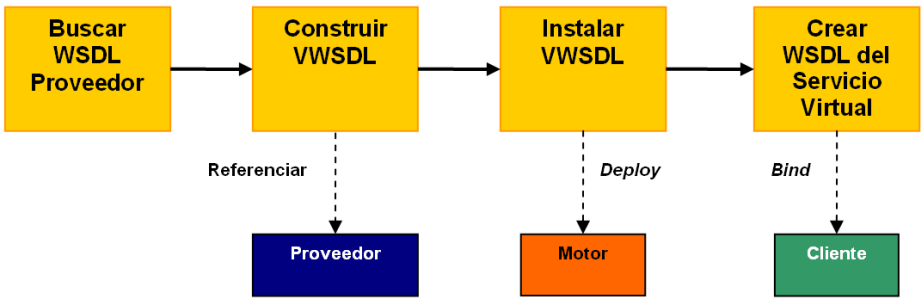

Figura 7.1. Procesos con VWSDL.

contiene los siguientes elementos principales (ejemplo 7.1):

- `types`: Al igual que su homónimo del WSDL, se utiliza para definir tipos complejos de datos (estructuras, matrices, listas, etc.)
- `provider`: Cada documento VWSDL podrá contener uno o más de estos elementos, que se utilizan para definir un proveedor de un servicio web.
- `stylesheet`: Los elementos de este tipo se utilizan para definir hojas de estilo que podrán ser utilizadas en transformaciones de mensajes SOAP, para adaptar la entrada de los servicios virtuales a la entrada de los servicios reales (y lo mismo con la salida).
- `transform`: Los elementos `transform` cumplen la misma función que los elementos `stylesheet`, pero a través de un método diferente. Mientras estos últimos se basan en el uso de hojas de estilo estándar escritas en lenguaje XSL, los primeros se basan en un lenguaje propietario que ha sido diseñado específicamente para realizar transformaciones simples entre documentos XML similares. La razón de ser de este nuevo y sencillo lenguaje es aligerar la carga de proceso que incorporan las hojas de estilo estándar.
- `method`: Es el elemento más importante de cualquier documento, puesto que se utiliza para definir las características del servicio web que se está describiendo. Es equivalente (en funcionalidades) a los elementos `portType` utilizados en los documentos WSDL estándar.
- `service`: Es equivalente a su homónimo del lenguaje WSDL, pero con contenidos bien diferentes. Se utiliza para definir características propias del servicio virtual que se define, que son a la vez características comunes a todos los métodos en él contenidos.

En las secciones siguientes ofrecemos un pequeño análisis de cada una de las secciones principales, en relación a su estructura y su uso.

7.1.1. Elemento `types`

Al igual que ocurre con el lenguaje WSDL, dentro de los documentos VWSDL se pueden definir los tipos de datos complejos que serán recibidos (parámetros de entrada) o devueltos (parámetros de salida) por cada uno de los métodos definidos dentro del servicio. Este elemento contendrá definiciones de tipos que seguirán el esquema XSD (*eXtensible Schema Definition*) [139]. Definir tipos de datos no es obligatorio, pero será conveniente hacerlo cuando la complejidad de la estructura de los parámetros de entrada o de salida sea suficientemente elevada.

```
<?xml version="1.0" encoding="UTF-8"?>
<definition
  targetNamespace="http://dbaccess.sampleproviderha.com/"
  xmlns:tns="http://dbaccess.sampleproviderha.com/"
  xmlns="http://schemas.det.uvigo.es/vwsdl/10"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:vwsdl="http://schemas.det.uvigo.es/vwsdl/10"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <vwsdl:types>
    ...
  </vwsdl:types>

  <stylesheet name="p1ss1i">
    ...
  </stylesheet>

  <provider name="p1"
    location="http://localhost/SampleProviderHA/DBAccess.asmx?WSDL"
    check="false" service="DBAccess" port="DBAccessSoap" >
  </provider>

  <method name="datosCliente" type="equivalent" default="cos1">
    <select name="cos1"
      expression="0.5*reverseAdjust(responseTime)+0.4*availability">
    </select>

    <invoke name="1" type="provider" id="p1" operation="getClient">
      <mapin  type="stylesheet" name="p1ss1i" />
      <mapout type="copy" />
    </invoke>

    <input name="datosClienteRequest">
      <parm name="identificadorCliente"
            element="tns:datosClienteRequest"/>
    </input>

    <output name="datosClienteResponse">
      <parm name="datosCliente"
            element="tns:datosClienteResponse"/>
    </output>

  </method>

  <service name="DBAccess" group="VWSEngine" application="TestHA">
    <event type="connect" iterate="skip" iterations="1" />
    <define name="availability"
            expression="(successful/invocations)*100" />
  </service>

</definition>
```

Ejemplo 7.1: Ejemplo de documento VWSDL.

7.1.2. Elemento `provider`

Cada uno de los proveedores de servicios web que vaya a ser utilizado dentro de la definición del servicio virtual debe ser previamente definido como tal, antes de poder ser referenciado. Los datos que interesan de un proveedor giran en torno al documento WSDL que define el servicio que se va a utilizar en la construcción de un servicio web virtual. La URL del documento WSDL, el nombre del servicio o el puerto utilizado son algunos de los datos necesarios para definir un proveedor.

Los proveedores pueden ser definidos individualmente fuera del punto en el que serán referenciados (elemento `method`) por las siguientes razones:

- **Una cuestión de organización**, manteniendo un lenguaje legible y bien estructurado, facilitando el análisis de los documentos.
- **Una cuestión práctica**. Puesto que un servicio WSDL puede contener varios `portTypes`, y un servicio VWSDL puede a su vez contener varios métodos, es conveniente que la definición de los proveedores se haga por separado, evitando tener que redefinir el proveedor innecesariamente cada vez que se haga referencia a más de uno de sus `portTypes`.
- **El aislamiento de los proveedores en uso**. Dentro de cada documento se dispondrá, de esta forma, de una única referencia aunque se utilicen varios servicios web del mismo proveedor. Así, se verán facilitadas las labores de mantenimiento y análisis que pudieran realizarse eventualmente.

El elemento `provider` es especialmente útil cuando estamos utilizando la tecnología de virtualización para crear servicios virtuales que son exactamente iguales a los reales, es decir, cuando el servicio virtual define varios métodos que son implementados a partir de varios `portTypes` de un mismo servicio web real. Un ejemplo de esto es la construcción de *clusters* de alta disponibilidad que serán analizados posteriormente, cuando se estudien las distintas aplicaciones de la tecnología de virtualización.

7.1.3. Elementos `stylesheet` y `transform`

La utilización de un elemento intermedio dentro de la arquitectura de servicios web obliga a la utilización de un modelo indirecto de invocación. De acuerdo a este modelo, en los motores independientes se reciben peticiones de invocación que son reenviadas a los proveedores delegados, quienes finalmente realizan las ejecuciones de los servicios. Para que este modelo funcione, el motor independiente debe recibir mensajes SOAP de los clientes

delegados y hacer progresar dichos mensajes hacia los proveedores delegados. Es decir, el motor independiente actúa como un intermediario SOAP [94].

Pero de acuerdo a nuestra arquitectura, esa no es la única función del motor cuando actúa como intermediario SOAP. Debido a que el esquema de la mensajería SOAP de los clientes delegados y los proveedores delegados puede ser distinta (según nuestra propuesta), el motor debe ser capaz de realizar las transformaciones oportunas a los mensajes SOAP que recibe (tanto de los clientes como de los proveedores) para enviar a los destinatarios de dichos mensajes los mensajes SOAP que sigan el esquema adecuado, el que espera recibir cada destinatario o receptor.

Dicho de otro modo, el lenguaje VWSDL permite la definición de transformaciones que pueden ser aplicadas a los mensajes SOAP que se reciben de los clientes delegados, para poder enviar a los proveedores los mensajes SOAP en el formato en que esperan recibirlos. Asimismo, los mensajes SOAP que se reciben de los proveedores delegados (respuestas a las invocaciones) podrán ser transformados de acuerdo al esquema que esperan recibir los clientes delegados.

Dichas transformaciones podrán ser definidas (y ejecutadas) de dos formas distintas. Por una parte, y siguiendo la norma de facto que nos conduce a crear lenguajes y sistemas basados en estándares, una transformación podrá ser definida como una hoja de estilo XSL. Para ello, y de forma análoga a como se definen los tipos de datos complejos dentro del elemento `types`, existe un elemento `stylesheet` que permite definir una hoja de estilo (*stylsheet*). Las hojas de estilo estarán asociadas a un método concreto y en un sentido concreto (entrada o salida).

A menudo nos encontraremos con que los esquemas que siguen los mensajes SOAP del cliente delegado y los de los proveedores delegados son muy similares. Es por ello que nuestro lenguaje de descripción de servicios incorpora además un sencillo lenguaje de transformación de documentos XML (mensajes SOAP en nuestro caso). Dicho lenguaje no utiliza patrones (*templates*) como ocurre con el XSL, ni ofrece un amplio conjunto de funciones, ni iteraciones o ejecución condicional. Su principal objetivo es permitir la realización de transformaciones simples de forma rápida y eficiente.

Nuestra propuesta de lenguaje de transformación, que hemos denominado *SimpleTxForm* (Transformación Simple), será explicada con más profundidad en el apéndice A.

Para construir un documento VWSDL que especifique el uso de una transformación basada en *SimpleTxForm* no se utilizará el elemento `stylesheet` (utilizado para referenciar documentos XSL), sino que se utilizará el elemento `transform` para referenciar un documento de nuestro

lenguaje.

El uso de transformaciones dentro de los documentos VWSDL (independientemente del lenguaje utilizado) es algo opcional, que será de especial utilidad cuando el motor independiente sea utilizado como un *broker* entre clientes y proveedores, cuyos clientes delegados y proveedores delegados utilizan distintos esquemas a la hora de producir y consumir mensajería SOAP.

7.1.4. Elemento `method`

Es el componente más importante de cualquier documento VWSDL, ya que la utilidad y funcionalidad de un servicio web virtual vendrá dada por sus elementos `method`, es decir, las operaciones que serán publicadas y ofrecidas a los clientes.

El elemento `method` del lenguaje VWSDL es equivalente a un `portType` del lenguaje WSDL. Ambos son utilizados para definir las operaciones que serán publicadas, así como sus parámetros de entrada y sus parámetros de salida. En realidad, la definición de parámetros es lo único que tienen en común, ya que la definición de un `method` debe incorporar, además de la definición de dichos parámetros, una serie de elementos adicionales que permiten especificar la forma en la que ha de ejecutarse dicho método, la forma en la que se han de transformar los mensajes, la forma en la que se ha de seleccionar el proveedor, etc. La estructura general de la definición de un método es la siguiente:

1. Datos del propio método, como su nombre, por ejemplo.

2. El tipo de implementación que se realizará para ejecutar las peticiones de invocación que se reciban para este método. Los distintos tipos de implementación determinan el modo de funcionamiento de la implementación del método. Por ejemplo, podemos crear métodos que nos permitan ofrecer alta disponibilidad, o que ofrezcan características de calidad de servicio o de control de acuerdos de nivel de servicio.

3. Una expresión que fijará el modo en el que el proveedor ha de ser seleccionado. En aquellos métodos en los que su implementación pueda ser realizada por más de un proveedor, dentro de la definición del método se ha de introducir los elementos `select` necesarios para llevar a cabo dicha selección. Cada elemento de este tipo contendrá una expresión. La selección se basa en el uso de expresiones matemáticas que permiten cualificar y evaluar cada proveedor por separado, obteniendo un valor único para cada uno de ellos, que además es comparable, ya que los resultados de las expresiones tomarán valores dentro de las mismas escalas y unidades.

4. Una lista de operaciones y proveedores que pueden ser invocados para responder a las invocaciones de este método. Con cada posible invocación se define además la forma en la que ha de realizarse la transformación de los mensajes entrantes y salientes, haciendo referencia a un elemento `stylesheet` o `transform`.

5. Los parámetros de entrada.

6. Los parámetros de salida.

7. Información relativa a eventos de error.

8. Información relativa a posibles optimizaciones de las invocaciones en base al uso de sistemas de *caching*.

7.1.5. Elemento `service`

Cumple una función similar a la de su equivalente en el WSDL, que es definir el servicio web como entidad, dándole el nombre correspondiente. A diferencia de aquél, no se establece ningún parámetro del tipo *location* o *binding*, ya que dichos parámetros han de ser establecidos en el momento en que un documento VWSDL es instalado en un motor y un nuevo documento WSDL es construido para ser utilizado por los clientes.

El elemento `service` dentro de un documento VWSDL cumple además una función adicional, que es la de actuar como contenedor de definiciones que serán utilizadas por otros elementos. Parámetros como el número de veces que debe reintentarse una invocación, o la definición de una variable utilizada para representar la disponibilidad de un proveedor, pueden ser establecidos globalmente en este punto, dentro del elemento `service`.

Muchos de los parámetros utilizados en la definición de servicios web virtuales pueden ser definidos globalmente dentro del elemento `service`, o bien, pueden definirse localmente dentro de algunos de los otros elementos que ya hemos mencionado. La forma y orden de precedencia en que han de evaluarse todos esos parámetros definidos local y/o globalmente la determina el alcance o ámbito (*scope*) de las variables.

7.2. Alcance (*scope*)

El concepto de alcance no es algo novedoso en el mundo del desarrollo de software. Un ejemplo muy claro de ello es la utilización de variables locales y variables globales en los lenguajes de programación. De acuerdo a esta diferenciación, y generalizando a cualquier tipo de ítem (no sólo variables), los ítems globales son accesibles desde cualquier parte del objeto dentro del

cual son definidos (una variable global dentro de un programa, por ejemplo). Por otra parte, podemos definir ítems locales que son sólo accesibles dentro del objeto en el que son definidos, que es a su vez un objeto definido dentro del objeto principal (variables locales definidas dentro de una función, por ejemplo).

Al mismo tiempo, y a medida que la jerarquía de objetos definidos aumenta en profundidad (más niveles en dicha jerarquía), los ítems pueden ser a la vez globales y locales. Por ejemplo, una variable A definida dentro de una función será una variable local desde el punto de vista del programa, pero si dentro de dicha función definimos otra función, con sus propias variables locales, la variable será global desde el punto de vista de la función interna, pero será local desde el punto de vista del programa completo.

En nuestro lenguaje hemos incorporado esta característica de la definición de ámbitos, y es utilizable a la hora de definir un tipo determinado de elementos del lenguaje que son comunes a otros elementos distintos del lenguaje. Por ejemplo, en relación al tratamiento de eventos, podemos establecer que si se produce un evento de no-conexión (`noConnect`), se reintente hasta tres veces (ejemplo 7.2). Esta definición de evento se puede hacer de forma global como en el ejemplo, para todo un servicio virtual o de forma individual para cada método que se defina.

```
<provider name="p1"
  location="http://localhost/SampleProviderHA/DBAccess.asmx?WSDL"
  check="false" service="DBAccess" port="DBAccessSoap" >

  <event type="fault" iterate="skip" iterations="1" />

</provider>


<service name="DBAccess" group="VWSEngine" application="TestHA">

  <event type="connect" iterate="retry" iterations="3" />
  <define name="availability"
          expression="(successful/invocations)*100" />

</service>
```

Ejemplo 7.2: Ejemplo de alcance con VWSDL.

Al mismo tiempo, la definición de un elemento puede ser realizada de forma diferente en distintas partes del documento. Si esto ocurre, las mismas leyes que se aplican al uso de variables globales y locales dentro de los lenguajes de programación, deberán ser aplicadas a los elementos que se definan en los documentos VWSDL. Por ejemplo, podemos definir una

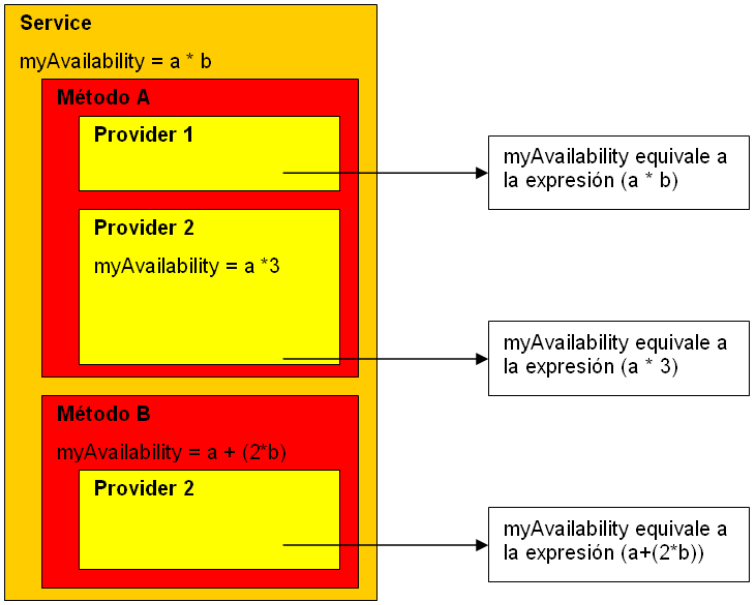

Figura 7.2. Alcance en VWSDL.

variable `myAvailability` dentro del elemento servicio, y la misma variable redefinirla dentro de un método B. De esta forma, el método A utilizará la definición global de `myAvailability`, mientras el método B utilizará su propia definición. O incluso establecer definiciones a nivel de proveedor, como se muestra en la figura 7.2,

La utilización de ámbitos no es aplicable a todos los elementos del lenguaje y, además, no tendría sentido. Sólo aquellos elementos que representan una definición susceptible de ser modificada dentro de algún sub-elemento se verán afectados por el uso de ámbitos. En la versión actual del lenguaje VWSDL, dichos elementos son los siguientes: `cache`, `event` y `define`.

7.2.1. Elemento `cache`

Una de las funcionalidades adicionales de nuestra arquitectura basada en un motor intermedio es la de optimización del rendimiento. En este sentido, nuestro lenguaje permite manejar situaciones en las que el propio motor es responsable de ofrecer una respuesta a una invocación sin necesidad de realizar la correspondiente invocación hacia un proveedor de servicio. Para ello, se puede definir, en diferentes puntos de la descripción de un servicio virtual, intervalos de tiempo en los que el motor actuará como una memoria *cache*. El elemento `cache` permite definir dichos intervalos, y puede ser añadido a un elemento `service` o a un elemento `method`. El orden de precedencia es el siguiente: primero se evalúan los intervalos de *cache* de

método, y luego se les añaden los intervalos de *cache* de servicio. Con los intervalos resultantes se comprueba la validez de los datos contenidos en la memoria *cache*.

7.2.2. Elemento `event`

Para que el motor independiente pueda ser útil como elemento intermedio de la arquitectura, debe ser capaz de tomar sus propias decisiones ante ciertas situaciones que se puedan producir a lo largo de la vida de una invocación. Para ello existe el elemento `event`, que permite especificar cuál ha de ser el comportamiento del motor independiente ante la imposibilidad de llevar a cabo una invocación. La especificación de eventos se realiza de forma jerárquica, al igual que la utilización de la memoria *cache*. Por ejemplo, podemos especificar a nivel de servicio, que si se produce un evento de no-conexión, se reintente con el siguiente proveedor equivalente. Sin embargo, puede ocurrir que para un proveedor determinado (por sus características únicas como proveedor de servicio, o por la razón que el creador de los servicios virtuales considere oportuna), deseemos que ante un evento de no-conexión se reintente dicha conexión un numero finito de veces antes de pasar a *probar* con el siguiente proveedor. En esas circunstancias, el creador del servicio virtual en cuestión ha de incorporar un elemento `event` dentro de la definición del proveedor. Dicho elemento será local a dicho proveedor, y tendrá precedencia ante eventos globales de la misma naturaleza.

Al igual que ocurre con los proveedores de servicios web, podemos especificar comportamientos especiales para un método determinado. Para ello, debemos añadir los elementos `event` necesarios al método que nos interese.

El orden de precedencia será el siguiente: se tienen en cuenta en primer lugar todos los elementos `event` detallados dentro del método que se va a invocar. A estos se le añaden todos los elementos `event` de distinta índole que hayan sido especificados en el proveedor que se va a invocar. Por último, el motor debe seleccionar los eventos globales (especificados a nivel de servicio) que no hayan sido detallados en niveles inferiores. Con todos los eventos seleccionados de los tres ámbitos señalados (método, proveedor y servicio) se crea un conjunto de eventos (*event-set*), que será el conjunto de eventos que el motor independiente manejará para resolver la invocación de un método y un proveedor dados.

Si al llevar a cabo una invocación, ésta es fallida, el motor deberá aplicar los eventos que correspondan, con el objetivo de minimizar el impacto de dicho fallo y poder reintentar la invocación del proveedor.

Cuando el motor se encuentra ejecutando un método dentro de un servicio, los únicos eventos que deben ser seleccionados de nuevo al cambiar de proveedor son, lógicamente, los eventos de proveedor, teniendo en cuenta

que si se reintenta una invocación con un proveedor nuevo, es posible que aparezcan nuevos eventos a tratar que anulen otro eventos que están siendo considerados en un nivel superior (`service`).

7.2.3. Elemento `define`

Como ya hemos comentado con anterioridad, nuestra propuesta se apoya, como uno de sus pilares fundamentales, en el uso de expresiones. Dichas expresiones incluyen variables, constantes, operadores y funciones.

El uso de variables es lo que permite definir, utilizando las expresiones, el comportamiento histórico de un proveedor, pudiendo así obtener una aproximación a su comportamiento en un futuro inmediato. El uso de expresiones se basa en seleccionar, para la siguiente invocación, aquel proveedor que haya presentado, históricamente, un mejor comportamiento.

Las variables utilizadas pueden ser del tipo tiempo de respuesta, disponibilidad, rendimiento, etc., o del tipo coste, calidad del producto, descuentos del proveedor, etc. (Esta peculiaridad de las variables, su origen dispar, será tratada con posterioridad).

En cualquier caso, e independientemente de su origen, las variables que definen el comportamiento de un proveedor son susceptibles de ser utilizadas dentro de distintos ámbitos. Para facilitar su manejo y potenciar la flexibilidad del sistema de expresiones existe el elemento `define`.

Dicho elemento nos permite definir (o redefinir) una variable con un nuevo valor o expresión, que ha de ser tomado en cuenta dentro del ámbito en el que se define y en ámbitos de niveles inferiores.

La definición de variables se hace mediante una asignación de expresiones que, con posterioridad, serán sustituidas convenientemente en aquellas expresiones en las que sean utilizadas. Es por ello que el elemento `define` puede ser utilizado para crear definiciones de dos tipos bien diferenciados:

- Constantes, dando valores concretos a variables.
- Expresiones, permitiendo que la forma en la que se calcula una variable sea diferente para cada servicio, proveedor o método, según convenga.

Por ejemplo, podemos establecer que el valor coste sea, a nivel de servicio (globalmente) un porcentaje de un coste fijo. También podemos redefinirlo para un proveedor determinado, haciendo que, en las expresiones que involucren a dicho proveedor, la variable coste tome el valor fijo 80 (como en el ejemplo 7.3.

La sustitución de variables por sus valores dentro de una expresión que debe ser evaluada se hace, como siempre, siguiendo un orden de precedencia:

```
<provider
   name="p1"
   location="http://localhost/SampleProviderHA/DBAccess.asmx?WSDL"
   check="false" service="DBAccess" port="DBAccessSoap" >

  <define name="cost" expression="80" />

</provider>

<service name="DBAccess" group="VWSEngine" application="TestHA">

  <event type="connect" iterate="skip" iterations="1" />
  <define name="cost" expression="fixedCost * 0.10" />

</service>
```

Ejemplo 7.3: Elemento `define`.

1. Se buscan variables del propio sistema que se encuentren almacenadas en una base de datos.
2. Se buscan las variables definidas dentro del método que se está invocando.
3. Se buscan las variables definidas para el proveedor que se va a invocar.
4. Se buscan las variables definidas en la globalidad del servicio.

A la hora de realizar la sustitución de las variables por sus valores/expresiones, se ha de tener en cuenta dicho orden de sustitución, así como la existencia de los ámbitos correspondientes.

CAPÍTULO 8

Lenguaje VWSEL

8.1. Necesidad de extensión

El uso de motores independientes (motores VWS), junto con la construcción de servicios web virtuales (VWS), constituye una gran oportunidad para otorgar control a estos últimos. Ese control permitirá introducir mejoras en el funcionamiento de la arquitectura de servicios web, como hemos mencionado anteriormente, y analizaremos más adelante con mayor profundidad.

En determinadas ocasiones, dicho control puede ser insuficiente para satisfacer las necesidades del cliente, que se ve imposibilitado para expresar sus necesidades al servicio web (o lo proveedor que lo ofrece). Por ejemplo, cuando aplicamos nuestra tecnología de servicios virtuales a técnicas de QoS, es el motor VWS, siguiendo las indicaciones contenidas en un documento VWSDL, el que decide que características del servicio deben considerarse para cada invocación. Esto es insuficiente desde el punto de vista del cliente, ya que no existe posibilidad alguna de que sea éste quien decida, o al menos influya en la decisión de las características de calidad de servicio que debe cumplir la invocación que se realiza.

Los ejemplos que acabamos de comentar muestran la necesidad de dotar a los clientes de mecanismos de comunicación con los servicios web, tanto en tiempo de invocación, como en instantes previos y posteriores. Estos mecanismos de comunicación no se refieren a información que es intercambiada entre cliente y proveedor de forma estándar, como, por ejemplo, una invocación; sino que se refieren a mecanismos que permiten añadir información de control a los mensajes que se intercambien entre cliente y proveedor.

Este intercambio de información de control no es algo nuevo. El propio esquema de la mensajería SOAP ya fue diseñado para permitir extensiones

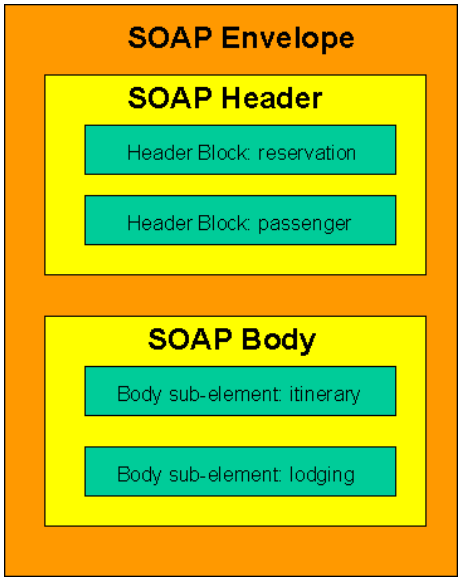

Figura 8.1. Estructura de un mensaje SOAP.

a los mensajes para que éstos puedan contener información de señalización o control, útil tanto para los componentes intermedios que reciban ese mensaje como para el elemento final que lo procese.

Un mensaje SOAP se compone, como todo documento XML, de un elemento raíz (`SOAP:Envelope`) que contendrá además (ver figura 8.1):

- Un elemento de cabecera (`SOAP:Header`), que es opcional y contiene, en general, información relativa al proceso del mensaje SOAP en el que está incluido.
- Un elemento con el contenido del mensaje (`SOAP:Body`), que contiene el mensaje que se desea enviar, es decir, los datos de aplicación.

El elemento cabecera es utilizado habitualmente para enviar información de proceso desde los remitentes de los mensajes SOAP a los destinatarios o a cualquier nodo intermedio que reciba dicho mensaje, tanto si es el destinatario final como si no lo es.

En el ejemplo 8.1 podemos ver un mensaje SOAP que contiene información de direccionamiento (*addressing*) dentro de la cabecera. Dicha información, en este ejemplo, muestra el uso de WS-Addressing [54] para especificar una dirección de respuesta (`wsa:ReplyTo`) utilizada para invocaciones asíncronas; una dirección de destino (`wsa:To`) utilizada para indicar a los nodos intermedios quien es el destinatario final del mensaje; una acción (`wsa:Action`) equivalente a la acción especificada en las cabeceras HTTP de

```
<SOAP:Envelope
      xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">

  <SOAP:Header>
     <wsa:MessageID>
       uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
     </wsa:MessageID>
     <wsa:ReplyTo>
       <wsa:Address>http://host.com/client1</wsa:Address>
     </wsa:ReplyTo>
     <wsa:To>http://otro.host.com/Purchasing</wsa:To>
     <wsa:Action>http://otro.host.com/SubmitPO</wsa:Action>
  </SOAP:Header>

  <SOAP:Body>
     ...
  </SOAP:Body>
</SOAP:Envelope>
```

Ejemplo 8.1: Ejemplo de uso de las cabeceras SOAP.

los mensajes SOAP; y un identificador de mensaje (`wsa:MessageID`) que sirve para poder mantener conversaciones (más de una petición/respuesta encadenadas) entre remitentes y destinatarios de mensajería SOAP, así como facilitar la implementación de sistemas ACID.

La implementación que hacemos dentro de nuestra arquitectura de los mecanismos de extensión (basados en el uso de las cabeceras SOAP) consiste principalmente en:

- Utilización de cabeceras SOAP, bajo las que se incorporará la información de control que sea necesaria. Es decir, las cabeceras son un elemento indispensable para permitir la extensión.

- Un lenguaje que permita realizar especificaciones de control que complementen a la información contenida en los documentos VWSDL. Dicho lenguaje, denominado VWSEL (*Virtual Web Services Extension Language*), permitirá, entre otras cosas, la creación de expresiones que modifiquen el comportamiento del motor VWS a la hora de interpretar la información contenida en un documento de descripción de servicio virtual. Esto permitirá introducir alteraciones en la forma en la que se realizan las invocaciones entre motor VWS y servicios web.

8.2. VWSEL

El lenguaje de extensión de servicios web virtuales (VWSEL) ha sido diseñado como un lenguaje independiente, cuya finalidad es permitir la construcción de expresiones que viajarán dentro de mensajes SOAP, y que serán interceptadas y manipuladas convenientemente por los motores VWS que reciban dichas peticiones.

Se podría haber creado VWSEL como un subconjunto de VWSDL, evitando así la necesidad de crear y gestionar un lenguaje adicional (y su correspondiente esquema). La razón fundamental para la creación de este lenguaje es la sobrecarga en el manejo de los esquemas a la hora de analizar documentos XML. Ya que VWSEL es un lenguaje muy sencillo (no utiliza tipos de datos, y maneja muy pocos tipos distintos de elementos bajo el elemento raíz), el tratamiento de los mensajes SOAP por parte de los motores VWS no se verá penalizado más de lo necesario. Esto es importante debido a la sobrecarga que pueden introducir los motores VWS cuando son encadenados en secuencia, alargando el camino entre cliente delegado y proveedor delegado.

Las expresiones escritas dentro de documentos VWSEL (en adelante expresiones VWSEL), pueden ser creadas con finalidades bien distintas, dependiendo de las necesidades del cliente delegado que las va a enviar. Pueden ser utilizadas como mecanismos de control de la calidad de servicio, como formas de gestionar una arquitectura de publicación/suscripción, etc. Independientemente de la razón por la que las expresiones VWSEL sean utilizadas, todos los clientes delegados deben escribirlas y enviarlas de la misma manera. Esto es:

1. Redactar las expresiones siguiendo el esquema del lenguaje VWSEL.
2. Encapsular dichas expresiones dentro de los mensajes SOAP, utilizando para ello una cabecera (`SOAP:Header`).

8.3. Estructura de un documento VWSEL

En la versión actual de VWSEL (cuya definición se muestra en el apéndice C), un documento debe contener al menos un elemento raíz (`extension`), que debe contener a su vez al menos uno de los siguientes elementos:

- Un elemento `select`, para indicar que la extensión es utilizada para depurar el método de selección del proveedor delegado en el motor VWS.

- Un elemento `cache`, si se desea especificar información de *cache* relativa a la mensajería SOAP.
- Un elemento `async`, si se desea realizar una invocación asíncrona.
- Un elemento `get`, si se desea solicitar la recuperación de un mensaje concreto.

8.3.1. Elemento `select`

Este elemento es la base para la implementación de un buen funcionamiento de un sistema de calidad de servicio con servicios web. El elemento `select` permite a los clientes delegados expresar sus preferencias a la hora de seleccionar un proveedor delegado. Estas preferencias pueden adoptar tres formas, dependiendo de las facilidades que el creador del servicio web virtual quiera otorgar al cliente delegado:

1. La opción más restrictiva. Sólo permite a los clientes especificar un identificador, que en la descripción del servicio web virtual estará asociado con una expresión de selección de proveedor delegado.
2. La opción menos restrictiva consiste en permitir al cliente delegado enviar, junto con el mensaje SOAP a procesar, una expresión que se utilice en la selección del proveedor delegado.
3. Una opción intermedia permite a los clientes delegados expresar un conjunto de preferencias a la hora de preseleccionar los proveedores que entrarán a formar parte del proceso de evaluación.

En la primera opción, representada por el tipo `engine`, el desarrollador del servicio web virtual deberá proveer un conjunto de expresiones alternativas, cada una de ellas asociada a un identificador único dentro del mismo servicio. Idealmente, cada expresión ponderará de distinta manera las bondades y defectos de los proveedores, obteniendo de esta manera un conjunto de clases de servicio. Tomemos como ejemplo el método virtual mostrado en el ejemplo 8.2.

Como podemos ver en la definición del método `datosCliente`, existen dos formas de seleccionar el proveedor delegado (cada una de ellas contenida en un elemento `select`). Cada elemento `select` lleva un identificador único (especificado en el atributo `name`).

Como vemos, en la expresión identificada con el nombre `cos1`, el creador del servicio web virtual ha dado unos pesos similares a la disponibilidad y el tiempo de respuesta, y ha incluido una variable de usuario (`confidence`), que representa la confianza que el usuario tiene en un proveedor concreto.

```
<method name="datosCliente"
        type="equivalent"
        default="cos1"
        extensionClient="false"
        extensionEngine="true"
        extensionRestrict="true" >

  <event type="fault" iterate="retry" iterations="3" />

  <select name="cos1" type="numeric"
          expression="0.6*adjust(availability)+0.4*g" >
    <define name="g" type="numeric"
      expression=
      "(reverseAdjust(responseTime)+adjust(confidence))/2"
    />
  </select>
  <select name="cos2" type="numeric"
          expression="reverseAdjust(responseTime) " />

  <cache from="T00:00:00" to="T23:59:59"
         interval="300" check="data" />

  <invoke name="provedorUno" provider="p1" operation="DBAccess">
    <mapin />
    <mapout name="p1ss1o" type="stylesheet" />
  </invoke>

  <invoke name="proveedorDos"
          provider="p2" operation="nombreOperacion">
    <mapin />
    <mapout name="transformacion2Out" type="transform" />
  </invoke>

  <invoke name="proveedorTres"
          provider="p3" operation="originalOperName">
    <mapin />
    <mapout name="transformacion2Out" type="copy" />
  </invoke>

  <input name="nombreMensajeEntrada">
    <parm name="dni" element="tns:obtenerDatosClienteRequest"/>
  </input>

  <output name="nombreMensajeSalida">
    <parm name="DatosCliente"
          element="tns:obtenerDatosClienteResponse"/>
  </output>

</method>
```

Ejemplo 8.2: Ejemplo de método virtual.

En la expresión de nombre `cos2`, sólo se tiene en cuenta el tiempo de respuesta histórico del proveedor.

Cada una de las dos expresiones que acabamos de describir, como veremos más adelante al hablar de calidad de servicio, representa lo que en nuestra propuesta denominamos una **clase de servicio**.

La clase `cos1` debería ser utilizada por aquellos clientes delegados cuyo mayor interés (a la hora de realizar una invocación) se centre en la disponibilidad. La clase de servicio `cos2` sería la que deberían utilizar aquellos clientes delegados preocupados fundamentalmente por el tiempo de respuesta de la invocación.

Un cliente que quisiese invocar el servicio web del ejemplo anterior, y desease utilizar una clase de servicio que le ofreciese la máxima disponibilidad, debería enviar un mensaje SOAP similar al mostrado en el ejemplo 8.3.

```
<SOAP:Envelope
      xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
      xmlns:vwsel="http://schemas.det.uvigo.es/vwsel/1.0">

  <SOAP:Header>
    <vwsel:extension>
      <vwsel:select type="engine">cos1</vwsel:select>
    </vwsel:extension>
  </SOAP:Header>

  <SOAP:Body>
     ...
  </SOAP:Body>

</SOAP:Envelope>
```

Ejemplo 8.3: Cabecera SOAP con información de control VWSEL.

Si el desarrollador de un VWS quiere dar a los clientes delegados una oportunidad de influir en el proceso de selección, con una mayor flexibilidad que la obtenida con las clases de servicio, pero sin permitirle modificar las expresiones, podrá optar por crear documentos VWS que incorporen expresiones que permitan a los clientes especificar un conjunto de condiciones previas que los proveedores delegados candidatos deben cumplir para poder ser elegidos.

En el ejemplo 8.4 podemos ver un fragmento con un ejemplo de ello, donde la expresión utilizada para seleccionar un proveedor delegado especifica que los candidatos deben cumplir dos condiciones antes de poder ser evaluados: su disponibilidad debe ser superior al 98 %, y su tiempo de respuesta

inferior a 20 milisegundos.

```
...
<vwsel:select type="restrict">
  (availability>98) & (responseTime<20)
</vwsel:select>
...
```

Ejemplo 8.4: Ejemplo de documento VWSEL con restricción.

Por último, si el creador de un documento VWS quiere dar plena libertad a los clientes delegados para que ellos mismos decidan el proceso de selección, deberá especificarlo en el documento VWS tal como se muestra en el ejemplo 8.2. Por su parte, los clientes delegados deberán enviar una expresión completa dentro de la cabecera del mensaje SOAP, tal como se muestra en el fragmento del ejemplo 8.5.

```
...
<vwsel:select type="client">
  0.5*adjust(availability) + 0.5*reverseAdjust(responseTime)
</vwsel:select>
...
```

Ejemplo 8.5: Ejemplo de documento VWSEL con expresión de cliente.

Como acabamos de ver, el creador de un documento VWSDL es el responsable de decidir qué opciones tendrán los clientes a la hora de utilizar alguno de los mecanismos que acabamos de describir. La forma que tiene de hacerlo es a través de la definición del método virtual.

Como podemos ver en el ejemplo 8.2, existen tres atributos, denominados `extensionEngine`, `extensionRestrict` y `extensionClient`, que permiten decidir si se habilita la selección de proveedor dirigida por el cliente mediante clases de servicio, restricciones o expresiones, respectivamente. Cada uno de los tres atributos tiene un valor por defecto de "`false`", y puede ser fijado a "`true`" de forma independiente.

8.3.2. Elemento `cache`

La utilización de nuestra propuesta como un sistema de mejora de rendimiento será tratada en el capítulo 13. Como comentaremos más adelante, existen distintos mecanismos de *cache* para distintos entornos. En el entorno más sencillo, la *cache* es especificada solamente en los documentos VWSDL, a modo de intervalo de validez de los mensajes, tal como se muestra en el ejemplo 8.2, en el elemento `cache`.

Otros mecanismos de *caching* en los que no intervienen elementos intermedios son también ofrecidos por nuestra arquitectura. Dichos mecanismos

requieren de la existencia de un diálogo entre clientes delegados y proveedores delegados. Debido a esto último nuestra arquitectura se basa en el VWSEL para permitir la comunicación de información de control entre ambas entidades.

Nuestra propuesta de *caching* se basa en tres modelos diferentes que aportan distintos grados de consistencia respecto de la información almacenada en la *cache* de los clientes. Cada cliente especifica, utilizando VWSEL, el tipo de consistencia que va a utilizar. Esto se hace utilizando el elemento `cache` del lenguaje VWSEL, tal como se muestra en el ejemplo 8.6.

```
<SOAP:Envelope
    xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:vwsel="http://schemas.det.uvigo.es/vwsel/1.0">

  <SOAP:Header>
    <vwsel:extension>
      <vwsel:cache type="secure">
        <wsa:From>http://a.client.com/application</wsa:From>
      </vwsel:cache>
    </vwsel:extension>

  </SOAP:Header>

   <SOAP:Body>
      ...
   </SOAP:Body>

</SOAP:Envelope>
```

Ejemplo 8.6: Ejemplo de petición SOAP con información de *cache*.

El elemento `cache`, y en general todo el lenguaje VWSEL, incorpora muy poca información, y, por lo tanto, genera una sobrecarga (*overhead*) muy pequeña a la hora de ser procesado. El elemento `cache`, cuando es enviado por un cliente delegado, sólo lleva información del tipo de *caching* que se va a utilizar y, cuando sea necesario, una dirección de notificación de cambios en la *cache*.

El mensaje SOAP de respuesta a una petición como la que acabamos de ver es también muy sencillo. En el ejemplo 8.7 podemos ver un ejemplo de ello, en donde el elemento `cache` sólo incorpora información relativa a la validez del mensaje de respuesta (si se puede almacenar en *cache*, su durabilidad, etc.)

```
<SOAP:Envelope
    xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:vwsel="http://schemas.det.uvigo.es/vwsel/1.0">

  <SOAP:Header>
     <vwsel:extension>
       <vwsel:cache enabled="true" interval="600">
         <wsa:MessageID>
            uuid:aaaabbbb-cccc-dddd-eeee-wwwwwwwwwww
         </wsa:MessageID>
       </vwsel:cache>
     </vwsel:extension>
  </SOAP:Header>

  <SOAP:Body>
     ...
  </SOAP:Body>

</SOAP:Envelope>
```

Ejemplo 8.7: Ejemplo de respuesta SOAP con información de *cache*.

8.3.3. Elemento `async`

Nuestra propuesta de arquitectura para los servicios web permite la invocación asíncrona de servicios. Esto se consigue gracias a la utilización de VWSEL. Para que un cliente delegado pueda realizar una invocación asíncrona hacia un proveedor delegado, será necesaria la utilización de uno de dos posibles mecanismos de implementación de asincronismo:

- Invocación directa. El cliente delegado envía una invocación asíncrona al proveedor delegado. Ambos deben entender el lenguaje VWSEL para que esto sea posible. Además, para que la respuesta a la invocación pueda ser enviada de forma asíncrona, el agente debe ser accesible desde el proveedor, desde el punto de vista del protocolo utilizado para las invocaciones (HTTP, actualmente).

- Invocación indirecta. Un cliente delegado realiza una invocación hacia un elemento intermedio. Este elemento se encarga de invocar a un proveedor delegado. Cuando tiene la respuesta, invocará al cliente delegado que inició el ciclo. Es decir, mediante este mecanismo podemos hacer que un cliente delegado realice una invocación asíncrona hacia un proveedor delegado que no soporta asincronismo, gracias a la utilización del elemento intermedio.

El elemento `async` permite especificar dentro de un documento VWSEL si se desea realizar una invocación asíncrona. Pero, además, permite especificar el tipo de invocación que se realizará: si se utilizará un sistema de retro-llamada (*callback*) o un sistema de notificación (*notify*).

Aunque el tratamiento del asincronismo será tratado específicamente en el capítulo 14, mostraremos aquí algún ejemplo de utilización del VWSEL para implementar asincronismo en las invocaciones. Así, en el ejemplo 8.8 podemos ver una petición de invocación que especifica que se desea realizar una invocación asíncrona y recibir el mensaje en cuanto esté lista la respuesta.

Como respuesta a una invocación asíncrona, los clientes delegados recibirán un mensaje con una cabecera VWSEL conteniendo un identificador de mensaje. Éste es el que garantiza la unicidad del mensaje de respuesta, y permitirá al cliente delegado lanzar varias peticiones de invocación antes de recibir las respuestas.

```
<SOAP:Envelope
    xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:vwsel="http://schemas.det.uvigo.es/vwsel/1.0">

  <SOAP:Header>
    <vwsel:extension>
      <vwsel:async type="callback" response="message">
        <wsa:ReplyTo>
          <wsa:Address>http://host.com/clnt</wsa:Address>
        </wsa:ReplyTo>
      </vwsel:async>
    </vwsel:extension>
  </SOAP:Header>

  <SOAP:Body>
     ...
  </SOAP:Body>

</SOAP:Envelope>
```

Ejemplo 8.8: Ejemplo de petición SOAP con información de asincronismo.

Una vez que el proveedor delegado ha procesado la petición ya puede enviar el mensaje SOAP conteniendo la respuesta. En el ejemplo 8.9 podemos ver una muestra de un mensaje de respuesta enviado por un proveedor delegado. Como vemos, en el documento VWSEL se indica el identificador único del mensaje de petición al que corresponde la respuesta que se envía. En el cuerpo del mensaje SOAP el cliente delegado encontrará el mensaje de respuesta a la invocación inicial.

```
<SOAP:Envelope
    xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"

    xmlns:vwsel="http://schemas.det.uvigo.es/vwsel/1.0">

  <SOAP:Header>
    <vwsel:extension>
      <vwsel:async type="message">
        <wsa:MessageID>
          uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
        </wsa:MessageID>
      </vwsel:async>
    </vwsel:extension>
  </SOAP:Header>

  <SOAP:Body>
     ...
  </SOAP:Body>

</SOAP:Envelope>
```

Ejemplo 8.9: Ejemplo de respuesta SOAP a una invocación asíncrona anterior.

Más detalles acerca del protocolo a seguir por cliente y proveedor y de las distintas formas de comunicación asíncrona serán mostrados en el capítulo 14

8.3.4. Elemento `get`

Como acabamos de ver, VWSEL permite gestionar de forma avanzada el intercambio de mensajes entre clientes delegados y proveedores delegados (ya sea a través de un motor VWS o no). Este intercambio de mensajes permite, de diversas formas, que los proveedores notifiquen a los clientes el hecho de que un nuevo mensaje está disponible para ser recibido y los clientes, mediante el mecanismo que vamos a explicar, soliciten la recepción o recuperación del mensaje.

Para permitir la recuperación de mensajes independientemente del origen de los mismos (notificación de invocación asíncrona finalizada, actualización de mensajes SOAP en *cache* del cliente, etc.), hemos creado la primitiva `get`, que permite a un cliente solicitar un mensaje mediante su identificador único.

En el ejemplo 8.10 podemos ver un mensaje SOAP enviado por un cliente para recuperar un mensaje SOAP concreto.

```
<SOAP:Envelope
    xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:vwsel="http://schemas.det.uvigo.es/vwsel/1.0">

  <SOAP:Header>
    <vwsel:extension>
      <vwsel:get>
        <wsa:MessageID>
          uuid:FGTR567A-27FB-0237-B31D-0000000000DA
        </wsa:MessageID>
      </vwsel:get>
    </vwsel:extension>
  </SOAP:Header>

  <SOAP:Body>
     ...
  </SOAP:Body>

</SOAP:Envelope>
```

Ejemplo 8.10: Ejemplo de petición de recuperación de un mensaje.

Sólo existen dos tipos de respuestas a un mensaje de petición de recuperación de mensaje por parte del proveedor del mismo:

- Un mensaje de error (un SOAP *fault*) si hay algo incorrecto en la petición o si el mensaje no puede ser recuperado (identificador erróneo o errores en el acceso, por ejemplo).
- Un mensaje SOAP conteniendo el mensaje solicitado en el cuerpo.

CAPÍTULO 9

Arquitectura de un motor

9.1. Arquitectura

Aunque no forma parte de nuestra propuesta, ya que la implementación es algo que se debe dejar en manos de las entidades interesadas en implementar nuestra solución, nos gustaría hacer algunos comentarios acerca de la forma en la que el motor VWS debería ser implementado. Las ideas expuestas en este estudio sobre la implementación del motor se basan en el conocimiento que tenemos sobre la arquitectura a la vez que se apoya en la experiencia de haber desarrollado un prototipo que implementa las funciones básicas del motor.

De acuerdo a la estructura mostrada en la figura 9.1, un motor VWS se apoya en la existencia de tres componentes principales:

- *Service Listener*.
- *Request Handler*.
- *Generic Web-Service Client*.

El *Service Listener* es el componente responsable de recibir las peticiones de los clientes. Sus tareas principales serán:

- Recibir las peticiones de invocación de los clientes a través de un servidor web.
- Gestionar la cola de peticiones entrantes, asignando prioridades y sirviendo las peticiones según las políticas aplicadas a la definición del funcionamiento del motor. Aunque la creación y gestión de una cola

.

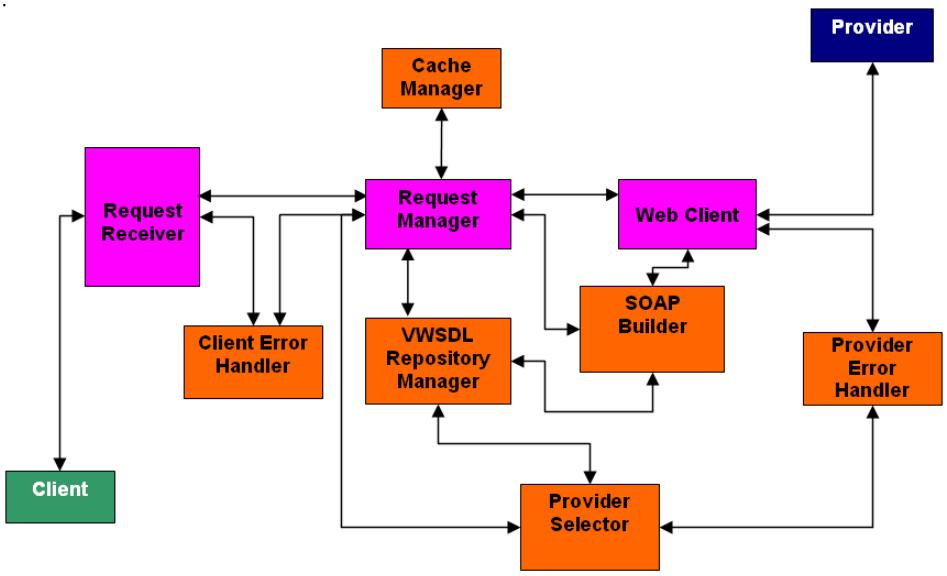

Figura 9.1. Arquitectura de un motor.

Componente Auxiliar	Función
Request Receiver	Es el componente encargado de dialogar con el servidor web, es decir, es el punto de entrada al motor VWS como aplicación que reside en un servidor de aplicaciones. Debe realizar las validaciones iniciales sobre las peticiones recibidas
Client Error Handler	Si una petición recibida es inválida, o como resultado de su ejecución se ha producido un error, este componente se encargará de comunicarlo a la entidad cliente de la forma más adecuada (error HTTP, *faultcode* de SOAP, etc.)

Cuadro 9.1. Componentes auxiliares del *Service Listener*.

de peticiones no es algo obligatorio, sí es algo conveniente, sobre todo en entornos de producción con un tráfico elevado de mensajería SOAP.

- Realizar una validación inicial de las peticiones recibidas, descartando aquellas que no correspondan con servicios web virtuales gestionados por el motor en cuestión o que sean, por algún motivo, erróneas.

Cuando una petición supera el *Service Listener*, pasa a las colas de entrada de mensajes del *Request Handler*. Este es el verdadero corazón del motor VWS, ya que será el que examine la petición e inicie la ejecución de acuerdo a las definiciones extraídas de los documentos VWSDL que definen el servicio que está siendo invocado. Las tareas que debe realizar el *Request Handler* son las siguientes:

- Es el responsable de procesar los mensajes SOAP recibidos de los clientes. Ello implica, por una parte, la extracción de la información VWSEL que pudiera contener la cabecera del mensaje SOAP recibido. Y por otra, la extracción del mensaje contenido en el cuerpo del mensaje SOAP, que será la información de entrada al servicio web virtual que ha invocado el cliente.

- La información VWSEL extraída del mensaje SOAP recibido del cliente podrá desencadenar acciones en el motor encaminadas a resolver peticiones de invocación mediante el uso de la *cache*, mediante el envío de mensajes almacenados correspondientes a invocaciones asíncronas anteriores, etc.; es decir, peticiones de invocación que pueden ser resueltas sin necesidad de realizar la invocación pertinente a un proveedor delegado.

- Con la información relativa a la petición de invocación recibida, este componente deberá buscar y cargar el documento VWSDL que define el servicio web virtual que está siendo invocado.

- El *Request Handler* debe iniciar un **proceso interno de ejecución** del motor que se encargue de realizar la ejecución según la definición establecida dentro del documento VWSDL del método invocado. Dentro del proceso interno de ejecución, el motor deberá implementar las funcionalidades adicionales de los servicios web virtuales, tales como gestión automática de errores, o mejoras de rendimiento mediante mecanismos de *caching* de la mensajería SOAP.

- Normalmente, una vez iniciado el proceso interno de ejecución, dicho proceso dará lugar a una invocación de un proveedor (aunque no siempre tiene porque ser así). En ese caso, y siguiendo las definiciones

contenidas en el documento VWSDL, se procederá a realizar la invocación del proveedor adecuado. Para ello, el *Request Handler* debe iniciar un proceso de evaluación que, a partir del conjunto de los posibles candidatos (contenidos en el documento de definición), permita obtener una lista ordenada de proveedores candidatos. El orden de dicha lista vendrá dado por las preferencias del diseñador del servicio web virtual y por las restricciones contenidas en la petición inicial de invocación.

Si la petición recibida en el motor da lugar a una invocación de un proveedor, entonces entrará en juego el tercer componente del motor, el *Generic Web-Service Client*. Este componente es el encargado de realizar las invocaciones hacia los proveedores seleccionados siguiendo los patrones de mensajería especificados en los documentos VWSDL. Este componente actúa como un cliente de servicios web desde el punto de vista de los proveedores invocados. Sus funciones son las siguientes:

- Transformar el mensaje SOAP recibido de la entidad cliente en un nuevo mensaje SOAP que siga el esquema que espera recibir el proveedor del servicio web real. Para ello, el motor ha de buscar el documento de transformación incluido (*embedded*) dentro del documento VWSDL. Dicho documento (una hoja XSL o una transformación simple) será aplicado al SOAP del cliente obteniendo un nuevo mensaje SOAP.

- El mensaje SOAP obtenido de la fase anterior es enviado al proveedor del servicio web, quien lo procesa y devuelve un nuevo mensaje SOAP. El *Generic Web-Service Client* debe gestionar la comunicación con el proveedor de la misma forma que lo haría cualquier aplicación cliente que realizase una invocación directa hacia dicho proveedor. Ello implica, entre otras cosas, la realización de una correcta gestión de errores, que permita al resto de componentes del motor reaccionar adecuadamente ante situaciones de error en las invocaciones al proveedor.

- El mensaje de respuesta obtenido del proveedor (si la invocación ha sido correcta) debe ser nuevamente transformado, creando un nuevo mensaje SOAP. Esta vez se trata del mensaje SOAP de respuesta a la invocación original de la entidad cliente. Al igual que en la primera fase, un documento de transformación debe indicarle al motor como ha de producirse el nuevo mensaje de respuesta.

Los tres componentes principales que acabamos de describir son los responsables de la ejecución de los procesos críticos del motor VWS. Pero dicha ejecución debe apoyarse en componentes auxiliares que permitan dividir las

Componente Auxiliar	Función
Request Manager	Se encarga de analizar las peticiones recibidas a través del *Service Listener* y encadenar la ejecución de los componentes auxiliares necesarios para llevar a cabo la ejecución de las peticiones entrantes, que son encoladas y atendidas mediante un sistema de prioridades que permite optimizar el rendimiento del motor.
VWSDL Repository Manager	La información contenida en una petición es insuficiente para llevar a cabo una ejecución. Información adicional debe ser extraída del documento VWSDL que describe el servicio. Este componente se encarga de gestionar el repositorio de documentos VWSDL gestionados por un motor VWS, facilitando así la obtención de información adicional a los componentes que lo soliciten.
Provider Selector	Se encarga de analizar la lista de proveedores candidatos y facilitar uno de ellos al componente que se lo solicite. Su funcionamiento está directamente ligado a las técnicas de *rating* y manejo de expresiones que hemos descrito en secciones anteriores.
Cache Manager	Es el responsable de realizar las labores de *caching* dentro del motor. Cada petición que se recibe en el motor, y cuando la definición del servicio virtual lo permita, deberá ser contrastada con los mensajes SOAP almacenados en la *cache*, con el fin de responder al cliente sin llegar a invocar al proveedor.

Cuadro 9.2. Componentes auxiliares del *Request Handler*.

Componente Auxiliar	Función
Web Client	Se encarga de coordinar las actividades de invocación hacia los proveedores de servicio web.
SOAP Builder	Es un componente auxiliar que permite construir y analizar mensajes SOAP. Los distintos componentes del motor se apoyan en él para crear y transformar mensaje que viajan hacia los clientes o los proveedores.
Provider Error Handler	El control de errores de proveedores ha de ser gestionado por separado, con el fin de facilitar al resto de componentes del motor dicha gestión.

Cuadro 9.3. Componentes auxiliares del *Generic Web-Service Client*.

tareas, facilitando la implementación del motor, a la vez que permiten dividir la ejecución de las invocaciones en tareas más pequeñas que pueden ser realizadas en paralelo. De esta forma, se mejora notablemente el rendimiento del motor VWS que, al fin y al cabo, es un elemento intermedio que, a parte de los beneficios que aporta, representa en realidad un nodo intermedio de proceso de mensajes que, como tal, introduce una sobrecarga en el proceso de las peticiones de invocación.

Las tablas 9.1, 9.2 y 9.3 muestran una posible composición de los componentes principales a partir de un conjunto de componentes auxiliares.

PARTE IV

Rating

CAPÍTULO 10

Rating

10.1. Rating

En la arquitectura propuesta es clave el elemento intermedio encargado de la invocación del proveedor más adecuado en cada ocasión. Dicha invocación debe ser posible gracias a la utilización de tres procesos bien diferenciados, cada uno de los cuales puede aplicarse a un elemento concreto de la arquitectura (cliente delegado, proveedor delegado, etc.):

- **Calificación**. Es el proceso por el cual un elemento de la arquitectura es analizado, obteniendo información estadística acerca de su funcionamiento, que será almacenada para un uso posterior.
- **Evaluación**. Es el proceso que permite obtener una valoración global para un elemento de la arquitectura concreto. Dicha valoración debe realizarse teniendo en cuenta toda la información relevante de que se dispone.
- **Selección**. Es el proceso por el cual, utilizando datos recogidos previamente, un elemento de la arquitectura es seleccionado entre un conjunto de ellos que ofrecen prestaciones similares.

Estos procesos (figura 10.1) son críticos para el correcto funcionamiento de la arquitectura, ya que son los que permiten que los intermediarios (motores VWS) elijan el proveedor de servicios web más adecuado a las necesidades del cliente que solicita una invocación.

Además, dichos procesos deben poder ser realizados de forma automática, sin intervención humana, ya que han de ser realizados continuamente, en cada invocación, dentro del motor VWS que la realiza.

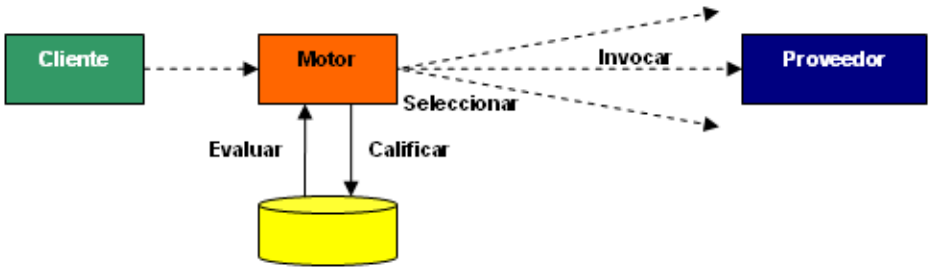

Figura 10.1. Esquema general de un sistema de rating.

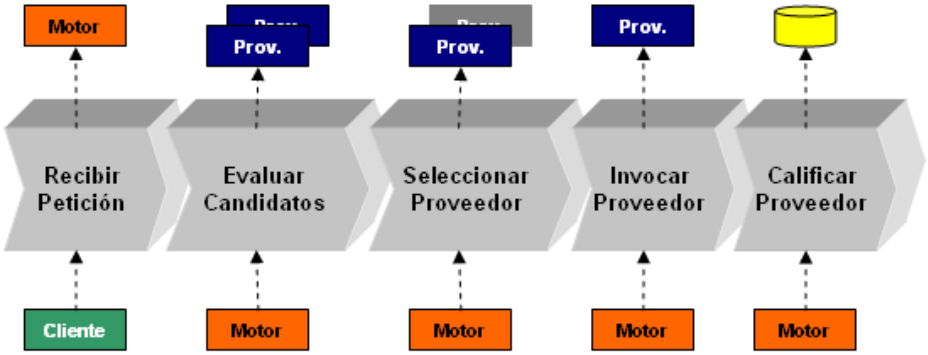

Figura 10.2. Fases del rating.

Cada vez que un cliente invoca un servicio web virtual, el motor VWS encargado de resolver dicha petición deberá acceder a una base de datos para obtener información estadística acerca de los posibles proveedores de servicios web que podrían resolver la petición en curso. Una vez obtenida la información, el motor debe **evaluar** a cada uno de los candidatos para, finalmente, **seleccionar** uno de ellos y realizar la invocación pertinente. Una vez finalizada la invocación hacia el proveedor de servicio web, el motor deberá **calificar** a dicho proveedor, obteniendo de esta forma nueva información que deberá ser reflejada en la base de datos de información histórica (fig 10.2).

Para que el sistema que acabamos de describir pueda funcionar de forma exitosa, deben cumplirse dos condiciones:

- Los procesos de calificación y evaluación deben ser realizados de forma automática, ya que se ejecutarán en cada invocación.
- La selección de un proveedor de servicio web debe realizarse mediante un sistema que, basándose en una valoración global de cada **proveedor de servicio** o **proveedor de servicio web**, permita construir

	r (ms.)	a (%)	t (invoc./seg.)
Servicio A	80	99.32	29
Servicio B	68	99.23	25
Servicio C	78	99.994	16

Cuadro 10.1. Ejemplo de variables de tres servicios.

una lista de candidatos que han de ser comparables en términos de magnitudes y unidades

A modo de ejemplo consideremos los tres servicios mostrados en el cuadro 10.1. Para cada uno de ellos consideraremos tres facetas que representan su comportamiento, entendiendo una faceta como un aspecto concreto del comportamiento global del elemento que se analiza. Dichas facetas son: tiempo de respuesta (r), disponibilidad (a) y rendimiento (t). La primera faceta o variable mide el tiempo que transcurre desde que se inicia una invocación hasta que termina de recibirse la respuesta, y se mide en milisegundos. La disponibilidad se mide en porcentaje, y representa la proporción de invocaciones exitosas respecto al número total de invocaciones. Por último, el rendimiento representa el número de invocaciones por unidad de tiempo que el proveedor de servicio web es capaz de atender. Esta última se mide en invocaciones por segundo.

El cuadro 10.1 nos muestra, a modo de ejemplo, el resultado del proceso que hemos definido previamente como calificación. Es decir, como resultado de un conjunto de invocaciones, el motor ha calificado distintas facetas de cada uno de los servicios A, B y C.

Si en un momento dado, el motor VWS recibe una invocación de un cliente, y debe elegir uno de los tres servicios para realizar la invocación hacia el proveedor de servicio web, deberá previamente seguir un proceso de evaluación de cada uno de los tres servicios, que conducirá a un posterior proceso de selección, que, como resultado, le indicará al motor VWS el proveedor de servicio web que debe ser invocado.

Para realizar la evaluación el motor deberá seleccionar una faceta y comparar su valor para cada uno de los servicios. Si la faceta elegida es r, el servicio seleccionado sería el B, puesto que tiene el menor tiempo de respuesta. Si la faceta es a, resultaría seleccionado el servicio C, el que tiene la disponibilidad más elevada. Y si resulta ser t, A sería seleccionado.

Para resolver situaciones como esta, y permitir que la evaluación se pueda realizar contemplando el mayor número de facetas posible, nosotros proponemos un sistema basado en el uso de **expresiones matemáticas**.

El número de facetas utilizadas en la composición de dichas expresiones es muy importante, ya que cuanto mayor es el número de facetas utilizadas, mejor es la descripción del comportamiento real del proveedor del servicio web.

10.2. Expresiones

Una expresión dentro de un documento VWSDL se construye a partir de un conjunto de variables y expresiones matemáticas que las enlazan. Las variables son utilizadas para representar facetas concretas del comportamiento de un elemento de la arquitectura, mientras las expresiones matemáticas (operaciones, funciones de ajuste,...) son utilizadas para aumentar o reducir (ponderar en general) la importancia que una faceta tiene dentro del comportamiento global de un elemento o un conjunto de elementos de la arquitectura.

A la hora de incorporar varias facetas a una misma expresión deben tenerse en cuenta varios factores que afectarán a dicha integración:

- Las unidades que se utilizan para medir cada faceta.
- Las magnitudes entre las cuales oscila cada faceta.
- La magnitud del resultado de la evaluación de la expresión.

Si queremos construir una expresión que combine las tres facetas mostradas en el cuadro 10.1, deberemos aplicarle previamente un algoritmo de adecuación a cada una de ellas. Dicho algoritmo debe dar como resultado, para cada faceta de forma independiente, un valor que oscile dentro de una escala concreta, predecible. En segundo lugar, el algoritmo debe unificar las unidades de las facetas, de tal forma que todas ellas se midan de acuerdo a las mismas unidades.

Si, además, adaptamos las facetas para que todas tomen valores dentro de la misma escala, podremos construir expresiones que combinen todas las facetas y arrojen como resultado un valor único cuya escala será predecible y unificada.

Tomando como ejemplo el cuadro 10.1, podemos convertir las facetas (f) en unas nuevas facetas (f') que tomen valores dentro de una escala predecible. Para ello, la ecuación (10.1) es una forma válida de hacerlo.

$$f' = \frac{ls + (hs * (f - min(F)))}{(max(F) - min(F))} \tag{10.1}$$

	r'	a'	t'
Servicio A	1	0.12	1
Servicio B	0	0	0.69
Servicio C	0.83	1	0

Cuadro 10.2. Valores de faceta adaptados.

Donde hs es el valor más alto de la escala resultante y ls el valor más bajo. Es importante tener en cuenta que F representa al conjunto de valores originales de la faceta, es decir, los valores de otros elementos de la arquitectura para la misma faceta, con lo que $max(F)$ y $min(F)$ dan como resultado los valores máximo y mínimo del conjunto de valores conocidos de la faceta respectivamente, no del conjunto de los valores posibles de f. Una expresión como la (10.1), aplicada al conjunto de valores de la faceta, nos dará como resultado un conjunto de valores nuevos que oscilaran entre ls y hs.

Por ejemplo, si fijamos ls con valor 0 y hs con valor 1, obtenemos la expresión (10.2) que nos permitirá obtener una nueva representación de cada uno de los valores de la faceta, donde todos los valores variarán entre 0 y 1.

$$f' = \frac{f - min(F)}{max(F) - min(F)} \tag{10.2}$$

De esta forma, aplicando la expresión (10.2) a los datos de el cuadro 10.1, obtendríamos los valores representados en el cuadro 10.2.

El uso de expresiones como (10.1) nos permite obtener finalmente un valor único, que nos permite manejar las facetas conjuntamente, ya que sus valores se mueven dentro de la misma escala y son representados utilizando las mismas unidades. Si la expresión para la selección de un elemento de la arquitectura es la de la fórmula (10.3), la puntuación global para cada elemento de la arquitectura puede ser obtenida fácilmente aplicando dicha expresión a los valores de la faceta de cada elemento.

$$S = \frac{r' + a' + t'}{3} \tag{10.3}$$

Donde S es la puntuación global de un elemento de la arquitectura. De acuerdo a los valores mostrados en el cuadro 10.2, las puntuaciones para cada uno de los servicios serían las mostradas en el cuadro 10.3. De acuerdo a este cuadro, un motor VWS que tuviese que seleccionar un servicio de entre los tres del ejemplo y, teniendo en cuenta que la expresión utilizada para la selección es la (10.3), optaría por el servicio A, que es el que ha

	S
Servicio A	0.71
Servicio B	0.23
Servicio C	0.61

Cuadro 10.3. Puntuación global con la expresión (10.3).

	S
Servicio A	0.37
Servicio B	0.56
Servicio C	0.39

Cuadro 10.4. Puntuación global corregida.

obtenido la puntuación más alta (0.71) dentro de la escala utilizada ($ls = 0$ y $hs = 1$).

Sin embargo, hay un pequeño detalle que no ha sido tenido en cuenta en la expresión (10.3), que ha inducido a un error en la selección. Dicho error está causado por un ajuste incorrecto de la faceta r'. Tal como se ha construido la expresión (10.3), los servicios que tengan un valor de r' elevado, tendrán un valor de S elevado, es decir, S es directamente proporcional a r'. Si tenemos en cuenta que r representa el tiempo de respuesta, nos damos cuenta de que estamos puntuando positivamente aquellos elementos de la arquitectura que tienen un mal tiempo de respuesta (r elevado).

La forma correcta de escribir S dependerá siempre del significado de las facetas que estemos considerando. Unas facetas han de ser integradas en la expresión de selección de tal forma que sean proporcionales, mientras otras han de ser inversamente proporcionales. En nuestro ejemplo, podemos redefinir S de la forma que se muestra en la expresión (10.4).

$$S = \frac{(1 - r') + a' + t'}{3} \tag{10.4}$$

Volviendo a calcular las puntuaciones globales para cada uno de los servicios obtendremos el cuadro 10.4. De acuerdo a este cuadro, el servicio que debería ser seleccionado sería el servicio B.

Lo que acabamos de describir aquí forma parte de dos procesos que hemos definido previamente como evaluación y selección. Tal como hemos comentado antes, existe un tercer proceso, la calificación, que consiste en analizar el comportamiento de un elemento de la arquitectura y obtener y almacenar nuevos valores para sus facetas.

	r (ms.)	a (%)	t (invoc./seg.)
Servicio A	80	99.92	14
Servicio B	69	99.12	24
Servicio C	78	99.994	16

Cuadro 10.5. Ejemplo de variables de tres servicios.

De acuerdo a nuestro ejemplo, cuando el motor VWS, después de haber seleccionado el servicio B, realice una invocación, deberá obtener unos nuevos valores para las facetas de dicho servicio. Supongamos que los valores obtenidos, para las mismas tres facetas, son los mostrados en el cuadro 10.5, junto con los valores, que no han sufrido modificaciones, de los otros dos servicios.

De acuerdo a los datos de nuestro supuesto, los valores de las facetas que representan al servicio B han sufrido leves modificaciones. Pero lo importante de esas modificaciones es que afectan al cálculo de las expresiones, ya que se han modificado los máximos (como en r), y los mínimos (como en a). Esto hace que cuando la expresión (10.3) vuelva a ser utilizada, sea necesario recalcular los resultados de las funciones $max(F)$ y $min(F)$. Situaciones como esta deben ser tenidas en cuenta a la hora de implementar un motor VWS, ya que no se podrán, generalmente, tener dichos valores precalculados.

Es muy importante el correcto desarrollo de la expresión utilizada para el ajuste de escala. Dicha expresión debería incorporar siempre funciones de máximo y mínimo, que permitan crear una nueva escala independientemente de los valores que se manejen, que deben ser calculados en el momento de evaluar la expresión.

En los ejemplos que aquí hemos mostrado se han incluido dos funciones simples (máximo y mínimo) que permiten crear un conjunto básico de expresiones lineales con ciertas limitaciones en su uso potencial. Es función del motor proveer funciones que permitan la utilización de escalas logarítmicas, la utilización de funciones de media ponderada, media deslizante (**moving average**), etc. Estos tipos de funciones permitirán el desarrollo de expresiones más complejas que se adapten mejor a las necesidades de los clientes, a las de los creadores de documentos VWSDL, y a la realidad del funcionamiento de los proveedores de servicio y de servicio web.

La ecuación 10.4 está basada en los valores adaptados de las facetas r, a y t. En realidad, una ecuación que nos dé una puntuación global a partir de los valores reales de las facetas debería haber sido escrita tal como se muestra en la ecuación 10.5.

$$S = \frac{(1 - \frac{r-min(R)}{max(R)-min(R)}) + \frac{a-min(A)}{max(A)-min(A)} + \frac{t-min(T)}{max(T)-min(T)}}{3} \tag{10.5}$$

Para simplificar la creación de expresiones y facilitar la homogenización de las variables (sus magnitudes y sus tipos) dentro de dichas expresiones, nuestra propuesta contempla la existencia de la función **adjust**. Dicha función se encarga de calcular y adaptar la escala para una variable determinada en función de los valores de que dispone, dando como resultado, para cada valor de faceta, un valor entre 0 y 1. La función **adjust** se define tal como aparece en la expresión 10.6, donde f es una faceta, y F es el conjunto de valores de dicha faceta.

$$adjust(f) = \frac{f - min(F)}{max(F) - min(F)} \tag{10.6}$$

De esta forma, la expresión 10.4 podría rescribirse tal como se muestra en la expresión 10.7.

$$S = \frac{(1 - adjust(r)) + adjust(a) + adjust(t)}{3} \tag{10.7}$$

Para simplificar el tratamiento de las variables inversamente proporcionales a las puntuaciones globales (como el tiempo de respuesta), se define la función **reverseAdjust** como se muestra en la expresión 10.8, con lo que la expresión 10.4 todavía podría simplificarse más, quedando tal como aparece en la expresión 10.9.

$$reverseAdjust(f) = 1 - adjust(f) \tag{10.8}$$

$$S = \frac{reverseAdjust(r) + adjust(a) + adjust(t)}{3} \tag{10.9}$$

10.3. Facetas

Las facetas pueden ser de distintos tipos atendiendo a una clasificación según el tipo de dato que se utilice para representarlas. Existen facetas **cuantitativas**, como puede ser el tiempo de respuesta o la disponibilidad, que pueden ser fácilmente medidas y utilizadas. Pero existe otro tipo de facetas, las facetas **cualitativas**, cuya medición inmediata no proporciona un resultado numérico.

Valor cualitativo	Valor cuantitativo
Muy Alto	4
Alto	3
Medio	2
Bajo	1

Cuadro 10.6. Faceta cualitativa, nivel de seguridad (sl).

Cuando queremos escribir expresiones que manejan facetas cuantitativas, lo único que tenemos que hacer es incorporarlas y garantizar que los datos están disponibles. Pero cuando queremos incorporar facetas cualitativas, debemos realizar un proceso de adaptación previo, que nos permita obtener un derivado de dicha faceta que pueda pasar a formar parte de cualquier proceso de evaluación y selección.

Existen técnicas, como las utilizadas en lógica difusa [143], que nos permiten asignar rangos de valores a una expresión cualitativa para convertirla en cuantitativa. Por ejemplo, en el cuadro 10.6 se muestran los posibles valores de una faceta denominada nivel de seguridad (sl).

Para convertir dicha faceta en una faceta cuantitativa, la primera opción sería atribuir un valor cuantitativo a cada uno de los valores cualitativos, como se muestra en el cuadro 10.6.

Supongamos ahora que tenemos una faceta multivaluada, como pueden ser los mecanismos de seguridad utilizados por un servicio (sl). Para este tipo de facetas, la conversión ha de realizarse mediante la asignación de un valor cuantitativo a cada uno de los posibles valores cualitativos. El resultado de convertir la faceta multivaluada se obtendrá mediante la aplicación de una expresión al conjunto de valores presentes.

Por ejemplo, el cuadro 10.7 muestra los mecanismos de seguridad utilizados por un servicio web. A cada uno de los posibles valores (v_i) se le asigna una valoración de 1, es decir, se considera que todos los mecanismos aportan la misma seguridad al sistema. Como esto no es lo habitual, se puede hacer una valoración no equitativa, proporcional a la seguridad que aporta el mecanismo (cuadro 10.7).

Cuando hemos asignado valores cuantitativos a cada uno de los posibles valores de la faceta cualitativa, debemos aplicar una expresión que nos dé como resultado un valor cuantitativo único. Para ello hemos de utilizar una expresión que combine todos los valores. La expresión (10.10) combina todos los valores de forma equitativa. También es posible ponderar cada uno de los valores según nuestro interés. Si hacemos esto último, podemos utilizar la valoración equitativa sin mayor problema, ya que la expresión, implícitamente, convierte los valores equitativos en proporcionales mediante

Mecanismo	Variable	Valoración Equitativa	Valoración Proporcional
HTTPS	v_1	1	2
Secure SOAP	v_2	1	2
Usuario/contraseña	v_3	1	1
XMLEncryption	v_4	1	3

Cuadro 10.7. Faceta multivaluada, mecanismos de seguridad.

el uso de factores. La expresión (10.11) es un ejemplo de ello.

$$f' = sum(v_i) \tag{10.10}$$

$$f' = 2 * v_1 + 2 * v_2 + v_3 + 3 * v_4 \tag{10.11}$$

Independientemente de cual sea la forma en la que asignamos valores (equitativa, proporcional, exponencial o logarítmicamente), lo que conseguimos es una forma de operar todos los valores obteniendo un valor único, cuantitativo, que será el que representa a la faceta multivaluada original. Y una vez que una faceta cualitativa ha sido convertida en cuantitativa, podrá ser incluida en cualquier expresión de selección al igual que cualquier otra faceta.

Además de ser cuantitativas o cualitativas, las facetas pueden estar relacionadas con distintos elementos de la arquitectura. Dicha relación tendrá un impacto considerable en la forma en que se escriben y evalúan las expresiones. Atendiendo a esta clasificación, las facetas podrán estar relacionadas con el proveedor del servicio web o con el proveedor del servicio.

Las facetas relacionadas con el proveedor del servicio web (en adelante facetas de servicio) definen su comportamiento. Algunos ejemplos ya los hemos comentado en secciones anteriores, como puede ser el tiempo de respuesta, la disponibilidad o el rendimiento.

Las facetas relacionadas con el proveedor del servicio (en adelante facetas de proveedor) definen su comportamiento independientemente de que dicho servicio sea accedido a través de un servicio web o no. Las facetas de proveedor tienen que ver con características de los productos o servicios que ofrece dicho proveedor. Por ejemplo, el tiempo que un proveedor tarda en enviar a un cliente una compra que dicho cliente ha realizado previamente (por Internet, por teléfono, o de la forma que sea), o la calidad del embalaje con que son enviados los paquetes a los clientes, son facetas que definen el comportamiento del proveedor en cuanto a su función como entidad que ofrece un servicio.

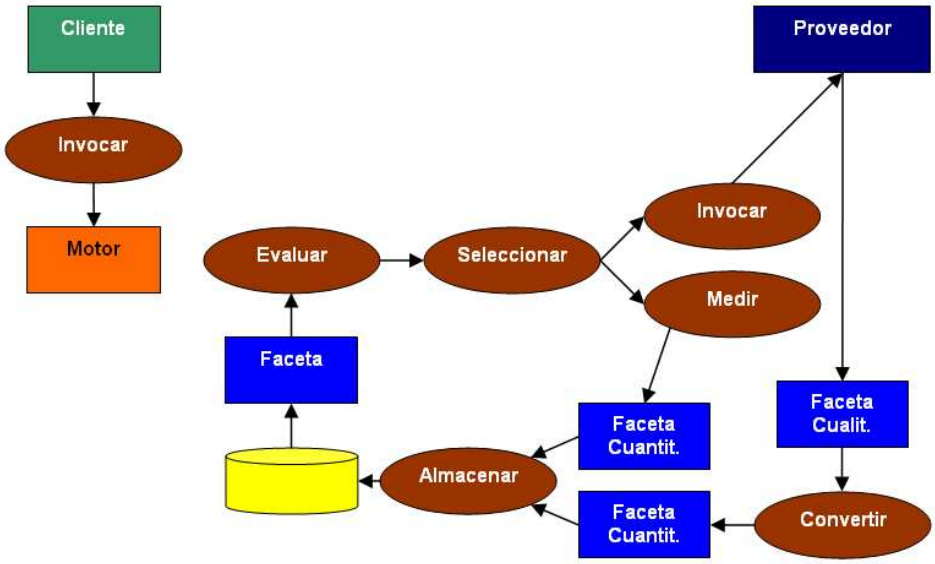

Figura 10.3. Proceso global de un sistema de rating.

Una representación esquemática de todo el proceso se muestra en la figura 10.3.

10.4. Extensiones a las expresiones

Las expresiones, tal como han sido presentadas aquí, ofrecen una buena forma de realizar una selección de proveedores candidatos. Sin embargo, el uso de expresiones, en base a factores de ponderación y variables, es bastante limitado a la hora de resolver situaciones más complejas. Por ejemplo, no es posible incorporar datos históricos a una expresión. Para solventar esta y otras carencias de las expresiones en su interpretación más simple surgen las extensiones, que son un conjunto de mecanismos que permiten ampliar las funcionalidades de las expresiones habilitando:

- Una mayor precisión en la forma en que se selecciona un proveedor.
- La posibilidad de que los clientes influyan en el proceso de selección desde el exterior del motor VWS.
- Formas para agregar a las expresiones facetas de comportamiento externas.

El primero de los mecanismos lo forman las variables externas, o facetas manejadas por el usuario, **facetas de usuario**. Este tipo de facetas no

son gestionadas por el motor VWS, ni representan el comportamiento del proveedor de servicio o del servicio web. Son variables externas al proceso de evaluación que permiten influir en él, dirigiendo, en cierto modo, la forma en que se efectúa la selección. Las facetas de este tipo deben existir en la base de datos en la que se almacena la información histórica, pero el encargado de garantizar su existencia ha de ser un agente externo, es decir, la arquitectura de servicios web virtuales que nosotros proponemos utilizará estas variables como un origen de datos (sólo las leerá), nunca serán actualizadas por el motor VWS como resultado de una invocación ni por otro motivo.

La utilidad de este tipo de facetas dependerá de las necesidades de los creadores de los documentos VWS. Por ejemplo, podemos crear una faceta "peso" que nos permita asignar un peso a un proveedor. De esta forma, podemos controlar desde el exterior del proceso de selección la forma en la que se evalúan las expresiones. Si aplicamos una faceta peso a toda una expresión, tendremos la oportunidad de decidir, en un momento dado:

- Que se fuerce el uso de un proveedor determinado.
- Que no se utilice un proveedor concreto bajo ningún concepto.
- Favorecer el uso de un proveedor o un conjunto de ellos en detrimento del resto de candidatos.

Si a la expresión (10.4) le aplicamos una faceta de este tipo, que denominaremos w, podemos tener control sobre la forma en que se evalúa, como se muestra en la expresión (10.12).

$$S = w * \frac{(1 - r') + a' + t'}{3} \tag{10.12}$$

Por ejemplo, supongamos que creamos la faceta externa w para cada uno de los servicios del cuadro 10.1. Si queremos que el proceso de selección dé como resultado el servicio A, sólo tendremos que asignarle un peso de 1 a dicho servicio, y 0 al resto (columna a en cuadro 10.8). Si lo que queremos es que el servicio C no resulte seleccionado independientemente de que los datos que definen su comportamiento sean buenos, bastará con asignarle un peso 0 al servicio C y un peso 1 al resto de servicios (columna b en cuadro 10.8). Y si lo que queremos es darle una mayor relevancia a un servicio respecto de los otros, permitiendo de esta forma que sea seleccionado en mayor número de ocasiones, deberemos asignar pesos según convenga. Un ejemplo de ello es la columna c de el cuadro 10.8.

Otro mecanismo eficaz, sobre todo en el momento de crear expresiones que alcancen cierto nivel de complejidad en cuanto a los elementos que incorporan, es la definición de **facetas complejas**. Una faceta compleja se

	a	b	c
Servicio A	1	1	0.60
Servicio B	0	1	0.30
Servicio C	0	0	0.30

Cuadro 10.8. Métodos de control sobre el proceso de selección.

crea asignando una expresión a una nueva faceta, que quedará definida a partir de ese momento, y podrá ser incluida, como una faceta estándar, dentro de expresiones que podrán ser utilizadas con posterioridad. Por ejemplo, la expresión (10.13) muestra la creación de la faceta gs, que consiste en la obtención de una puntuación global para un elemento de la arquitectura, y que incorpora varias facetas de dicho elemento.

$$gs = 0{,}5 * (1 - r') + 0{,}3 * a' + 0{,}2 * t' \tag{10.13}$$

A partir de la definición de gs podemos construir nuevas expresiones que incorporen gs como una faceta más, como en la expresión (10.14), donde la calificación final de un servicio vendrá dada por un valor que pondera por igual el coste del servicio ($cost$) y una puntuación global (gs) calculada previamente, que incorpora información acerca del tiempo de respuesta, la disponibilidad y el rendimiento. Al igual que ocurre con las expresiones de ajuste de escala, los valores de las facetas complejas no pueden estar precalculados, ya que eventuales modificaciones en los datos deben ser reflejadas cada vez que se utilice el valor de dicha faceta.

$$score = gs * 0{,}5 + cost * 0{,}5 \tag{10.14}$$

Otro mecanismo útil a la hora de escribir expresiones es el uso de valores históricos dentro de las expresiones, las llamadas **facetas temporales**. Utilizando este tipo de facetas podemos escribir expresiones que involucren valores presentes y valores pasados de las mismas facetas. Un ejemplo de ello lo podemos ver en la expresión (10.15), donde se utiliza el valor último de gs, conjuntamente con el valor anterior.

$$ts = \frac{gs_{i-1} * (st - 1) + gs_i}{st} \tag{10.15}$$

La expresión (10.15) nos da una puntuación total (ts) para un elemento dado partiendo de una faceta compleja (gs), que es utilizada a la vez como una faceta temporal, inluyendo en la misma expresión los dos últimos valores de gs, es decir, gs_{i-1} y gs_i. La faceta st, donde $st \in [0, 1]$, actúa como un

mecanismo estabilizador, que permite dar mayor relevancia a los valores más recientes respecto de los valores más antiguos de la misma faceta.

10.5. Extensiones a la arquitectura

La arquitectura basada en expresiones, tal como ha sido expuesta aquí, restringe la posibilidad de control sobre las invocaciones única y exclusivamente al motor VWS. Será este quien decida, evaluando expresiones y seleccionando, cual será el proveedor elegido en cada momento. No existe ninguna posibilidad de que el cliente exprese sus necesidades a la hora de realizar el proceso de selección.

Para dotar de flexibilidad a la arquitectura, nuestro modelo ofrece tres mecanismos adicionales, cada uno de los cuales dota a los clientes de un nivel de flexibilidad creciente respecto a las anteriores:

1. Clases de servicio.
2. Restricciones.
3. Expresiones de usuario.

Estos tres mecanismos extienden las funcionalidades del sistema basado en expresiones hacia el cliente, y es éste quien ha de decidir como quiere que se evalúen las expresiones, influyendo de esta forma en el proceso de selección. Para que estos mecanismos sean útiles, los clientes han de poder comunicarse con los motores VWS con el fin de poder indicar la forma en la que quieren controlar la evaluación. El procedimiento para realizar esta comunicación se basa en el uso de las ya mencionadas cabeceras SOAP.

El primero de los mecanismos, las **clases de servicio**, permite a los clientes seleccionar una expresión entre un conjunto de ellas disponibles. Para ello, el servicio web virtual ha de incorporar en su documento de definición un conjunto de expresiones, cada una de ellas identificada de forma única con un nombre. De esta manera, cuando una aplicación cliente desee utilizar una expresión determinada, deberá enviar en la cabecera de la petición SOAP, un documento VWSEL, tal como se muestra en el ejemplo 10.1.

Cuando el motor VWS recibe una petición SOAP conteniendo un documento VWSEL, deberá extraerla, procesarla convenientemente, y eliminarla del mensaje SOAP, ya que dicha cabecera sólo es útil para el motor VWS.

Si la cabecera VWSEL contiene un mecanismo de clase de servicio, el motor deberá comprobar la existencia de dicha clase, y utilizar la expresión asociada al identificador recibido en la cabecera cuando se realice el proceso

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:vwsel="http://schema.det.uvigo.es/vws10/vwsel">

  <soap:Header>
    <vwsel:extension>
      <vwsel:select type="engine">cos#1</vwsel:select>
    </vwsel:extension>
  </soap:Header>

  <soap:Body>
    <pedido xmlns="http://tempuri.org/Pedido/">
      <cantidad>98</cantidad>
    </pedido>
  </soap:Body>

</soap:Envelope>
```

Ejemplo 10.1: Ejemplo de mensaje SOAP.

de evaluación. De esta forma, los clientes disponen de una libertad limitada a la hora de seleccionar el proveedor que desean utilizar. En realidad, lo único que hacen es seleccionar la expresión que ha de ser utilizada en la evaluación.

El segundo mecanismo de extensión, el que hemos denominado **restricciones**, consiste en permitir a las aplicaciones cliente podar el árbol de proveedores disponibles utilizando condiciones *booleanas*. Así, dentro de la cabecera SOAP, los clientes pueden enviar una expresión *booleana* que amplíe las condiciones que ha de cumplir un proveedor para ser elegible, reduciendo de esta forma la lista de candidatos. Así, el cliente puede tener un mayor control sobre el proceso de evaluación, dejando fuera del mismo a aquellos proveedores que no cumplan las condiciones especificadas por el cliente. En el ejemplo 10.2 se muestra un ejemplo de ello.

El último mecanismo disponible, denominado **expresiones de usuario**, es el que permite que sean las aplicaciones cliente las que decidan como se ha de realizar la evaluación. Para ello, dentro de las cabeceras SOAP, los clientes han de especificar la expresión exacta que el motor ha de utilizar en el proceso de evaluación (ejemplo 10.3)

Los mecanismos de extensión basados en restricciones y en expresiones de usuario pueden plantear ciertos problemas, ya que permiten que sea el usuario quien controle los datos de entrada al proceso de evaluación (la

```
...
<SOAP:Header>
  <vwsel:extension>
    <vwsel:select type="restrict">
       (availability>98) & (responseTime<20)
    </vwsel:select>
  </vwsel:extension>
</SOAP:Header>
...
```

Ejemplo 10.2: Ejemplo de utilización de cabeceras SOAP.

```
...
<soap:Header>
  <vwsel:extension>
    <vwsel:select type="client">
      0.5*adjust(availability) + 0.5*reverseAdjust(responseTime)
    </vwsel:select>
  </vwsel:extension>
</soap:Header>
...
```

Ejemplo 10.3: Expresiones de usuario.

lista de proveedores elegibles) en el caso de las restricciones, y el algoritmo de evaluación (expresiones matemáticas en realidad) en el caso de las expresiones de usuario.

Los errores que se pueden producir, si el proceso de evaluación no se realiza correctamente, podrían conducir a que no se produzca la selección de ningún proveedor, en cuyo caso el motor VWS deberá decidir si selecciona un proveedor de forma aleatoria (o siguiendo algún mecanismo predeterminado), o devuelve un error (un mensaje *fault*) a la aplicación cliente que originó la invocación.

Los mecanismos, tal como han sido descritos aquí, no son excluyentes, es decir, una aplicación cliente podría decidir en un momento dado enviar una cabecera SOAP especificando una clase de servicio y una restricción, o una expresión de usuario y una restricción. Si lo que especifica es una clase de servicio y una expresión de usuario, el motor debería utilizar la expresión, ya que es el mecanismo que más libertad otorga al cliente en el proceso de evaluación. Esta combinación última es en realidad incongruente, y debería generar un error.

Queda fuera del alcance de este trabajo decidir qué situaciones deberían producir errores y cuales no. La razón para ello es cierta carencia de capacidad semántica del lenguaje XSD, lo que se traduce, en nuestro caso, en la imposibilidad de expresar la forma de resolver determinadas situaciones

a la hora de describir el lenguaje VWSDL. Debido a todo esto, la responsabilidad de decidir cómo resolver estos tipos de situaciones se transfiere a los implementadores de motores independientes, ya que serán estos últimos los que tengan que interpretar el contenido de los documentos VWSEL.

10.6. Selección

El proceso de selección es el objetivo final con el que se crean todos los mecanismos que acabamos de describir aquí. La selección se basa en la obtención de una lista de candidatos que han sido previamente calificados utilizando algún mecanismo o algoritmo que permite que dicha selección pueda ser realizada de forma neutral. Para ello, procesos de unificación de escalas y de unidades han tenido que ser aplicados previamente.

Una vez que se obtiene una lista de candidatos válidos comienza el proceso de selección. Dicho proceso puede ser controlado, como acabamos de ver, por el motor VWS, y codirigido por el cliente que solicita una invocación (mediante alguno de los mecanismos de extensión de la arquitectura).

El objetivo de la selección no es obtener el mejor proveedor, sino una lista ordenada de proveedores susceptibles de ser invocados. Es importante tener en cuenta esta diferencia, ya que los mecanismos de control y recuperación de errores del motor VWS pueden actuar a partir de esta lista. Es decir, no sólo se obtiene un candidato para realizar una invocación, sino que se obtiene un conjunto de candidatos equivalentes en funcionalidad (candidatos alternativos), aunque no en prestaciones.

A la hora de realizar una invocación hacia un proveedor y, dependiendo de cómo se haya redactado el documento VWSDL, el motor podría decidir invocar varios servicios en paralelo, y devolver al cliente la respuesta más rápida. O podría invocar al primero de la lista, y, ante un eventual fallo del proveedor, invocar al segundo. Y así sucesivamente hasta conseguir obtener una respuesta del proveedor o hasta agotar la lista de candidatos, en cuyo caso un *fault* debe ser devuelto al cliente.

10.7. Calificación

Si la evaluación y la selección son los procesos críticos en la toma de decisiones a la hora de elegir un proveedor, la calificación es todavía más crítica, ya que una buena evaluación depende directamente de disponer de información precisa acerca de cada proveedor.

La calificación es el proceso por el cual, una entidad alimenta una base de datos con información relativa al funcionamiento de un elemento de la

arquitectura. La definición puede parecer un poco vaga en los términos utilizados. Ello se debe a que la "entidad" que alimenta la base de datos no tiene que ser necesariamente un elemento de la arquitectura; no tiene que ser el motor VWS necesariamente. Cualquier agente, interno o externo, puede realizar dichas actualizaciones. Por otra parte, el elemento calificado no tiene por que ser un servicio web. Puede ser cualquier elemento de la arquitectura que nos interese calificar, como el proveedor de un servicio web, o el mismo proveedor del servicio.

En realidad, las expresiones tal como han sido definidas aquí, pueden contemplar el comportamiento de cualquier elemento que intervenga en la arquitectura. Lo único que se necesita para ello es disponer de variables y sus valores.

El proceso de calificación puede ser dividido en dos procesos fundamentales:

1. **Toma de medidas**. Es el mecanismo por el cual un proceso que tenga lugar dentro de la arquitectura planteada es analizado desde distintas ópticas con el fin de obtener datos numéricos que representen el comportamiento de las entidades que intervienen en dicho proceso.

2. **Almacenamiento**. La información obtenida como resultado del procedimiento anterior, y una vez tratada convenientemente, debe ser almacenada en algún sitio con el fin de poder ser utilizada posteriormente por un eventual proceso de evaluación.

10.7.1. Toma de Medidas

La toma de medidas dentro de una arquitectura de servicios web es un proceso que ha de ser estudiado detenidamente, y que ya ha sido tratado en otros trabajos [119, 100]. Es muy importante tener en cuenta que nuestra propuesta incluye un elemento intermedio en la arquitectura que será el que permita llevar a cabo, de forma completamente **automática**, una gran parte de las mediciones necesarias para describir con exactitud el comportamiento de cada elemento de la arquitectura.

Tal como hemos comentado anteriormente, los procesos de calificación no quedan restringidos únicamente a los servicios web o proveedores de servicios web. Otros elementos de la arquitectura pueden ser calificados con el fin de poder ser utilizados en los procesos de evaluación de expresiones. En este caso, la calificación no podrá ser realizada por el motor VWS. Aún en el caso de que pudiese ser llevada a cabo automáticamente, la calificación tendría que ser ejecutada por un agente externo. En realidad, los motores VWS pueden calificar automáticamente todos aquellos elementos que estén directamente relacionados con las invocaciones de servicios web, pero

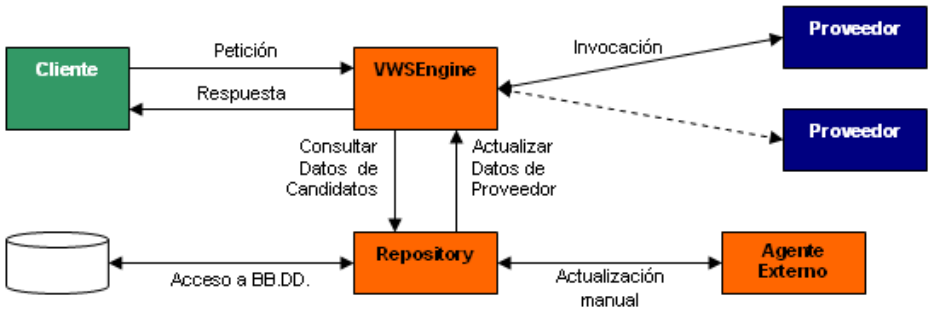

Figura 10.4. Procesos involucrados en el rating.

	Cuantitativa	Cualitativa
Proveedor de Servicio Web	Automática	Automática diferida
Proveedor de Servicio	Automática diferida	Manual

Cuadro 10.9. Tipos de calificación.

no podrán calificar elementos de la arquitectura cuya intervención en una invocación dure más allá del propio ciclo de vida de la invocación (figura 10.4).

Existe una relación clara entre los tipos de facetas y los tipos de calificaciones que se pueden aplicar. En el cuadro 10.9 se muestra la relación existente entre ambos dependiendo del tipo de elemento de la arquitectura que se está realizando.

Las calificaciones del tipo "Automática diferida" pueden ser realizadas automáticamente pero no durante el ciclo de vida de una invocación, ya que requieren la intervención de un agente externo a dicha invocación. La calificación "Manual" debe ser llevada acabo por un agente externo de forma manual.

10.7.2. Almacenamiento

Una vez obtenidas las medidas necesarias, tanto si provienen del motor como si lo hacen de un agente externo, dichas medidas deben ser almacenadas para su posterior utilización en la evaluación de expresiones. Es importante destacar que nuestra arquitectura no impone una forma estándar de medir o de almacenar mediciones. Es responsabilidad de los implementadores de motores VWS decidir qué medidas estarán disponibles y como se almacenarán dichas medidas.

Independientemente de cuales sean las medidas, la actualización de la

base de datos que contenga la información sobre las facetas utilizadas podrá realizarse de alguna de las siguientes formas:

1. *Online*.
2. Por ráfagas (*bursting*).
3. A intervalos de tiempo predefinidos.
4. Por muestreo (*sampling*).

En la actualización **online**, cada vez que el motor VWS finaliza una invocación, recoge los datos relativos a dicha invocación y se los comunica al encargado de almacenarlos.

Cuando la actualización se realiza por **ráfagas**, el motor VWS espera a tener un bloque de datos suficientemente grande antes de comunicárselo al encargado de almacenarlo. El mayor interés de la actualización por ráfagas reside en la minimización del impacto de dicha actualización en el motor VWS y, en particular, en el proceso de invocación.

En aras de un buen rendimiento de la base de datos y del propio motor VWS, la actualización puede realizarse, independientemente de los datos recogidos, a **intervalos de tiempo** definidos previamente, es decir, en diferido.

Por último, la forma de actualización que menos impacto causa en el funcionamiento de la arquitectura consiste en no utilizar la totalidad de los datos de que se dispone, sino solamente una **muestra** de ellos. Esta técnica de muestreo plantea a su vez un inconveniente importante que ha de ser tenido en cuenta a la hora de utilizarla, que es la calidad de la muestra. Al igual que otras áreas en las que se aplican técnicas de muestreo (publicidad, estudios sociales,...), el tamaño de la muestra ha de ser suficientemente grande para que ésta sea representativa de la totalidad del universo observado.

Las técnicas de muestreo son perfectamente válidas dentro de la arquitectura que proponemos. Pero ha de ser el diseñador de los servicios web virtuales, basándose en las facilidades que le ofrezca el motor VWS que esté utilizando, quien decida que técnica de actualización utilizar en cada momento.

10.8. Repositorios

Aunque nuestra arquitectura no obliga a una implementación concreta del componente encargado del almacenamiento de la información estadística

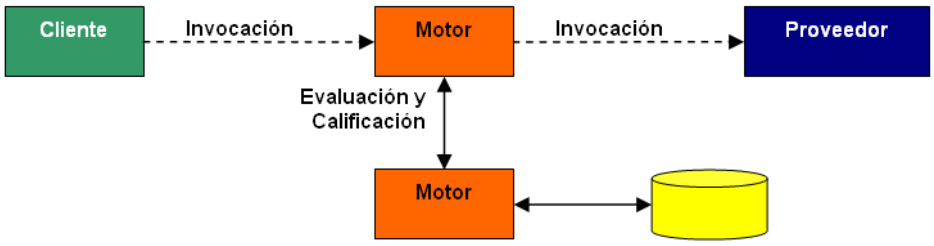

Figura 10.5. Arquitectura con 4 entidades.

relativa a los elementos de la arquitectura, a continuación haremos una breve reflexión acerca de las implicaciones que conlleva dicha implementación.

El primer aspecto a tener en cuenta es la localización de la base de datos. Básicamente se puede reducir a dos opciones: Internet e Intranet.

Si la base de datos está alojada en una Intranet, en el entorno corporativo y típicamente actualizada desde los motores VWS corporativos, los datos que se almacenen serán datos relativos a invocaciones efectuadas desde los nodos de la Intranet o hacia los mismos, dependiendo de si actúan como clientes o como proveedores dentro de nuestra arquitectura.

Los datos almacenados en una base de datos, utilizada de esta manera, son datos fiables, correspondientes a transacciones realizadas con proveedores o clientes de la entidad que gestiona la base de datos. En este caso, el componente encargado del almacenamiento está actuando como un repositorio de *rating* dentro del entorno corporativo.

Si el repositorio está localizado en una zona pública, accesible a través de Internet, la forma en que se utiliza es completamente distinta. En primer lugar, nos encontramos ante una nueva forma de funcionamiento de la arquitectura, donde una tercera entidad es la responsable de aportar información a los motores VWS. Debemos tener en cuenta que el motor VWS y el encargado del almacenamiento no tienen porque ser necesariamente la misma entidad, con lo que podríamos llegar a arquitecturas de producción donde la ejecución de una invocación involucre a cuatro entidades: cliente de servicio web, proveedor de servicio web, motor VWS y repositorio (figura 10.5).

Cuando el repositorio está gestionado por una entidad independiente, tres aspectos deben ser considerados:

1. Los motores que se basen en repositorios públicos están depositando su confianza en una tercera entidad, de la que desconocen el origen de la información que les facilitan.

2. Las entidades que gestionan repositorios públicos son las responsables

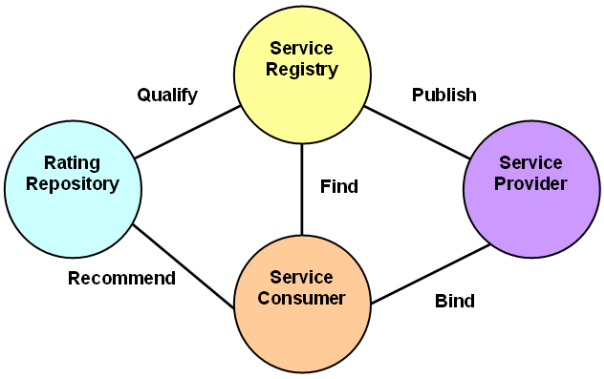

Figura 10.6. Roles en una arquitectura de recomendación.

de verificar la validez de los datos que son añadidos a sus bases de datos.

3. Las entidades cliente, como usuarias de servicios web virtuales, y las entidades proveedoras, como elementos utilizados por dichos servicios web, dependen completamente de la calidad ofrecida por las otras dos entidades (motor y repositorio) en cuanto a su funcionamiento y la veracidad de la información que proporcionan.

En lo que a la entidad gestora del repositorio se refiere, es importante tener en cuenta que la información contenida en sus bases de datos será más fiable cuanto más dispar sean sus fuentes. Es decir, cuantos más motores VWS alimenten la base de datos, más precisos serán los datos que los repositorios ofrezcan a los motores.

El hecho de tratar el repositorio como un elemento independiente de la arquitectura (tanto si es manejado por una entidad distinta al motor como si no), abre las puertas a una nueva forma de entender la calidad de servicio. De acuerdo a esta afirmación, es posible implementar arquitecturas de recomendación [141].

Tal como se muestra en la figura 10.6, en una arquitectura de este tipo existe un componente adicional, el repositorio de *rating*, responsable de recomendar, a los clientes que lo soliciten, el proveedor más adecuado a sus necesidades. El repositorio de *rating* se alimenta de datos proporcionados por un directorio (registro UDDI o equivalente).

Tal como se explica en [141], una forma más precisa de construir ese repositorio de servicio se basa en alimentar dicho repositorio con datos proporcionados por los propios clientes, ya que son éstos quienes saben realmente como funciona un determinado proveedor de servicio web. Según este esquema, los clientes de servicios web califican a los proveedores de dichos

servicios enviando datos al repositorio de *rating*. El mismo u otro cliente solicitará al repositorio, en un momento posterior, ”su opinión” acerca de un determinado proveedor de servicio con el fin de seleccionarlo para una eventual invocación (figura 2.21).

PARTE V

Aplicaciones de la arquitectura propuesta

CAPÍTULO 11

Alta Disponibilidad

11.1. Introducción

El *clustering* [81, 124] y los sistemas tolerantes a fallos [91, 130] son los mecanismos más utilizados para tratar problemas de disponibilidad, así como los de escalabilidad o continuidad.

En las soluciones de alta disponibilidad para aplicaciones web, basadas en el uso de *clusters*, se utilizan dos o más servidores de aplicaciones que se encargan de recibir las peticiones de invocación de los servicios web y ejecutar dichos servicios. Si alguno de los servidores (nodos dentro de un *cluster*) falla, el resto de los nodos procesará su carga de trabajo y el *cluster*, en su conjunto, atenderá todas las peticiones. Pero toda la carga de trabajo que estaba pendiente en el nodo que falló no podrá ser atendida, es decir, las entidades cliente recibirán un error y deberán reintentar la invocación.

Adicionalmente, la implementación de sistemas *cluster* se hace siempre utilizando software propietario, debido, en gran parte, a la elevada necesidad de integración que existe entre el software de gestión del *cluster* y los sistemas operativos que se ejecutan dentro de cada nodo del *cluster*. Así, no podremos formar un *cluster* con un nodo basado en Windows y un nodo basado en Linux, o en zSeries. Más aún, el software de gestión de *clusters* no está estandarizado, con lo que no podemos formar un *cluster* con nodos que, aún disponiendo de la misma versión de sistema operativo, no utilicen el mismo software de *cluster*. Por ejemplo, en entornos Windows, se puede utilizar software de *cluster* basado en Microsoft (MSCS), o en Veritas, entre otros. Ambos sistemas son incompatibles entre sí.

En los entornos altamente tolerantes a fallos, la disponibilidad de un sistema sin redundancia (no hay maquinas duplicadas) dependerá al 100 % de los componentes que forman parte de dicho sistema, ya que la disponi-

bilidad que ofrece el sistema globalmente se limita a la que pueda ofrecer una única máquina.

Desde otro punto de vista, menos ligado a las implementaciones de bajo nivel, la mejora de disponibilidad de un servicio web, basada en la implementación de dicho servicio sobre un sistema *cluster* (a través de servidores de aplicaciones redundantes), se centra en la protección de las implementaciones de los servicios web, pero no su interfaz. Aunque el interfaz es a menudo considerado un mero contrato (*contract*) entre partes o una especificación en la que ambos se ponen de acuerdo, nosotros consideramos que dicho interfaz también debe ser protegido en tiempo de ejecución.

Esta consideración se basa en el hecho de que el contrato entre cliente delegado y proveedor delegado (el interfaz al fin y al cabo), no es un simple contrato de servicio (como el que puedan firmar el proveedor de servicio y el cliente). La diferencia entre ambos tipos de contratos radica en que los contratos entre entidades delegadas tienen implicaciones más allá de la "firma" de dicho contrato, llegando a extenderse hasta las invocaciones en tiempo de ejecución.

Como consecuencia de la existencia de los contratos entre delegados, si el proveedor modifica su interfaz, la disponibilidad será, con toda probabilidad, de cero, ya que la modificación unilateral del interfaz hará que la forma en la que se envían y reciben los mensajes entre delegados se vea modificada, impidiendo, por completo, la comunicación entre ambas partes.

¿Cómo podemos construir servicios web con características de alta disponibilidad? La mejor infraestructura hardware sería, sin duda alguna, aquella formada por un *cluster* cuyos nodos presenten características de tolerancia a fallos en su construcción. ¿Cuál sería la solución software más adecuada bajo esta óptica?

La infraestructura software que utilicemos para resolver problemas de disponibilidad deberá ofrecer soluciones a tres problemas principales:

1. Identificación de nodos pertenecientes al *cluster* que estén presentando algún problema en su funcionamiento.

2. Provisión de los mecanismos adecuados para poder realizar mantenimiento del *cluster*.

3. Modificación de la estructura del *cluster* permitiendo añadir y eliminar nodos según sea necesario.

11.2. Solución propuesta

Desde el punto de vista de la disponibilidad, nuestra solución puede ser aplicada como un mecanismo que da soporte a la construcción de **clusters de servicios web**. Es muy importante este concepto, y es igual de importante no confundirlo con un *cluster* de servidores de aplicaciones que ofrece servicios web.

Un *cluster* de servicios web se construye creando un servicio web virtual por cada servicio al que queramos dotar de un sistema de alta disponibilidad. Necesitaremos además un nodo concentrador, que será el gestor del *cluster*. Este nodo es en realidad un motor VWS.

Proponemos el uso de servicios web virtuales que encapsulen los servicios reales para, de esta forma, dar solución a los siguientes problemas:

- Proteger el interfaz del servicio web real.
- Permitir la construcción de *clusters* de servicios web, y no *clusters* de servidores de servicios web.

La arquitectura de la solución propuesta se basa en la construcción de *clusters* que estarán formados, típicamente, por dos tipos básicos de nodos: un nodo controlador del *cluster*, que denominaremos **nodo principal**, y un conjunto de nodos secundarios, denominados **nodos de servicio**. El nodo controlador será el encargado de recibir las peticiones de ejecución al *cluster*, y reencaminarlas convenientemente a algún nodo de servicio, encargado de ofrecer la ejecución de servicios web. El diagrama que corresponde a un *cluster* de servicios web se muestra en la figura 11.1.

Utilizando el esquema de virtualización que hemos planteado, la construcción de un *cluster* en los términos que aquí se han definido se basa en la utilización de servicios web virtuales, que deben ser ofrecidos desde el nodo controlador del *cluster*. Dicho nodo debe ser implementado como un motor VWS y, por tanto, los servicios web virtuales que se ofrezcan desde dicho motor han de tener su correspondiente documento VWSDL. Utilizando los elementos `method` e `invoke` del lenguaje VWSDL, podremos establecer la relación existente entre servicios web virtuales y servicios web reales. Esto nos conduce de una manera sencilla a la creación de una arquitectura en *cluster*, en la que el nodo controlador o principal (un motor VWS) será el responsable de recibir peticiones de invocación de los clientes delegados y distribuir la carga de trabajo entre los nodos secundarios del *cluster*, los nodos de servicio (implementados como proveedores delegados).

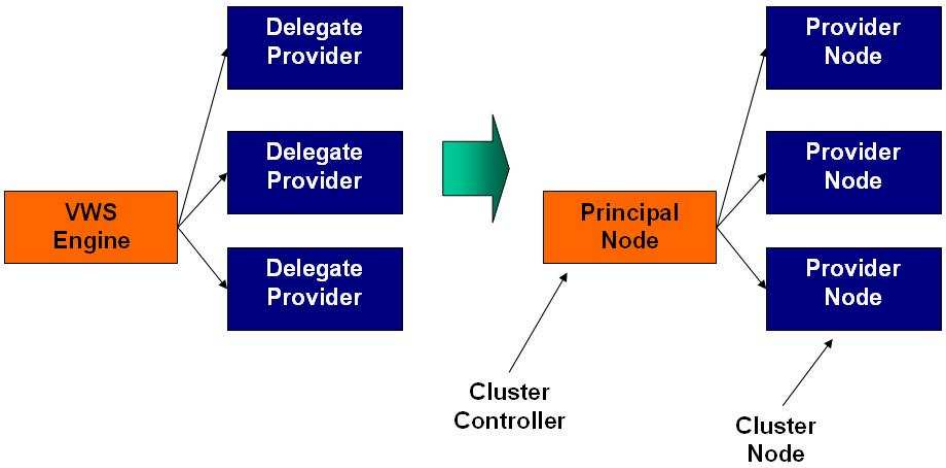

Figura 11.1. Construcción de un *cluster* a partir de la arquitectura virtual.

11.3. Diseño e Implementación

De acuerdo a la estructura planteada, si un proveedor delegado (un nodo de servicio) falla, el motor VWS (nodo controlador) redistribuirá las invocaciones pendientes, de tal forma que los nodos todavía operativos del *cluster* se responsabilizarán de la carga de trabajo que ha quedado sin asignar debido a la caída (fallo) de uno de los nodos de servicio.

Pero todavía quedan problemas por resolver: ¿Cómo podemos implementar el sistema de reparto de carga? ¿Cómo podemos construir un *cluster* que nos permita realizar paradas planificadas (o no planificadas) de un nodo de servicio? ¿Cómo puede el nodo controlador seleccionar el nodo de servicio más adecuado en cada momento? Las siguientes secciones contienen las respuestas a estas cuestiones.

11.3.1. Construcción de un *cluster* de servicios web

Supongamos la existencia de un servicio web (WS1) que contiene un método (M1). Si queremos convertir WS1 en un servicio web altamente disponible, debemos hacer su despliegue en un *cluster*. Dicho proceso de despliegue requiere que el servicio web sea instalado en cada uno de los nodos secundarios del *cluster*. De esta forma, más de una instancia del servicio web puede ser utilizada, y cada una de ellas podrá ejecutarse en un nodo diferente. Cada una de estas instancias (instalada en un nodo diferente) es equivalente al resto; todas ellas son instancias equivalentes.

Para que nuestro *cluster* de servicios web esté plenamente operativo, debemos crear un documento VWSDL que defina un servicio web virtual

(llamémosle VWS1), que debería contener, al menos, un método virtual (VM1). Recordemos aquí que los clientes de un motor VWS (los clientes de un *cluster* de servicios web en nuestro caso) no utilizan los documentos VWSDL para sus procesos de *binding*, sino que utilizan un documento WSDL estándar derivado a partir de un documento VWSDL.

Como hemos comentado en capítulos anteriores, dentro de cada documento VWSDL debe especificarse cómo ha de llevarse a cabo la ejecución de cada método virtual. Por ello, necesitamos incluir en cada documento VWSDL al menos tres elementos del lenguaje: un elemento `service` que describe el servicio web virtual; un elemento `method` que describe el método virtual que se publica (incluyendo sus elementos `input` y `output`); y uno o más elementos `invoke`. Éstos últimos son los responsables de especificar cómo y dónde ha de realizarse la implementación del método. Veamos el funcionamiento con un ejemplo.

Cuando llega al motor VWS (controlador de un nodo) una petición de ejecución para el método VM1, el motor debe seleccionar el proveedor delegado (nodo secundario o de servicio) más adecuado, con el objetivo de cumplimentar la petición de invocación. Una vez que el motor ha seleccionado un proveedor delegado (utilizando mecanismos como los descritos en el capítulo 10), un servicio web real será invocado, a quien se le enviarán los parámetros de invocación según corresponda. Después de la ejecución del servicio, el motor VWS recibirá los parámetros de retorno del servicio real, y los reenviará de vuelta al cliente delegado que originó la invocación al servicio web virtual.

Para que el *cluster* funcione de forma precisa, el motor VWS debe decidir en cada momento cuál es el nodo de servicio más adecuado para ejecutar una petición de invocación. El motor, como hemos comentado antes, deberá utilizar mecanismos de selección que permitan maximizar el rendimiento del *cluster* en su conjunto.

11.3.2. Selección de nodos

Una vez que un nodo controlador recibe una petición, éste debe examinar la definición del método virtual (contenida en el documento VWSDL) y extraer de él una lista de proveedores delegados (nodos de servicio) que serán los candidatos a ejecutar el servicio que solicita el cliente. El motor VWS debe seleccionar el más adecuado antes de realizar (redirigir) la invocación recibida hacia un proveedor o nodo de servicio. Aquí entra en juego el sistema de *rating* que hemos descrito en capítulos anteriores.

Las facetas que nos interesará reflejar en las expresiones que manejarán los nodos controladores deberían estar relacionadas con temas relativos a la implementación de los proveedores delegados, como puede ser el tiempo de

respuesta, la disponibilidad, la latencia, el rendimiento, etc. En general, no será relevante el uso de facetas como el coste, o valoraciones sobre el proveedor del servicio, ya que, a la hora de construir un *cluster*, normalmente, todos los nodos estarán gestionados por el mismo proveedor y todos los nodos del mismo *cluster* ofrecerán los mismos servicios, que son los que se publican en el/los nodo controlador.

La existencia de un nodo controlador pudiera crear la sensación de la existencia de un punto único de fallo. Aunque no forma parte de este trabajo, nos gustaría constatar que la alta disponibilidad del nodo controlador se logra, como se hace en otros entornos utilizando direcciones IP virtuales [133] y repartidores de red [4].

Adicionalmente, es posible construir un *cluster* de servicios web con más de un nodo controlador. De hecho, no hay nada que ligue de forma estrecha los nodos controladores con los nodos de servicio. Es, únicamente, la existencia de los documentos VWSDL la que crea la sensación de *cluster*. Nos damos cuenta rápidamente, que la estructura de *cluster* de servicios web es una estructura muy flexible, pudiendo darse, entre otras, las siguientes circunstancias:

- Un nodo controlador puede serlo a la vez de varios *cluster* diferentes. Debido a que la sensación de *cluster* la dan los documentos VWSDL, dependiendo de cómo hayan sido construidos éstos, un mismo nodo controlador puede redirigir invocaciones a nodos de servicio completamente diferentes desde el punto de vista de su ubicación (intra- o extra-empresa) o su pertenencia al *cluster*.

- Un nodo de servicio puede serlo a la vez de varios *clusters*. Como en realidad, los nodos de servicios no dejan de ser proveedores delegados, un mismo nodo puede atender peticiones que le lleguen de distintos controladores al mismo tiempo.

- Un nodo de servicio puede responder a peticiones de invocación recibidas directamente de clientes delegados. Como acabamos de observar, un nodo de servicio, como proveedor delegado que es, puede servir peticiones que lleguen de un nodo controlador (un motor VWS) o de una aplicación cliente. En realidad, para el nodo de servicio no hay diferencia entre un nodo controlador y una aplicación cliente: ambos son percibidos como clientes delegados.

- Un *cluster* puede tener más de un controlador. Por las mismas razones que hemos expuesto hasta ahora, un *cluster* puede ser gestionado por cualquier número de nodos controladores. Como hemos comentado, la *"sensación de cluster"* la crean los documentos VWSDL con lo que, desplegándolos adecuadamente, podemos tener tantos controladores como deseemos.

Llevando todas estas propiedades de los nodos a un extremo, podemos construir nodos que sean a la vez nodos controladores y nodos de servicio, es decir, que reciban peticiones de invocación y las reencaminen, o que las reciban y las resuelvan directamente. Es decir, estaríamos ante la circunstancia de que un motor VWS puede atender a la vez peticiones de invocación de servicios reales (servicios web estándar) y de servicios web virtuales (publicados como servicios web estándar).

La posibilidad de crear elementos de la arquitectura que desarrollen un doble rol da lugar a la existencia de un tercer tipo de nodos en los *clusters* de servicios web: los **nodos mixtos**. Como hemos comentados antes, un nodo de este tipo podrá funcionar al mismo tiempo como nodo controlador (sirviendo peticiones virtuales) y como nodo de servicio (sirviendo peticiones reales). Los nodos mixtos nos permiten obtener una serie de funcionalidades adicionales en los *clusters* de servicios web:

- Posibilidad de construir *clusters* a base de nodos mixtos. Es decir, podemos construir un *cluster* en el que todos sus nodos sean capaces de controlar el *cluster* a la vez que son capaces de servir peticiones de invocación reales. Esto permite que cualquier nodo del *cluster* reciba peticiones y decida si desea reencaminarlas o resolverlas en función de su estado y del estado del resto del *cluster*.

- La utilización de nodos mixtos simplifica enormemente la gestión del *cluster*, ya que es factible eliminar un nodo del *cluster* (por decisiones de gestión o por mantenimiento) sin preocuparnos de cual es su rol dentro del *cluster* de servicios web. Es decir, podemos eliminar un nodo y el *cluster* sigue teniendo controladores.

Como ya hemos comentado en capítulos anteriores, el uso de expresiones permite a un motor VWS decidir hacia donde redirige una invocación de servicio web virtual. Esto, aplicado a *clusters* de servicios web, significa, que el/los nodo controlador, conocen en cada momento información estadística acerca de los nodos de servicio que son capaces de resolver una petición de invocación real.

La utilización de expresiones en los procesos de evaluación y selección dentro de los motores VWS presenta las mismas características que presentaría en otras aplicaciones como QoS o gestión de SLA. Es decir, podemos construir expresiones que incorporen expresiones complejas o expresiones temporales. Igualmente, es factible la utilización de variables externas, no gestionadas por los motores VWS cuyos valores son establecidos desde el exterior del mismo.

11.4. Gestión de un *cluster*

11.4.1. Mantenimiento del *cluster*

Para que un *cluster* de servicios web pueda alcanzar una disponibilidad del 100 %, deben existir mecanismos que permitan realizar su mantenimiento. Cuando realizamos mantenimiento, podemos encontrarnos con la posibilidad de tener que detener parte del sistema. Si queremos detener un nodo de servicio con el fin de realizar tareas de mantenimiento disruptivas (en cuanto al servicio) debemos disponer de un mecanismo que permita que los nodos del *cluster* dejen de seleccionar a dicho nodo como ejecutor de peticiones.

Para ello, la manera más sencilla se basa en utilizar las ya mencionadas variables externas. Tomemos como ejemplo la expresión 11.1, en donde la evaluación de nodos de servicio se basa en valorar el tiempo de respuesta, la disponibilidad y el caudal (*throughput*, peticiones atendidas por unidad de tiempo) de cada uno de los candidatos.

$$G = 0{,}34 * reverseAdjust(r) + 0{,}33 * adjust(a) + 0{,}33 * adjust(t) \quad (11.1)$$

Si hacemos que toda la expresión dependa de una variable externa, podremos influenciar el proceso de selección hasta conseguir que un nodo determinado no resulte nunca seleccionado. Esto se muestra en la expresión 11.2, donde la variable `upDown` representa el estado en que queremos que se encuentre el nodo. Si `upDown` vale 0 para un nodo n1, y 1 para el resto de los nodos, la evaluación de la expresión para n1 valdrá siempre 0, y para el resto de los nodos valdrá lo mismo que valdría la expresión original (11.1).

$$G = upDown * (0{,}34 * reverseAdjust(r) + 0{,}33 * adjust(a) + 0{,}33 * adjust(t)) \quad (11.2)$$

De esta forma, cuando queramos quitar un nodo del *cluster*, sólo necesitamos poner su valor correspondiente de `upDown` a 0. A continuación, esperar a que el nodo termine las ejecuciones pendientes, con lo que pasará de estar en estado operativo a estado de drenaje, y dejará de recibir nuevas peticiones de invocación, como resultado de no ser seleccionado por el nodo controlador. En cuanto la actividad cesa por completo en el nodo (el nodo ha entrado en estado de parada), el nodo se considera inactivo desde el punto de vista del *cluster* y se puede realizar sobre él cualquier tarea de mantenimiento, incluido el apagado físico.

Una vez que hemos realizado el mantenimiento pertinente, y el nodo esta disponible para ser reincorporado al *cluster*, bastará con volver a poner su `upDown` a 1, y el nodo empezará a recibir invocaciones de nuevo.

En realidad la variable `upDown` es una simplificación de lo que se conoce dentro de las tecnologías de *cluster* como **afinidad** [73, 64]. La afinidad representa la preferencia del *cluster* para dirigir una invocación hacia un nodo determinado, por razones ajenas a la propia gestión del *cluster*. Por ejemplo, la afinidad es utilizada por los servidores de aplicaciones en *cluster* para forzar que una sesión, iniciada en un nodo concreto del *cluster*, continúe ejecutándose en el mismo nodo cuando lleguen las peticiones subsiguientes, lo que permite que los objetos creados a partir de dicha sesión (alojados en la memoria de un nodo concreto) sean accesibles en todas las invocaciones relativas a una misma sesión. Supongamos la existencia de la variable externa `affinity`, gestionada por el usuario, incluida en una expresión de selección como la mostrada en la formula 11.3.

$$G = affinity * (0{,}5 * adjust(a) + 0{,}5 * adjust(t)) \tag{11.3}$$

Los distintos valores de la variable `affinity` para cada nodo del *cluster* harán que el sistema de selección y evaluación pueda ser condicionado de acuerdo a dichos valores. Si queremos llevar la afinidad a su caso extremo, es decir, forzar la afinidad a un nodo en todas y cada una de las invocaciones recibidas (por cuestiones relativas a la implementación de servicio web real, por ejemplo), dicha variable debería tomar valor uno para el nodo afín, y cero para el resto de los nodos. Es decir, llegamos a la ya mencionada variable `upDown`.

11.4.2. Detección de errores

Durante la operatoria normal de un *cluster* pueden surgir situaciones que, básicamente, podrían producir dos tipos de errores: incrementos no deseados del tiempo de respuesta y errores graves, como la caída de un nodo. El motor VWS (el nodo controlador) será el elemento responsable de tratar estos y otros errores que pudieran producirse.

Supongamos que el tiempo de respuesta ofrecido por un nodo de servicio determinado se ve incrementado más de lo normal. En este caso, el correcto funcionamiento del sistema de expresiones gestionado por el nodo controlador hará que dicho nodo deje de ser invocado, siempre que las expresiones tomen en cuenta el tiempo de respuesta como en el ejemplo anterior mostrado en la expresión 11.2.

En el segundo tipo de errores, si un nodo deja de responder a las peticiones, el motor VWS debería dejar de seleccionarlo para enviarle solicitudes

de invocación. Para tratar con este tipo de situaciones, el nodo controlador debería hacer uso de la ya mencionada variable `upDown`, o algo equivalente. De tal forma que si se detecta que el sistema no responde a las solicitudes de invocación, la variable `upDown` será puesta a cero.

¿Cuándo podrá el proveedor volver a utilizar el nodo de servicio en error? Procedimientos de test iterativos (como un PING (*Packet InterNet Groper*) o bucles de conexiones TCP) serán una forma adecuada de detectar la disponibilidad del nodo. Un método más apropiado sería aquel que se basase en notificaciones, en lugar de en sondeos (*polling*). Para ello, los motores VWS deben habilitar un servicio web (llamémosle `upDownService`) que permita a los nodos de servicio invocarlo para notificar cambios en su estado. En realidad, el servicio `upDownService` lo único que ha de hacer es modificar la variable `upDown` correspondiente al nodo de servicio que se le indique en los parámetros de entrada.

11.4.3. Escalabilidad

Los problemas de escalabilidad son solventados, habitualmente, mediante dos tipos de soluciones: **escalabilidad vertical** y **escalabilidad horizontal**.

En la escalabilidad vertical se trata de aumentar la capacidad del *cluster* mediante el aumento de la capacidad (memoria, CPU, etc.) de uno, varios o todos los nodos del *cluster*.

En la escalabilidad horizontal, se trata de aumentar la capacidad del *cluster* mediante el aumento del número de nodos. Es decir, la implementación física de cada uno de los nodos no se ve modificada, sino que se agregan nodos al *cluster*.

En el caso de los *clusters* de servicios web, si optamos por la escalabilidad vertical, no tendremos ningún problema, ya que los servicios web (virtuales o reales) son completamente independientes de la infraestructura en la que se alojan. Simplemente deberemos realizar el mantenimiento de acuerdo al procedimiento explicado en el apartado anterior (o algún procedimiento similar), y actualizar los equipos hardware convenientemente.

Pero si decidimos utilizar escalabilidad horizontal, debemos tener en cuenta lo que hemos denominado "*sensación de cluster*". Como hemos explicado anteriormente, dicha sensación la crean en realidad los documentos VWSDL, y no la configuración del hardware o del software de los elementos que componen el *cluster*. Es por ello que la escalabilidad horizontal puede parecer compleja, ya que la adición de un nuevo nodo al *cluster* consiste en realidad en modificar los documentos VWSDL, añadiendo elementos `invoke` a cada método. Aunque pueda parecer una labor tediosa, se puede

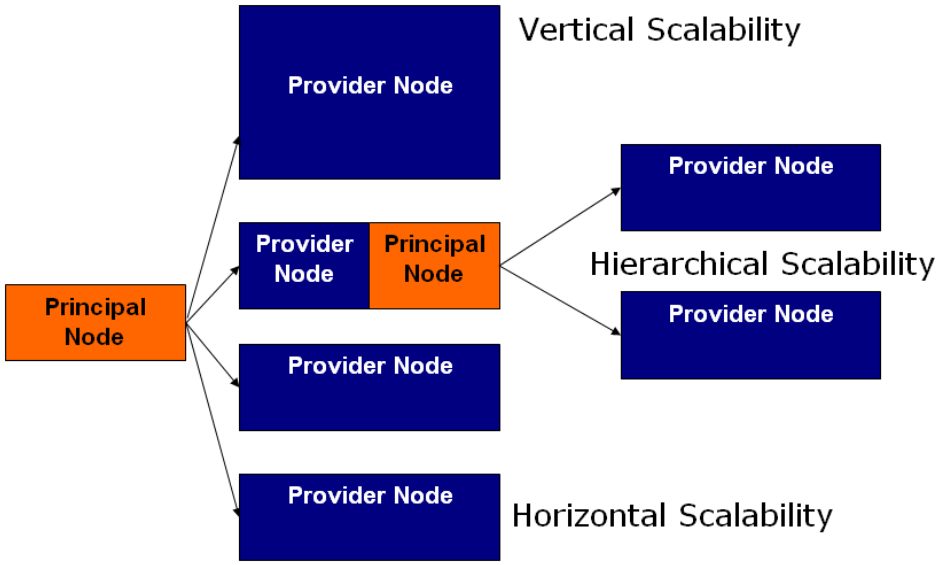

Figura 11.2. Tres tipos de escalabilidad.

automatizar de una forma muy sencilla. Recordemos que los documentos VWSDL son documentos basados en XML, y siguen un esquema concreto (el del lenguaje VWSDL). Es por esto que la modificación masiva de documentos VWSDL (y sus elementos `method` e `invoke`) resulta fácilmente automatizable mediante mecanismos de programación (manejando XML y XPath).

Independientemente de lo interesante que nos resulte la escalabilidad horizontal en un *cluster* de servicios web, nuestra propuesta permite la utilización de estructuras de *cluster* alternativas.

En nuestro modelo de virtualización, un *cluster* es interpretado como un árbol, cuya raíz es un motor VWS (el nodo controlador) y los proveedores delegados (nodos de servicio) son sus hojas. Así pues, la escalabilidad horizontal, tal como la acabamos de comentar, consiste en añadir hojas al árbol (figura 11.2).

Dentro de nuestra propuesta tiene cabida una tercera forma de escalabilidad: la que hemos denominado **escalabilidad jerárquica**. Cuando publicamos un servicio web virtual (como un servicio estándar), le asociamos uno (o más) servicios reales, que representan su implementación. Estos servicios son reales, es decir, se publican utilizando un documento WSDL que ha de ser utilizado por los clientes delegados para formalizar las invocaciones.

Pero en realidad, desde el punto de vista del cliente delegado, o de un motor VWS que invoca un servicio real, no hay diferencia entre un servicio

web real y uno virtual, ya que éste último se comporta como aquél. Es por ello que los nodos de servicio pueden ser en realidad motores VWS que publican servicios web virtuales. En este caso, un nodo de este tipo representaría nodos intermedios del árbol que constituye un *cluster* (figura 11.2).

Los nodos intermedios de un *cluster* no son nodos mixtos, aunque puedan serlo. Son, por definición, motores VWS que redirigen las invocaciones hacia otros nodos, es decir, son nodos de servicio desde el punto de vista del controlador del *cluster* que recibe una petición de invocación de un cliente delegado; pero son a la vez nodos controladores de un *cluster* hacia el que se reencaminan las peticiones de invocación.

La diferencia entre nodo mixto y nodo intermedio es clara. Mientras el primero puede recibir y ejecutar peticiones de invocación, el segundo las recibe de un nodo controlador y las reencamina hacia nodos de servicio. Es decir, el nodo mixto presenta los dos roles simultáneamente (controlador y servicio), mientras el nodo intermedio presenta un rol hacia un *cluster* al que pertenece (rol de nodo de servicio) y otro rol distinto, el de controlador, hacia un *cluster* que gestiona.

11.5. Validación de la propuesta

La solución al problema de la alta disponibilidad que proporciona la arquitectura propuesta ha sido validada mediante la construcción de un prototipo cuya arquitectura se muestra en la figura 9.1. El prototipo desarrollado no implementa todas las funcionalidades que requiere nuestra propuesta de lenguaje de descripción de VWS, pero si las necesarias para comprobar su correcto funcionamiento.

El experimento consistió en la realización de distintas pruebas y simulaciones a partir de los siguientes componentes:

1. Una aplicación cliente que será la encargada de emular la carga de trabajo, es decir, realizar invocaciones masivas a servicios web virtuales publicados en nuestro prototipo.

2. Un motor VWS, que es nuestro prototipo.

3. Un simulador de problemas, que será el componente por el cual podemos recrear distintas situaciones de error en los proveedores delegados.

4. Dos proveedores delegados, dos servicios web, que serán los que se utilicen para construir el servicio virtual que será publicado en el motor VWS.

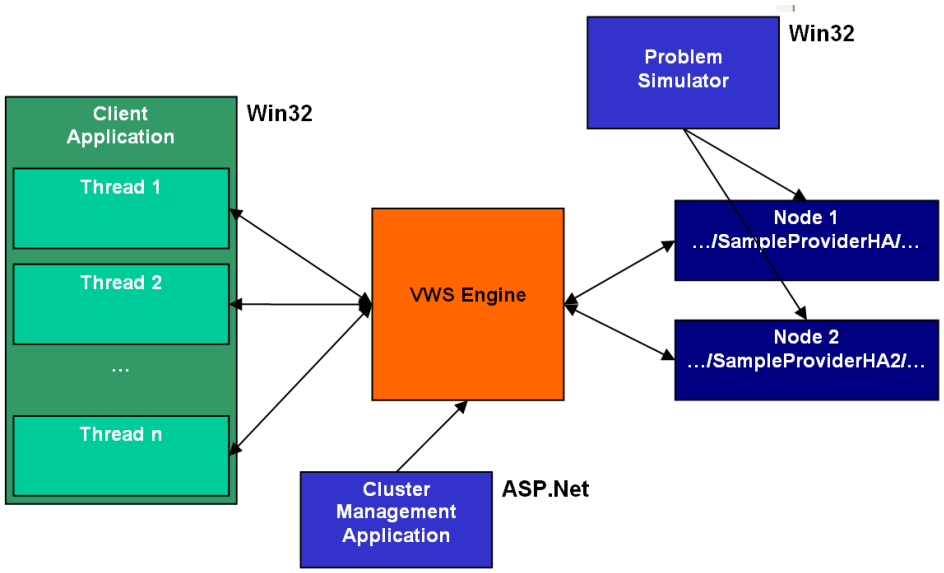

Figura 11.3. Arquitectura del experimento.

5. Una aplicación para control del *cluster* que permite realizar las tareas de configuración, operación, análisis, etc.

Los componentes que acabamos de enumerar se comunican formando la arquitectura mostrada en la figura 11.3. Pero, en realidad, no hemos construido el experimento como un conjunto de 5 componentes independientes, sino que hemos desarrollado tres componentes básicos que aglutinan a los 5 componentes originales de la forma que sigue (mostrada en la figura 11.4):

1. Una aplicación Win32 (a la que nos referiremos como *Naoussa*), que será la responsable de realizar la ejecución masiva de invocaciones hacia el servicio virtual. Al mismo tiempo, dicha aplicación se utiliza como simulador de problemas, permitiendo configurar los tiempos de respuesta que presentarán los servicios reales con los que se ha construido el servicio web virtual que invoca el lanzador masivo. *Naoussa* tiene el aspecto de una sencilla aplicación Windows, tal como se muestra en la figura 11.5. En la ventana se aprecian dos partes bien diferenciadas. La parte izquierda es el lanzador masivo de transacciones, y muestra continuamente el resultado de las ejecuciones. La parte derecha es la que corresponde con el simulador de problemas, y está mostrando continuamente el estado de cada uno de los proveedores delegados en uso (invocaciones recibidas, tiempos de respuesta, etc.), denominados `HA` y `HA2`.

2. Una aplicación web, desarrollada como aplicación ASP.Net, que está

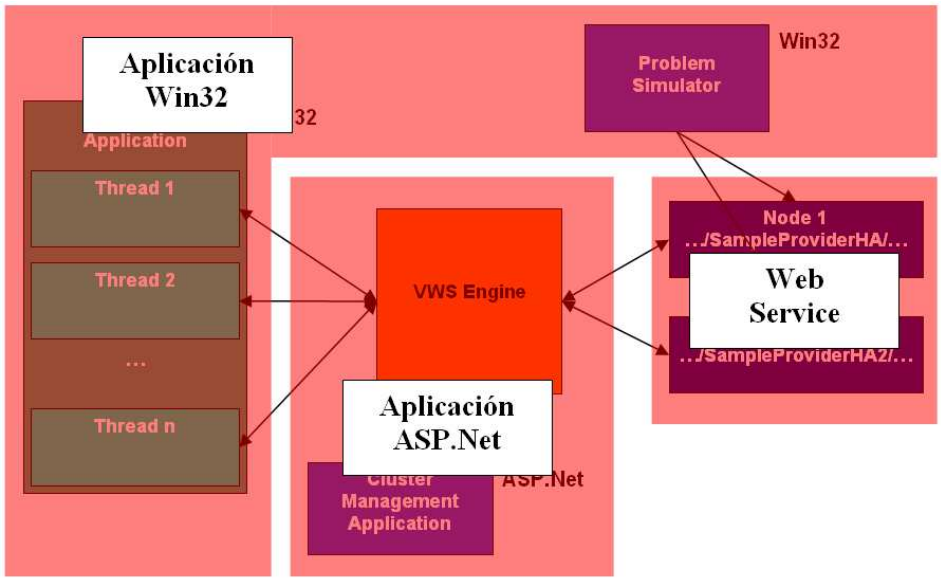

Figura 11.4. Esquema software del experimento.

formada por dos componentes principales:

- Un motor VWS, implementado como una aplicación web que es invocada a través de una URL determinada, capaz de recibir invocaciones virtuales, redireccionarlas a servicios web reales y enrutar las respuestas correspondientes a las aplicaciones cliente.
- Una pequeña aplicación web que permite configurar el motor VWS, así como publicar servicios web virtuales, o examinar la operativa del motor.

3. Dos servicios web idénticos, pero accesibles a través de dos URL distintas, y con hilos de ejecución completamente independientes.

Con los componentes que acabamos de describir, hemos llevado a cabo tres experimentos que enumeramos a continuación:

1. **Reparto de carga**. La prueba consistió en demostrar que el motor VWS, con las expresiones adecuadas, es capaz de repartir la carga entre los distintos proveedores utilizados para la implementación del servicio virtual.

2. **Controlador de** *cluster*. Para comprobar la efectividad del motor VWS como nodo controlador, se introdujo un aumento significativo en el tiempo de respuesta de uno de los proveedores delegados. Esto se hizo utilizando el simulador de problemas.

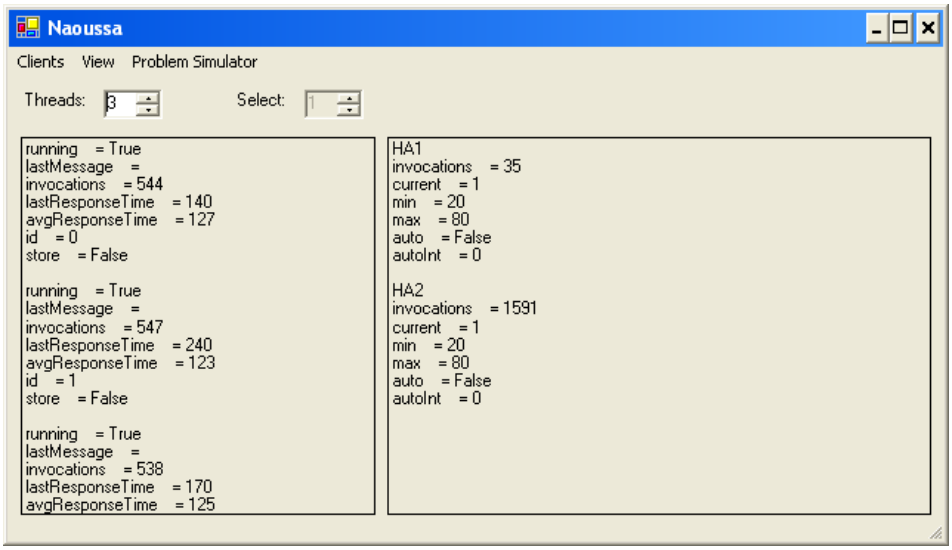

Figura 11.5. Aplicación para la ejecución de las pruebas.

3. **Parada de uno de los proveedores delegados**. Utilizando el simulador de problemas, detuvimos por completo uno de los proveedores delegados, con el fin de comprobar que el motor VWS dejaba de seleccionar a dicho proveedor, al ser un proveedor erróneo.

Para la realización de la prueba configuramos *Naoussa* para que realizara 10.000 invocaciones desde tres clientes hacia un servicio web virtual implementado con dos proveedores. Para analizar los resultados, guardamos un fichero con el número de invocaciones por intervalo recibidas en cada proveedor. Estos datos son los que se reflejan en el gráfico mostrado en la figura 11.6. Con los datos recogidos construimos un segundo gráfico con los valores acumulados, con el fin de facilitar la interpretación de los resultados. Este segundo gráfico es el que se muestra en la figura 11.7

En ambas figuras hemos señalado dos momentos importantes (A y B). Entre el inicio de la prueba y el momento A, los proveedores delegados tuvieron un tiempo de respuesta similar, con lo que el número de invocaciones recibidas por ambos era prácticamente igual. En el momento A, el tiempo de respuesta del segundo proveedor pasó de oscilar entre 20 y 80 milisegundos a hacerlo entre 50 y 90. Como vemos, entre los momentos A y B, el segundo proveedor fue invocado un menor número de veces.

Finalmente, en el momento B, el segundo proveedor fue configurado (utilizando como antes el simulador de problemas) para ofrecer un tiempo de respuesta superior a un minuto. A partir de ese momento, el motor dejó de seleccionarlo y todas las invocaciones fueron a parar al primer proveedor.

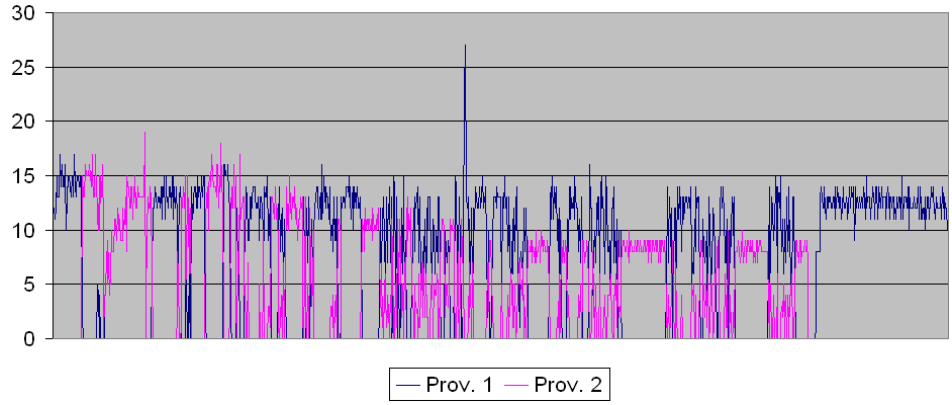

Figura 11.6. Número de invocaciones por proveedor a intervalos.

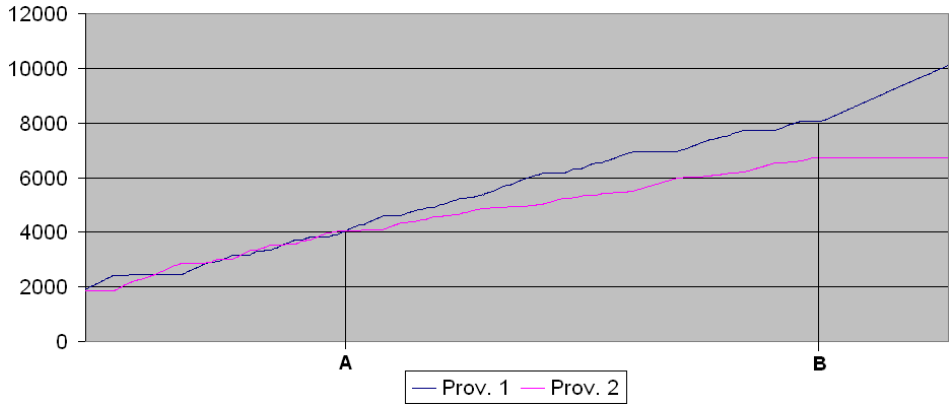

Figura 11.7. Número de invocaciones acumulado por proveedor.

Cada uno de los tres experimentos que quisimos probar está reflejado en la prueba que acabamos de describir:

1. **Reparto de carga**. Entre el momento inicial y el momento A, el motor seleccionó, prácticamente sin distinción, a ambos proveedores.
2. **Controlador de** *cluster*. Entre el momento A y el momento B el motor mostró sus preferencias por el primer proveedor, puesto que era el que estaba dando mejor tiempo de respuesta
3. **Parada de uno de los proveedores delegados**. A partir del momento B, el motor dejó de seleccionar el segundo proveedor, puesto que dejó de responder a las peticiones dentro del intervalo permitido.

11.6. Conclusiones

Nuestra propuesta de virtualización aplicada a los problemas de alta disponibilidad produce resultados muy satisfactorios, como hemos comprobado en nuestros experimentos. Además, la creación de *clusters* de servicios web, y no de servidores de aplicaciones de servicios web, ofrece una forma muy flexible de implementar el *cluster*.

Lo que conseguimos con la creación de *clusters* de servicios web es en realidad elevar un nivel el punto en el que se construye el *cluster*, alejándolo de su implementación física para acercarlo a la lógica de negocio a la que ha de dar soporte.

Este acercamiento a la lógica de negocio nos brinda un conjunto de oportunidades que no serían alcanzables si el *cluster* se construye en un nivel más bajo. Por ejemplo, gracias a la utilización de servicios virtuales podemos proteger el interfaz de los servicios web reales. La protección del interfaz representa una mejora sustancial en los procesos de mantenimiento de las aplicaciones servidoras, ya que permite modificar el interfaz en los nodos del *cluster* mientras las aplicaciones cliente siguen invocando el mismo interfaz con el que fueron desarrolladas, sin necesidad alguna de modificación.

La protección del interfaz nos permite obtener nuevas funcionalidades muy interesantes dentro del *cluster*, debido al hecho de poder mantener distintos interfaces en cada uno de los nodos de servicio, y publicando un interfaz único y distinto en el nodo controlador:

- Simplificación de los procesos de *deploy* o despliegue. El hecho de poder utilizar interfaces distintos en cada uno de los nodos de servicio nos permite introducir modificaciones a la implementación y/o al interfaz de un servicio publicado en uno de los nodos, manteniendo el

interfaz del resto de los nodos de servicio y el publicado en el controlador inalterados. Esto facilita el manejo de versiones de software de proveedor (servicios web, al fin y al cabo) dentro del *cluster*, a la vez que permite hacer una introducción paulatina de nuevas versiones.

- Posibilidad de construir *clusters* de servicios web a partir de servicios web con distinto interfaz. Es decir, no es necesario desarrollar un servicio web y publicarlo en un conjunto de nodos, podemos construir un servicio web de alta disponibilidad por agregación de servicios web existentes, ofrecidos por distintos proveedores.

Otra ventaja de la construcción de *clusters* de alto nivel es la posibilidad de construir *clusters* heterogéneos. En los sistemas *cluster* tradicionales, no es posible crear un *cluster* de servidores de aplicaciones entre un Oracle IAS y un IIS, o un iPlanet y un Tomcat, etc., ya que la tecnología subyacente en cada uno de estos servidores de aplicaciones está ligada al fabricante y es altamente dependiente de él, debido a la ausencia de estándares en las implementaciones de sistemas *cluster*.

Por el contrario, cuando hablamos de servicios web, la utilización de estándares no es un objetivo o una meta a perseguir, es en realidad el punto de partida. No se concibe la tecnología de servicios web sin la utilización de estándares. Es por esto que nuestra propuesta permite la construcción de *clusters* heterogéneos, ya que lo único que liga a cada uno de los nodos de un *cluster* de servicios web (y a su/s nodo/s controlador/es) es un lenguaje estándar, el WSDL, y una expresión que define el proceso de selección.

La heterogeneidad de los nodos de un *cluster* nos permitiría, en contra de lo que es habitual en los sistemas *cluster* tradicionales, una hipotética construcción de un *cluster* de servicios web que se extienda a lo largo de Internet. Es decir, no hay nada en nuestra tecnología de virtualización que obligue a que la construcción del *cluster* se haga utilizando tecnología de Intranet, o de red local en general. En realidad, las únicas limitaciones vendrían dadas por los tiempos de respuesta o la utilización de un entorno menos seguro. Esto último podría paliarse combinando el uso de VWSDL y WSDL con el de otras tecnologías como WS-Security [47].

Es importante recalcar que la existencia de un nodo controlador no limita la operatividad ni la alta disponibilidad de nuestro sistema de *clusters* de servicios web, ya que, como hemos comentado anteriormente, es posible construir *clusters* de servicios web con más de un nodo controlador, llegando incluso a ser posible el caso extremo en el que todos los nodos del *cluster* sean controladores.

Esta flexibilidad en la arquitectura con la que se construyen los *clusters* puede llevarnos incluso a situaciones todavía más curiosas, como la posibilidad de disponer de nodos que pertenezcan a más de un *cluster*. En nuestra

definición de *cluster* de servicio web, lo único que une a un nodo controlador con uno de servicio son las definiciones contenidas en los documentos VWSDL. Por lo tanto, nada impide que un nodo de servicio lo sea a la vez de más de un nodo controlador. O incluso, como hemos comentado anteriormente, que un nodo del *cluster* sea a la vez un nodo de servicio y un nodo controlador.

Aunque no ha sido comentado con anterioridad, y es un tema ligeramente ajeno al alcance de este trabajo, nos gustaría resaltar que la disponibilidad de los nodos controladores (independientemente del modelo que se haya escogido para un *cluster*, con un nodo controlador o más de uno) dentro de nuestro modelo de virtualización, se consigue mediante la introducción de motores VWS redundantes y repartidores de red.

CAPÍTULO 12

Calidad de Servicio

12.1. Introducción

La calidad de servicio (QoS, *Quality of Service*) engloba un conjunto de técnicas y tecnologías utilizadas con el objetivo de poder predecir los resultados de una acción concreta. Un sistema que utilice técnicas de QoS puede no llegar a alcanzar dichos resultados en un momento dado, pero dichas técnicas permitirán al sistema ejecutar las acciones adecuadas para poder alcanzar los mencionados resultados predecibles.

Todas las actividades necesarias para aplicar QoS a un sistema implican una necesidad de control sobre todas las partes que lo componen, por lo que se hace necesario un sistema independiente que permita predecir los resultados de cada una de las partes intervinientes en el sistema.

Para poder predecir los resultados es necesario, en primer lugar, medirlos y, en segundo lugar, evaluar dichas mediciones. Dicho de otra manera, no podemos garantizar resultados predecibles si no somos capaces de medir el sistema y cada una de las partes que lo componen. Y si se producen desviaciones en las mediciones (las medidas se alejan de la predicción), deberemos evaluar dichas medidas para corregir (modificar) el funcionamiento del sistema para, de esta forma, mantener el sistema en funcionamiento dentro de los márgenes admitidos por los resultados predecibles fijados de antemano.

En resumen, un proceso de calificación (medición) y uno de evaluación serán necesarios para poder aplicar técnicas de calidad de servicio a un sistema. Ambos procesos, calificación y evaluación, serán los procesos clave que dirigirán un sistema de calidad de servicio.

12.2. Calificación y Evaluación

Como hemos comentado en capítulos anteriores, las facetas (variables) son utilizadas dentro de nuestra propuesta con el objetivo de lograr la automatización de dos procesos:

- **Proceso de calificación**. Un motor VWS obtiene, computa y almacena información estadística en una base de datos después de seleccionar e invocar un proveedor delegado.
- **Proceso de evaluación**. Un motor VWS lee información estadística de una base de datos con la intención de seleccionar e invocar el proveedor delegado más adecuado en cada momento.

Ambos procesos están relacionados indirectamente por el acceso que realizan a una base de datos de información estadística. El proceso de actualización de esta base de datos debe ser realizado de una forma automatizada. Para que esto sea posible, es necesario que las variables utilizadas en el proceso puedan ser obtenidas en el instante siguiente a una invocación de un proveedor delegado.

Por ejemplo, el motor VWS no puede actualizar una variable que represente el tiempo invertido por un proveedor de servicio en atender los pedidos, ya que este tiempo queda fuera del alcance de lo que se circunscribe a una invocación del proveedor delegado correspondiente. En realidad, las variables que no pueden ser obtenidas a partir de una invocación de un proveedor delegado suelen ser variables relacionadas con el proveedor del servicio, como el tiempo de envío de un pedido (véase capítulo 10 para consideraciones relativas a los tipos de facetas o variables).

El conjunto creado por las facetas, las expresiones y los procesos de calificación y evaluación, conforma la infraestructura necesaria para construir un sistema de calidad de servicio. Utilizando estos tres componentes, podemos desarrollar sistemas que ofrezcan resultados predecibles o, cuando menos, se ajusten por sí mismos para acercarse en la medida de lo posible a dichos resultados.

¿Cómo construimos un sistema de calidad de servicio? En primer lugar es importante destacar que las garantías de calidad de servicio, dentro de nuestra propuesta, se aplican específicamente a cada método de un servicio web virtual, y no al servicio web virtual en su conjunto.

Nuestra arquitectura permite la construcción de expresiones que permiten a los clientes especificar las propiedades del servicio web a utilizar o del proveedor más adecuado a sus necesidades. La gestión de la calidad de servicio basada en el uso de expresiones permite a los creadores de servicios

web virtuales construir diferentes **clases de servicio** que serán ofrecidas a los clientes. Cada clase de servicio será identificada por una expresión que permitirá seleccionar el proveedor delegado y el proveedor de servicio más adecuados a las necesidades de cada cliente y cliente delegado.

Como hemos comentado en capítulos anteriores, es muy importante remarcar el hecho de que las clases de servicio no están solamente relacionadas con los servicios web reales (proveedores delegados según nuestra arquitectura), sino que una clase de servicio puede ser utilizada para tener en cuenta características asociadas al propio proveedor del servicio, es decir, la entidad que ofrece algún tipo de servicio a través de un servicio web.

Según nuestra propuesta, las técnicas de calidad de servicio serán aplicadas a cada proveedor delegado y, además, a un método (`port` de WSDL) específicamente. La razón para esto es que la implementación de un método virtual (definido dentro de un servicio web virtual) se hace mediante un conjunto de proveedores delegados que ofrecen un servicio web y un puerto WSDL equivalentes en funcionalidad. La forma en la que el motor VWS elige el proveedor más adecuado en cada momento se basa en el ya comentado uso de expresiones. Estas expresiones son las que nos ofrecerán los resultados predecibles.

Como hemos explicado en capítulos anteriores, cuando un motor VWS tiene que seleccionar un proveedor delegado, debe aplicar un proceso de evaluación y uno de selección. El proceso de evaluación, descrito previamente, dará como resultado una lista de proveedores delegados y un valor que corresponde con su puntuación de acuerdo a la expresión utilizada.

Después del proceso de evaluación debe realizarse el proceso de selección. En principio un motor VWS debería escoger siempre el de puntuación más elevada de acuerdo a la expresión utilizada en cada momento. Pero en realidad, el motor podría no seleccionar el mejor, y escoger algún otro proveedor delegado de la lista de candidatos siempre que este cumpla las condiciones requeridas para ser candidato.

¿Cuales son las condiciones que deben cumplir los candidatos? Pues dependerá de las necesidades de los clientes delegados. Como hemos mencionado en capítulos anteriores, los clientes pueden utilizar VWSEL para expresar sus preferencias a la hora de elegir proveedores. Dichas preferencias pueden ser expresadas en términos de clases de servicio y, al mismo tiempo, contener un conjunto de condiciones que deben cumplir los candidatos.

12.3. Validación de la propuesta

Hemos utilizado *Naoussa*, nuestro simulador de cliente delegado, para realizar pruebas de nuestra propuesta de arquitectura de QoS. Para ello he-

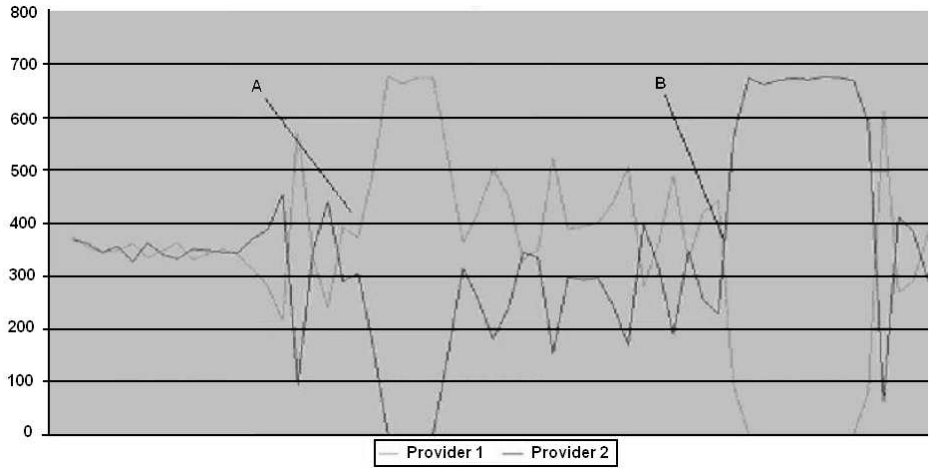

Figura 12.1. Invocaciones recibidas por proveedor delegado.

mos desarrollado dos proveedores delegados y, sobre ellos, hemos construido un servicio web virtual que se basa en la expresión 12.1 para los procesos de evaluación y selección de proveedor, donde I el número total de invocaciones realizadas hacia un proveedor, C es el coste de cada invocación (lo que el proveedor de servicio factura por ejecutar un servicio), y R es el tiempo de respuesta de un proveedor delegado. La faceta I ha sido introducida en la expresión para disponer de un mecanismo de reparto entre todos los proveedores utilizados, es decir, los proveedores menos invocados obtienen valores elevados en la expresión `reverseAdjust(I)`, lo que con el paso del tiempo tiende a igualarlos en el proceso de selección, siempre teniendo en cuenta el valor del resto de variables utilizadas en la expresión.

$$0{,}4 * reverseAdjust(I) + 0{,}4 * reverseAdjust(C) + 0{,}2 * reverseAdjust(R) \tag{12.1}$$

En la figura 12.3 podemos ver los resultados de la ejecución de nuestro servicio web virtual. El eje X representa el tiempo y en el eje Y se muestra el número de invocaciones recibidas por cada proveedor delegado. En el punto A de dicha figura, el coste del proveedor 2 fue incrementado en un 50 %, y en el punto B, el coste volvió a su valor de partida. Podemos pues observar como el sistema tiende a estabilizar el número de invocaciones a la vez que tiene en cuenta el coste y el tiempo de respuesta.

12.4. Conclusiones

Nuestra arquitectura basada en virtualización no impone ningún tipo de restricción a la hora de implementar un modelo de calidad de servicio: el rendimiento global de la solución propuesta dependerá fundamentalmente de las facetas y la formulación utilizadas para construir el sistema.

Debe tenerse en cuenta que el motor VWS introduce una ligera sobrecarga (*overhead*) dentro del proceso de ejecución, ya que las peticiones deben ser recibidas y enrutadas hacia el proveedor delegado apropiado. Sin embargo, esta sobrecarga es insignificante en comparación con los beneficios obtenidos con nuestra propuesta de arquitectura de QoS. Recordemos que el objetivo del uso de las técnicas de la calidad de servicio (con o sin elementos intermedios) es poder ofrecer resultados predecibles, y eso es lo que permite nuestra propuesta.

Un componente adicional, el repositorio de *rating*, puede ser añadido a la arquitectura propuesta, como hemos comentado en capítulos anteriores, permitiendo que distintos proveedores delegados, u otros repositorios, se nutran de los datos estadísticos ya almacenados, pudiendo llegar a la ya mencionada arquitectura de recomendación.

Finalmente, nos gustaría señalar que nuestra propuesta, en contra de lo que pudiera parecer, no es una propuesta orientada hacia el centralismo, ya que cada componente de la arquitectura estándar puede decidir si utilizar un motor VWS o no, dependiendo de las funcionalidades que necesite de sus proveedores en cada momento.

CAPÍTULO 13

Mejoras de rendimiento

13.1. Introducción

Aparte de las funcionalidades principales de nuestra arquitectura (alta disponibilidad y calidad de servicio), existen otras funcionalidades de menor dimensión que permiten a los motores VWS obtener mejoras de rendimiento en base al uso de técnicas de *caching*. De acuerdo a este tipo de técnicas, los motores pueden capturar todos los mensajes entrantes y salientes, almacenarlos y, cuando sea posible, responder peticiones de clientes delegados sin necesidad de invocar a proveedores delegados. Estamos hablando de utilizar prácticas que ya son de uso común en otros entornos, como por ejemplo en los servidores de acceso a Internet (*proxies*), o en sistemas de mejora de rendimiento de un sistema operativo (mantener en memoria estructuras de directorios, por ejemplo), o de servidores de bases de datos, por mencionar algunos casos.

Evidentemente, los sistemas de *caching* no son siempre utilizables, ya que se basan en el uso de información que sufre pocos cambios a lo largo del tiempo, y esto no ocurre con cualquier tipo de aplicación.

Es fácil comprender como implementar un sistema de *caching* cuando hablamos de páginas web, o de recursos web en general (imágenes, sonidos, etc.). Pero, ¿cómo aplicar dichas técnicas a un entorno de servicios web? En el primer caso estábamos hablando de información estática, pero en el segundo, la información que se intercambia entre clientes y servidores es siempre información dinámica. Basándonos en nuestra arquitectura de servicios web virtuales, nosotros hemos creado una solución para implementar *caching* en un entorno de servicios web.

Como prerrequisito para la implementación de nuestra propuesta, los servicios web sobre los que queramos implementar estas técnicas deben cum-

plir las siguientes condiciones:

- El número de peticiones de invocación para un servicio web determinado debe ser suficientemente elevado para que la mejora de rendimiento sea efectiva.
- Existe al menos un método en el servicio web que queremos optimizar cuyas respuestas a lo largo del tiempo se producen sin variación alguna, o con muy pocas variaciones.
- Un método susceptible de utilizar los mecanismos de *caching* no debe producir efectos colaterales inherentes a su ejecución, como actualización de una base de datos o un fichero, salvo que dicha actualización no sea representativa para el negocio (como escribir un fichero de *log*, por ejemplo).

Algunos ejemplos de aplicaciones que podrían beneficiarse de mecanismos como los que vamos a describir serían:

- Un servicio web que ofrece la previsión del tiempo.
- Un servicio web que ofrece la cotización en bolsa de un valor. Aunque este es un servicio típicamente *online*, es decir, dinámico, considérese la situación que se plantea cuando se realizan invocaciones entre el cierre del mercado un día y la apertura consiguiente el día posterior.

13.2. Arquitectura

Se pueden distinguir fácilmente dos arquitecturas de *caching* (figura 13.1). La primera sería una arquitectura de *caching* en dos niveles (2LCA) que tiene en cuenta la existencia de únicamente dos entidades: cliente y servidor. Esta es una estructura típica de *caching*, como la utilizada por un navegador web que se conecta directamente a un servidor web, sin intermediarios. Una segunda estructura sería aquella que utiliza tres niveles (3LCA), en donde un tercer elemento, el intermediario, se añade a la 2LCA. Un ejemplo clásico de esto lo representan los servidores *proxy* de acceso a Internet, que realizan labores de *caching* entre navegadores y servidores web.

Dependiendo de la implementación que se haga de los elementos intervinientes en una 3LCA podemos distinguir dos subtipos:

- Arquitectura homogénea 3LCA (HoCA). Los tres componentes se comunican utilizando los mismos protocolos. Por ejemplo, un navegador web, un servidor *proxy* y un servidor web.

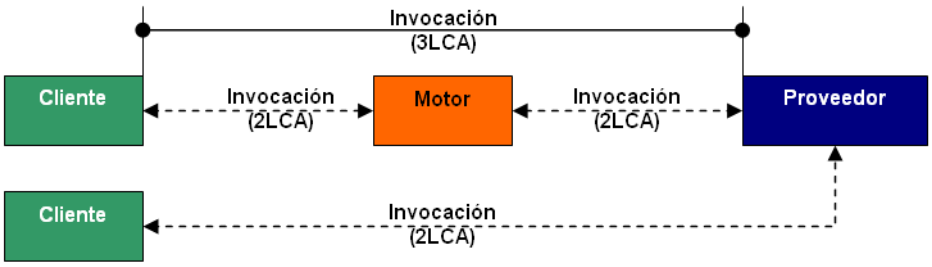

Figura 13.1. Arquitectura de *caching*.

- Arquitectura heterogénea 3LCA (HeCA). Dos de los componentes utilizan el mismo protocolo de comunicación, mientras un tercero utiliza uno distinto. El componente intermedio, obviamente, dialoga con los otros dos componentes utilizando ambos protocolos. Un ejemplo de esto sería una aplicación cliente que llama a un componente remoto EJB, quien a su vez invoca un servicio web.

La razón para la diferenciación entre HeCA y HoCA radica en la forma en la que las técnicas de *caching* podrán ser aplicadas.

Una arquitectura de *cache* de n niveles también puede existir, pero siempre será una combinación de 2LCA y/o 3LCA. En realidad, una arquitectura de n niveles puede ser vista como un sistema jerárquico de *caching*, donde los clientes delegados son las hojas, la raíz es el proveedor delegado y los nodos intermedios estarán formados por enrutadores SOAP o motores VWS.

13.3. Implementación

Como hemos comentado en capítulos anteriores, la implementación de nuestra propuesta de *caching* se basa en la utilización de elementos SOAP-Header. Lo que proponemos es añadir información de *caching* a las cabeceras de los mensajes SOAP, que permitan a los clientes tratar adecuadamente cada mensaje, evitando invocaciones innecesarias en la medida de lo posible. La información contenida en las cabeceras SOAP debe estar escrita en lenguaje VWSEL. Es importante destacar que los clientes deben entender VWSEL para poder beneficiarse de nuestra propuesta de *caching*. Es igualmente importante tener en cuenta que aquellos clientes que no entiendan VWSEL pueden trabajar sin problemas aunque reciban algún mensaje con información de *cache*, ya que ésta será despreciada y el cliente funcionará como si no existiese dicha información.

Los servicios web son los responsables de decidir qué información será

incluida en la cabecera de los mensajes SOAP que devuelvan a los clientes que los han invocado. Dichos servicios deben utilizar VWSEL para enviar información de *caching* a los clientes. Un documento VWSEL puede ser utilizado para expresar un periodo de tiempo en el cual el mensaje SOAP que porta dicho documento es válido como respuesta a la invocación que le dio origen. Por lo tanto, una invocación subsiguiente del mismo servicio dentro de dicho periodo de tiempo con los mismos datos de invocación sería inútil, ya que la respuesta no variaría.

Aunque los mecanismos de *caching* no son nada nuevo, creemos que existe una necesidad sin cubrir dentro de la actual arquitectura de servicios web, ya que consideramos que los servicios web no deberían depender de su capa de transporte (HTTP actualmente) para utilizar mecanismos de *caching*. El responsable de decidir qué mensajes pueden ser almacenados en una *cache* no debería ser el protocolo subyacente, tal como se propone en [111], sino el propio servicio web. ¿Qué ocurriría con el *caching* basado en HTTP si la capa de transporte de los servicios web deja de estar basada en HTTP para basarse en otro protocolo?

Nosotros proponemos pasar el control de los mecanismos de *caching* desde los elementos que controlan el protocolo subyacente (los servidores web y los clientes web al fin y al cabo) a los desarrolladores de servicios web. De esta forma diferentes invocaciones sobre el mismo *port* pueden tener distintas respuestas de *caching*. Pensemos por ejemplo en un servicio que devuelve las cotizaciones de valores en bolsa. La respuesta de *caching* será muy distinta dependiendo de si el mercado está abierto o cerrado.

13.4. Métodos de *caching*

La utilización del *caching* de manera efectiva puede requerir una eventual actualización de los mensajes alojados en las memorias *cache* de los clientes para mantener la consistencia entre los clientes delegados y los proveedores delegados. Se puede definir dos tipos de consistencia cuando consideramos un entorno desacoplado como el de los servicios web [60]: consistencia débil y consistencia fuerte. Cuando trabajamos con consistencia débil, puede ocurrir que un cliente esté utilizando un mensaje desactualizado como resultado de utilizar su *cache* de mensajes SOAP. En la consistencia fuerte, los mensajes alojados en las memorias *cache* de los clientes y las respuestas entregadas por los servicios web siempre son consistentes.

La existencia de dos modelos de consistencia exige el uso de dos métodos de *caching*. Al utilizar consistencia débil, los proveedores delegados deben enviar a los clientes delegados información acerca de la caducidad de los mensajes que les envían. Esto es lo que denominamos *caching débil*. De esta forma, los clientes delegados pueden utilizar mecanismos del tipo TTL

(*Time To Live*) para determinar si un mensaje alojado en la *cache* sigue siendo válido o es necesaria la invocación de un proveedor delegado para refrescar la *cache*.

La implementación de los métodos de *caching* débil es muy sencilla, pero, por el contrario, puede producir la utilización de mensajes incorrectos en el lado de los clientes delegados. Para evitar este tipo de problemas en entornos en los que la autenticidad de los mensajes sea vital, proponemos el uso del *caching fuerte*.

La implementación del *caching* fuerte se basa única y exclusivamente en el uso de dos mecanismos: el **sondeo** (*polling*) y la **invalidación**. Cuando se utilizan técnicas de sondeo, los clientes delegados son los responsables de interrogar a los proveedores delegados acerca de la validez de un mensaje SOAP determinado. Para que esta técnica produzca el uso de mensajes consistentes, es necesario que los clientes delegados interroguen a los proveedores delegados cada vez que se dispongan a utilizar una copia de un mensaje alojado en su *cache*. Esta interrogación origina el envío de un mensaje del cliente delegado al proveedor delegado, con la consiguiente latencia de red, independientemente de si la copia de la *cache* es válida. A pesar de esta inconveniencia, las técnicas de sondeo pueden ser muy útiles cuando los mensajes SOAP son muy largos, o cuando el tiempo de proceso en el proveedor delegado es también muy elevado.

El segundo mecanismo, invalidación, es un método de *caching* más complejo, donde los proveedores delegados informan a los clientes delegados acerca de la validez de un mensaje SOAP concreto. Utilizando este método, los clientes no tienen que preocuparse por la validez de la información alojada en su *cache*, ellos deben asumir siempre que la información es válida.

Para que la invalidación funcione, los proveedores delegados deben tener comunicación, directa o indirectamente, con los clientes delegados, con el fin de informarles de los cambios que se pudieran producir en el contenido de su *cache*. Esta circunstancia podría producir mensajes inválidos, debido a la eventual imposibilidad de un proveedor para informar de la invalidación de un mensaje a los clientes delegados que tienen copias de dicho mensaje enviados con anterioridad por aquel proveedor.

Para implementar la invalidación proponemos el uso de dos mecanismos: **invalidación relajada** e **invalidación segura**. Cuando un proveedor delegado utiliza invalidación relajada, lo único que ha de hacer para invalidar un mensaje SOAP enviado previamente, es enviar un mensaje de invalidación, sin esperar respuesta por parte del cliente delegado. Debe quedar claro que los clientes son los responsables últimos de eliminar los mensajes de su *cache*. Este mecanismo es útil cuando se utilizan servicios web de "casi-sólo-lectura", es decir, que apenas produzcan modificaciones en los mensajes de respuesta a lo largo del tiempo.

La invalidación segura se utiliza cuando queremos garantizar la consistencia a toda costa, es decir, el proveedor delegado decide perder parte de las mejoras de rendimiento en favor de una consistencia 100 % fiable. La invalidación segura se basa en el uso de dos mecanismos para invalidar mensajes: una invalidación con respuesta y un mecanismo de TTL. Es el mecanismo que hemos denominado iTTL (invalidación con TTL).

Cuando un proveedor delegado desea invalidar un mensaje SOAP que ha sido previamente introducido en la *cache* de algunos clientes debe, primeramente, enviar notificaciones de invalidación a todos los clientes que contengan una copia del mensaje a invalidar y esperar por las respuestas. Si algún cliente no responde a la petición de invalidación, el proveedor debe esperar a que el TTL expire. Obviamente, los mensajes SOAP enviados previamente a los clientes deben tener también un TTL, y éste debe ser menor o igual al TTL utilizado por el proveedor delegado al enviar la notificación de invalidación. De esta manera, el proveedor delegado tiene garantizado que cuando envíe un nuevo mensaje, los clientes han borrado la copia anterior de la *cache* o bien su TTL ha expirado.

13.5. Implementación del *caching* con VWS

Como hemos comentado anteriormente, nuestra propuesta se basa en el uso de VWSEL y de VWSDL para ofrecer mecanismos de mejora basados en *caching*. Podemos, de acuerdo con lo visto hasta ahora, diferenciar 5 patrones de *caching* que pueden ser implementados utilizando ambos lenguajes.

13.5.1. Primera: HoCA de 3 o más niveles

En este tipo de *caching*, entra en juego el VWSDL, permitiendo realizar *caching* en elementos intermedios. No es necesario el uso de VWSEL, si bien puede existir en las invocaciones de cliente delegado a intermediario y de éste a proveedor delegado, ya que, como hemos comentado, una 3LCA es una combinación de dos 2LCA.

Para habilitar *caching* en esta modalidad, lo único que se debe hacer es especificar los elementos `cache` necesarios según los crea conveniente el diseñador del servicio web virtual. El funcionamiento de la *cache* basada en VWSDL ya ha sido explicado en el capitulo 7.

13.5.2. Segunda: 2LCA con TTL

Este tipo de *caching* es prácticamente igual al que se utiliza en el protocolo HTTP para mejorar el rendimiento de la navegación en web. Se basa en el uso de un Time-To-Live que es enviado con los mensajes SOAP de respuesta. En el ejemplo 13.1 podemos ver la cabecera SOAP que envían los clientes que desean utilizar algún sistema de *cache*, en este caso de tipo débil. Obviamente, los clientes delegados deben generar las cabeceras VWSEL, es decir, deben entender dicho lenguaje.

Si un proveedor delegado recibe una petición SOAP con información de *cache*, podrá indicar, en la respuesta, si el mensaje de respuesta se puede almacenar en *cache* o no. Tal como muestra el ejemplo 13.2, la respuesta debe indicar si el mensaje se puede almacenar (`enabled=''true''`) o no, y durante cuanto tiempo (`interval=''600''`).

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache type="weak" />
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.1: Mensaje SOAP de petición con *cache*.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="true" interval="600" />
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.2: Respuesta SOAP con información de *cache*.

13.5.3. Tercera: 2LCA con sondeo

Los sistemas de sondeo presentan algunos inconvenientes como la sobrecarga del sistema de comunicaciones debido a las continuas peticiones de validación que viajan de clientes delegados a proveedores delegados. Sin embargo, no es necesariamente una solución a descartar, ya que puede ser muy útil en entornos en los que la mensajería SOAP de respuesta sufre muy pocas variaciones. En estas circunstancias, un sistema de sondeo con un intervalo de *polling* elevado puede ser la solución más adecuada.

En los ejemplose 13.3 y 13.4 se muestran dos mensajes SOAP (petición y respuesta, respectivamente) con información de *cache* del tipo sondeo.

Una vez que un cliente recibe una respuesta a una invocación, y dicha respuesta contiene información de *cache* habilitando el almacenamiento del mensaje, dicho cliente sólo tendrá que preocuparse de la validez del mensaje cuando venza el intervalo especificado en el propio mensaje. Hasta ese momento, si el cliente delegado realiza una invocación que responda a los mismos parámetros que el mensaje almacenado, la invocación no se realizará, y se devolverá al cliente delegado una copia del mensaje almacenado en la *cache*.

Cualquier mecanismo de *caching* 2LCA debería ser gestionado por los *proxies* y por los *wrappers*, para que todo el proceso sea transparente para las aplicaciones cliente y la lógica de negocio de los servicios web.

```
<SOAP:Header>

  </vwsel:extension>
    <vwsel:cache type="poll" />
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.3: Petición SOAP indicando *caching* de tipo sondeo.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="true" interval="600" >
      <wsa:MessageID>
        uuid:2345A3F5-2005-2WR4-D6HT-34857639023
      </wsa:MessageID>
    </vwsel:cache>
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.4: Respuesta SOAP conteniendo información para el sondeo.

Cuando un *proxy* lo decida, puede solicitar al proveedor delegado información acerca del estado de validez de un mensaje almacenado. Para ello, tal como se muestra en el ejemplo 13.5, el mensaje SOAP de petición debe ir acompañado de un identificador único, que es el que identifica el mensaje. Dicho identificador debe haber sido recibido por el *proxy* en el momento de la primera invocación. El elemento `cache` del mensaje SOAP de respuesta contendrá un elemento `MessageID` (de la especificación WS-Addressing), que identifica de forma única al mensaje de respuesta que se envía al cliente.

Cuando un servicio web recibe una petición de sondeo, lo único que ha de hacer es comprobar si el mensaje sigue siendo válido y responder de acuerdo a la respuesta obtenida. Si la respuesta es positiva (el mensaje es válido) deberá enviar un mensaje de respuesta como el que se muestra en el ejemplo 13.6. En caso contrario, debería enviar un mensaje como el que se muestra en el ejemplo 13.7, junto con el nuevo mensaje de respuesta, que debería sustituir al anterior en la *cache* del *proxy*.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache type="poll">
      <wsa:MessageID>
        uuid:2345A3F5-2005-2WR4-D6HT-34857639023
      </wsa:MessageID>
    </vwsel:cache>
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.5: Mensaje SOAP de comprobación de validez de un mensaje.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="true" interval="600" />
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.6: Respuesta positiva a una comprobación de validez.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="false" />
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.7: Respuesta negativa a una comprobación de validez.

13.5.4. Cuarta: 2LCA con invalidación relajada

La invalidación de tipo relajado supone un nivel mucho mayor de consistencia de los datos alojados en *cache* respecto a los otros sistemas de *cache*

que hemos mencionado hasta ahora. Si bien, no es un sistema consistente al 100 %, su utilización puede ser considerada útil en entornos de conectividad fuerte, como redes locales corporativas. La razón para este tipo de afirmaciones proviene del hecho de que la invalidación que los proveedores delegados envían a los clientes delegados no tiene garantía de entrega, ni de respuesta. Por lo tanto, y ante un eventual problema en la infraestructura de comunicaciones, los datos alojados en las memorias *cache* de los clientes podrían no ser consistentes con los datos ofrecidos por el proveedor delegado.

Cuando un cliente delegado envía un mensaje SOAP de petición de invocación, debe añadir en la cabecera un elemento del tipo `cache` con valor `relaxed`, tal como se muestra en el ejemplo 13.8. Si el proveedor delegado soporta ese tipo de *caching*, contestará, como se muestra en el ejemplo 13.9, indicando si está permitido o no el almacenamiento en *cache*. El mensaje debe, como en casos anteriores, disponer de una identificación única.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache type="relaxed">
      <wsa:From>http://a.client.com/application</wsa:From>
    </vwsel:cache>
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.8: Petición SOAP con solicitud de *cache* de tipo `relaxed`.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="true" >
      <wsa:MessageID>
        uuid:2345A3F5-2005-2WR4-D6HT-34857639023
      </wsa:MessageID>
    </vwsel:cache>
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.9: Respuesta SOAP de *cache* de tipo `relaxed`.

Una vez que un cliente delegado ha recibido un mensaje de respuesta con información de *cache*, ya no necesita preocuparse acerca de la validez del mensaje, como hemos visto en otros mecanismos anteriores. Esta vez será el proveedor delegado el que se encargue de comunicarle al cliente delegado

los cambios que se produzcan en el estado de validez del mensaje.

De esta forma, cuando un proveedor desee invalidar un mensaje enviado previamente, deberá enviar un mensaje como el mostrado en el ejemplo 13.10, donde se indica que la *cache*, para el mensaje identificado con el identificador contenido en `MessageID`, pasa a ser inválida (`enabled=''false''`).

El cliente debe responder al servidor con un mensaje SOAP exactamente igual, es decir, confirmándole que el mensaje en cuestión ha sido borrado de la memoria *cache*.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="false" >
      <wsa:MessageID>
        uuid:2345A3F5-2005-2WR4-D6HT-34857639023
      </wsa:MessageID>
    </vwsel:cache>
  <vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.10: Mensaje SOAP de invalidación.

13.5.5. Quinta: 2LCA con invalidación segura

Por ultimo, el único mecanismo 100 % consistente se basa en la utilización conjunta de dos de los mecanismos que hemos explicado hasta ahora: el TTL y la invalidación. Es lo que nosotros denominamos iTTL, (*invalidation with TTL*). Este proceso se basaría en la siguiente secuencia de mensajes:

1. El cliente delegado envía al proveedor delegado un mensaje como el de el ejemplo 13.11 para solicitar una invocación e indicar la disposición para realizar *caching* seguro.
2. El proveedor delegado le contesta que se permite el *caching* e indica, además, un identificador único del mensaje de respuesta (ejemplo 13.12).
3. Por último, cuando el proveedor delegado desea invalidar un mensaje que ha enviado previamente, debe enviar a los clientes delegados que lo hayan almacenado en su *cache* un mensaje de invalidación como el que se muestra en el ejemplo 13.13.

Es importante tener en cuenta que los dos últimos mecanismos que hemos analizado requieren la concurrencia de dos hechos:

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache type="secure">
      <wsa:From>http://a.client.com/application</wsa:From>
    </vwsel:cache>
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.11: Petición SOAP con solicitud de *cache* segura.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="true" interval="600">
      <wsa:MessageID>
        uuid:2345A3F5-2005-2WR4-D6HT-34857639023
      </wsa:MessageID>
    </vwsel:cache>
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.12: Respuesta SOAP a una petición de *cache* segura.

```
<SOAP:Header>

  <vwsel:extension>
    <vwsel:cache enabled="false">
      <wsa:MessageID>
        uuid:2345A3F5-2005-2WR4-D6HT-34857639023
      </wsa:MessageID>
    </vwsel:cache>
  </vwsel:extension>

</SOAP:Header>
```

Ejemplo 13.13: Mensaje SOAP de invalidación.

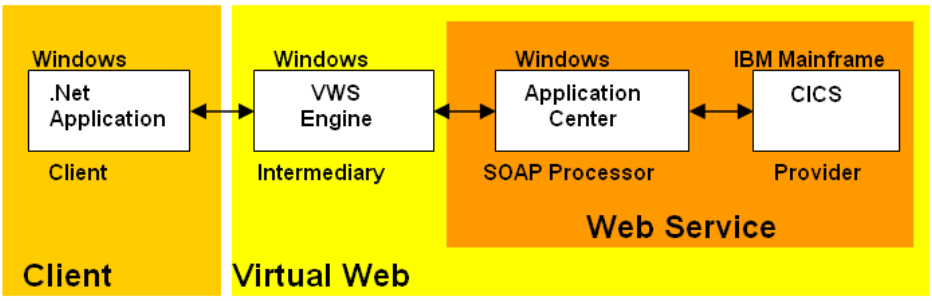

Figura 13.2. Arquitectura del experimento.

- El proveedor delegado puede enviar un mensaje SOAP al cliente delegado, con lo que habrá que tener en cuenta la infraestructura de comunicaciones y de seguridad.

- El cliente delegado debe ser capaz de recibir mensajes de invocación.

13.6. Validación de la propuesta

Hemos construido un simulador, *Rothenburg*, que hemos utilizado para demostrar las funcionalidades de nuestro prototipo de motor VWS a la hora de implementar funciones de *caching*. Más concretamente, hemos realizado diversas pruebas que demuestran la viabilidad de un sistema de optimización basado en *caching*. Para ser exactos, hemos implementado una arquitectura HoCA.

La validación del simulador y del prototipo han consistido en una batería de pruebas sobre una arquitectura como la mostrada en la figura 13.2. El proveedor delegado es un *mainframe* IBM (un modelo 303 de la familia z990, con unos 1300 MIPS de potencia) trabajando conjuntamente con un *Front-End Processor* para la mensajería SOAP, quien se encarga de convertir los mensajes en estructuras planas fácilmente procesables por el *mainframe*. El proceso de las invocaciones de los servicios web en el *mainframe* se realiza dentro de un OLTP (*On-Line Transaction Processor*), el CICS [5] (*Customer Information Control System*).

En la capa intermedia hemos situado nuestro prototipo de motor VWS. La capa cliente la forma *Rothenburg*, que ha sido desarrollado como una aplicación .Net y permite realizar invocaciones de forma masiva. La red LAN que interconecta todos estos equipos es una Ethernet a 100 MBps.

La configuración del software para la prueba era la siguiente:

	S/C	S/C	S/C	C/C	C/C	C/C
	T1	T2	T3	T1	T2	T3
Avg. Resp.(ms)	251.72	260.90	262.66	25.63	38.11	35.71
Total Time(s)	5034.5	5218.2	5253.2	516	766.2	718.9

Cuadro 13.1. Resultados de la prueba.

- Hemos seleccionado una transacción de la aplicación financiera que se ejecuta dentro de la región CICS, en el *mainframe*. Dicha transacción permite consultar el valor de cierre de cualquier día (actual o pasado) de alrededor de 300 fondos de inversión. Esta transacción fue convertida en servicio web utilizando el procesador SOAP antes mencionado.

- La aplicación cliente puede solicitar el valor de cualquier fondo en cualquier fecha. En este caso será nuestro simulador.

- El motor VWS (nuestro prototipo) forma la capa intermedia y ofrece un servicio web virtual que envuelve al servicio real publicado en el *mainframe*.

La batería de pruebas consistió en varias ejecuciones del simulador, cada una de ellas consistente en 20.000 invocaciones sucesivas bajo las siguientes condiciones:

- Test 1: Consultar el mismo fondo en diferentes fechas.

- Test 2: Consultar diferentes fondos en la misma fecha.

- Test 3: Consultar fondos diferentes en fechas diferentes.

Las 3 pruebas fueron ejecutadas dos veces con dos configuraciones diferentes del motor VWS: con *cache* y sin *cache*. Los resultados del test se muestran en el cuadro 13.1. Como podemos apreciar, los tiempos de respuesta observados por los clientes son muy diferentes según utilicemos *cache* o no.

Con los resultados de los tests, incluyendo los tiempos que requirió cada una de las 20.000 invocaciones, hemos creado un conjunto de gráficos que permiten comprobar la eficacia del método. Las figuras 13.3, 13.4 y 13.5 muestran los resultados del los test 1, 2 y 3 sin utilización de la *cache*, respectivamente. Los resultados con la *cache* activada se pueden examinar en las figuras 13.6, 13.7 y 13.8.

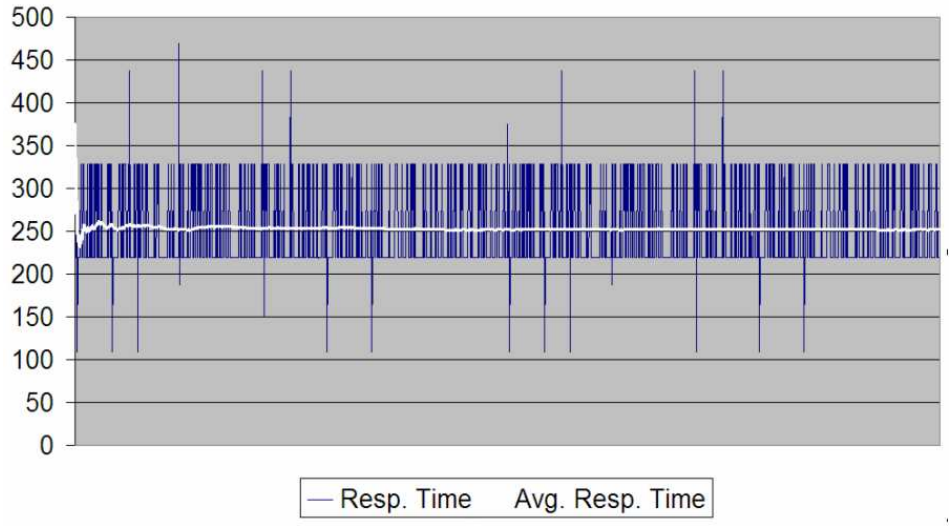

Figura 13.3. Resultados del test 1 sin *cache*.

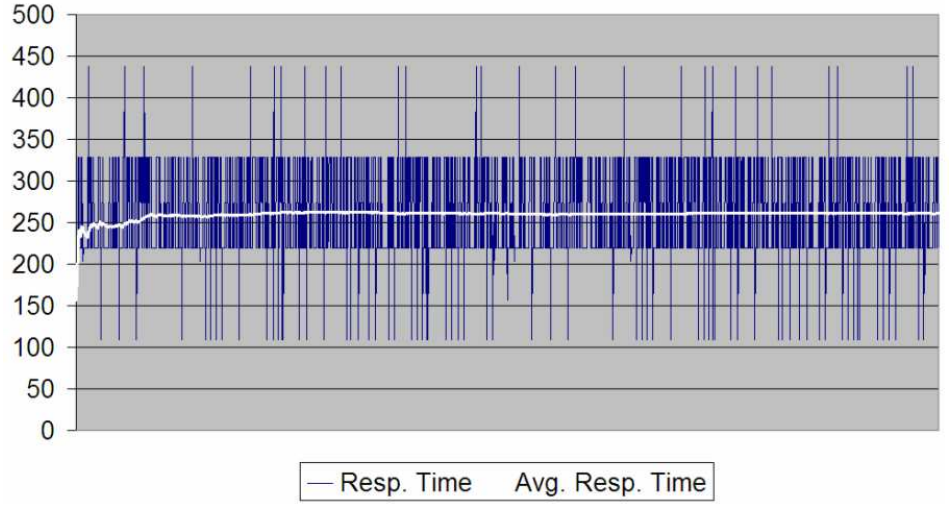

Figura 13.4. Resultados del test 2 sin *cache*.

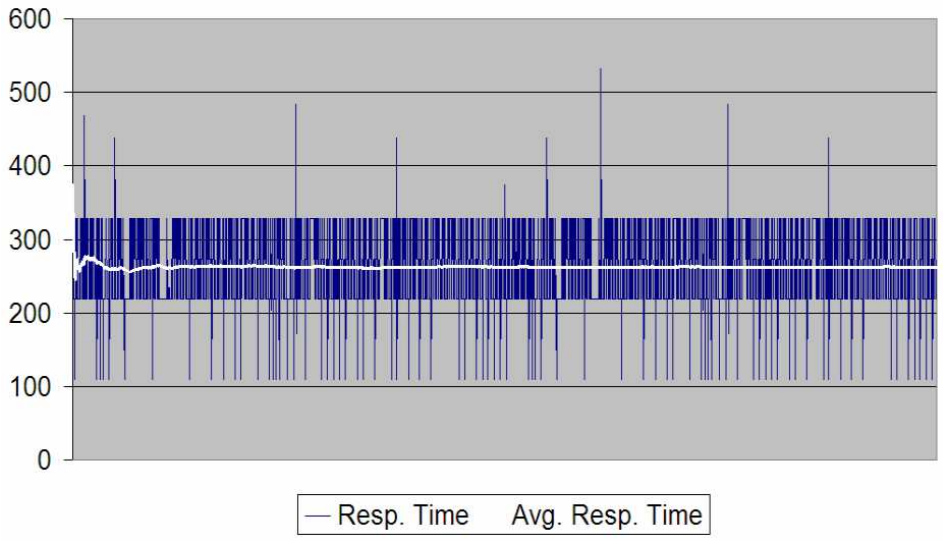

Figura 13.5. Resultados del test 3 sin *cache*.

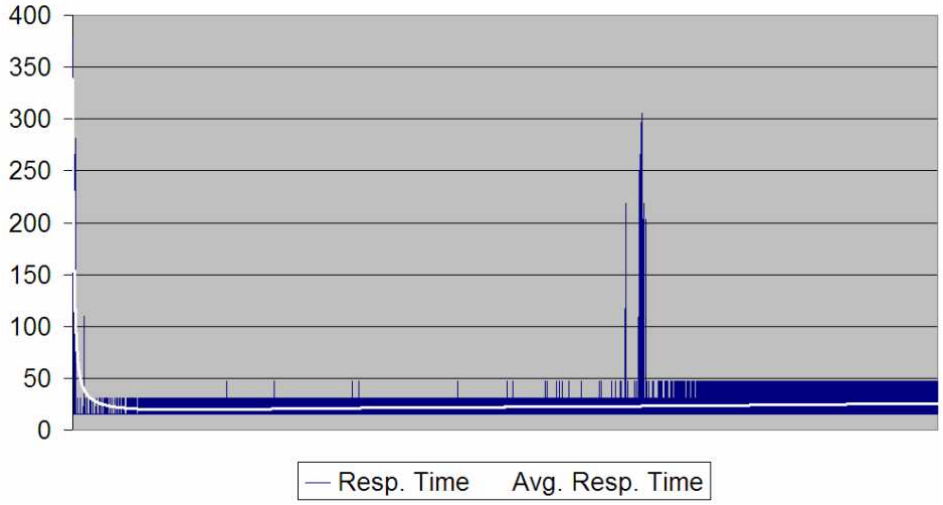

Figura 13.6. Resultados del test 1 con *cache*.

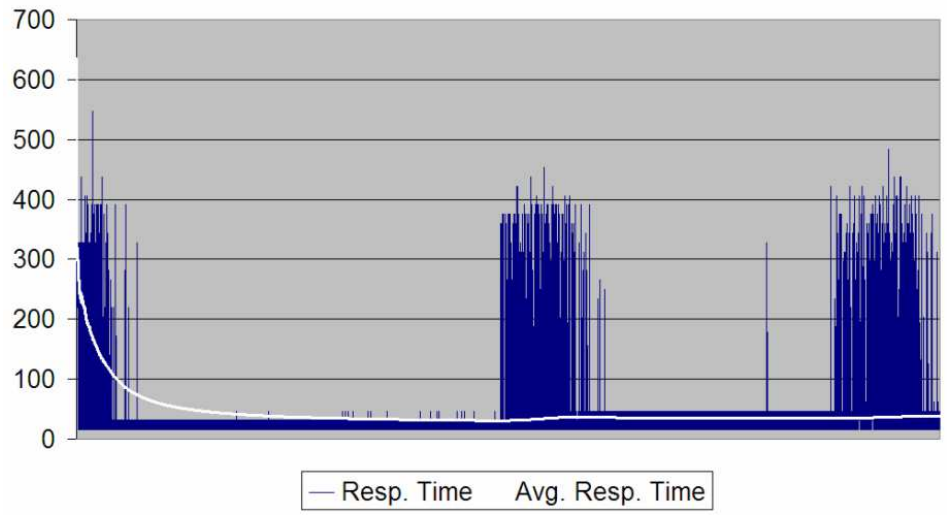

Figura 13.7. Resultados del test 2 con *cache*.

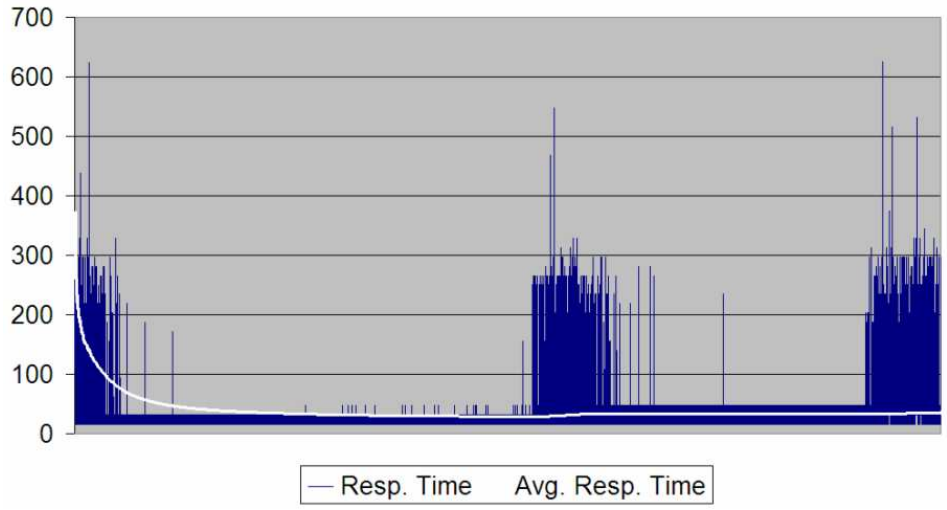

Figura 13.8. Resultados del test 3 con *cache*.

13.7. Conclusiones

Existen propuestas como [111], en donde el *caching* se basa en el uso de cabeceras HTTP, lo cual hace depender la tecnología de servicios web de su capa de transporte. Otras propuestas, como [86], eliminan de las invocaciones recibidas en los servicios web solamente el tiempo de proceso, ya que el mecanismo de *cache* esta implementado en la lógica del servidor. Esto hace que los mecanismos de *cache* empiecen a funcionar cuando se reciba la invocación en el servicio web, y no antes.

Nuestra propuesta ofrece una buena solución a los problemas de optimización de la ejecución, especialmente en entornos con alto volumen de invocaciones. Además, a diferencia de los sistemas basados en *cache* de HTTP, otorga un alto grado de libertad a los desarrolladores a la hora de gestionar las optimizaciones basadas en los sistemas de *caching*, ya que nuestra propuesta, basada en el uso de información adicional que es enviada con los mensajes, permite a los desarrolladores controlar la forma en que los clientes gestionan su *cache*.

Por otra parte, nuestro sistema no se limita al uso de sistemas basados en TTL, como el HTTP, sino que permite la creación de un diálogo (previamente negociado) entre clientes delegados y proveedores delegados. Esta negociación permite reducir las negociaciones al mínimo manteniendo una consistencia fuerte en el *caching* de mensajes.

Es importante tener en cuenta que la especificación contenida en VWSEL no limita la *cache* a un único mecanismo, sino que permite el uso de TTL, sondeo o invalidación, según sea más o menos conveniente en cada entorno de ejecución.

El uso de sistemas *cache* permite reducir el consumo de recursos en los proveedores delegados, mejorar los tiempos de respuesta percibidos por los clientes delegados y reducir la utilización de la red. Es decir, el *caching* no es solamente bueno para los clientes.

Nos gustaría resaltar que el rendimiento de los métodos de consistencia fuerte, como la invalidación segura, es altamente dependiente de la forma en la que se implementa el mecanismo de iTTL. Los valores del TTL tendrán un gran impacto en el rendimiento global del sistema. Un buen punto de partida para optimizar el rendimiento en este sentido sería el TTL Adaptativo [96] utilizado conjuntamente con invalidación, o cualquier otro mecanismo que ayude a predecir la durabilidad de los mensajes que se van a guardar en la *cache* de los clientes.

Aunque no ha sido discutido aquí, los algoritmos para manejar una *cache* de objetos web (LRU, LFU,...) deberían ser revisados y adaptados al nuevo entorno de los servicios web. Un análisis similar debería ser realizado sobre

las políticas de refresco de la *cache*, permitiendo la creación de memorias *cache* activas en intermediarios y/o clientes [65].

Otro punto a analizar en trabajos futuros será la privacidad del los mensajes almacenados en *cache*, especialmente si es una *cache* de intermediario. También nos gustaría señalar que otros esquemas de invalidación podrían ser utilizados, como el canal de invalidación propuesto en [104] pero, posiblemente, incrementando el ancho de banda.

Finalmente, debe quedar claro que todo lo que aquí hemos aplicado a la optimización basada en *cache* dentro de nuestra arquitectura es perfectamente portable a la arquitectura estándar, es decir, utilizando VWSEL, sin necesidad de que existan servicios virtuales o motores virtuales, se pueden implementar técnicas de *caching* en entornos estándar de servicios web.

CAPÍTULO 14

Asincronismo

14.1. Introducción

En la actualidad, la implementación de servicios web asíncronos se hace de forma no estándar. Por una parte, encontramos implementaciones experimentales y comerciales provenientes de distintas entidades. Por otra, estándares consolidados destinados a cubrir otras áreas de actividad de la tecnología de servicios web. Y por otra, un conjunto de iniciativas que por diversos motivos no han llegado a consolidarse.

Dentro del primer grupo encontramos iniciativas como la del *Web Services Invocation Framework* (WSIF [29]), o las implementaciones comerciales de Colaxa (recientemente adquirida por Oracle [17]) o Systinet [24]. Estas tres implementaciones ofrecen la posibilidad de realizar invocaciones asíncronas, pero cada una de ellas utiliza sus propias especificaciones y sus propios mecanismos.

El segundo grupo de soluciones al problema del asincronismo lo forman estándares consolidados como OASIS BTP, con su WS-CAF (*Composite Application Framework*) [113] o el grupo formado por IBM, Microsoft y BEA con su WS-Transaction/WS-Coordination [38]. Ambos son en realidad estándares diseñados originalmente para permitir la implementación de la transaccionalidad dentro de los servicios web, como paso previo a la implementación de las operaciones de negocio. Otros estándares como BPEL4WS, cuyo principal objetivo es facilitar la construcción de servicios complejos de negocio mediante agregación de servicios web, ofrecen también la posibilidad de implementar asincronismo a la hora de invocar servicios construidos siguiendo las especificaciones de BPEL.

En el último grupo encontramos un amplio conjunto de pequeñas especificaciones orientadas a permitir la implementación de asincronismo. Ini-

ciativas como WS-Callback [85], SOAP Conversation Protocol (SCP) [52], Asynchronous Web Services Protocol (AWSP) [131], Asynchronous Service Access Protocol (ASAP) [83] o Simple Workflow Access Protocol (SWAP) [20] han sido utilizadas en distintas experiencias sin llegar a consolidarse.

El primer grupo de iniciativas aporta una solución específica al problema, que ha sido resuelto de manera particular en cada una de las implementaciones mencionadas.

En cuanto al segundo grupo de soluciones, basadas en el uso de estándares, todo parece indicar que suponen la forma más adecuada de implementar asincronismo. El inconveniente que presentan es que no permiten hacer asincronismo de un servicio web cualquiera, es decir, se puede invocar asíncronamente una transacción de negocio implementada con BPEL, o gestionada con BTP, por ejemplo, pero no ofrecen una solución simple al asincronismo de un servicio web básico.

Por el contrario, las soluciones mostradas en el tercer grupo fueron específicamente diseñadas para dar soporte al asincronismo en la invocación de servicios web. Sin embargo, por falta de apoyos institucionales, o por la baja difusión de las soluciones aportadas, los estándares propuestos han dejado de utilizarse salvo, como hemos comentado, en algunas experiencias concretas. Por ejemplo, la especificación SWAP es anterior a la existencia de SOAP. AWSP parece ser la opción más actual y directamente enfocada a la aplicación de asincronismo a los servicios web. Sin embargo, presenta un grave inconveniente, y es que se basa en la utilización de nuevos métodos que permiten a los clientes conocer el estado de las invocaciones realizadas (un mecanismo de *polling*), pero la especificación no soporta los sistemas de *callback*, es decir, que la respuesta sea enviada de forma asíncrona desde el proveedor hacia el cliente.

14.2. Propuestas

A modo de extensión de las funcionalidades de nuestras dos propuestas de especificación para el uso de servicios web virtuales (VWSDL y VWSEL), proponemos la utilización de WS-Addressing para añadir funcionalidades de asincronismo dentro de nuestra arquitectura.

La razón por la que ofrecemos un nuevo mecanismo, que en realidad es igual a los existentes en cuanto a funcionalidad, es que se integra con nuestra propuesta, y sigue, como otros aspectos de nuestra propuesta, los principios de simplicidad y ligereza (en el sentido de facilidad de ejecución). Otra razón para ello, es que los mecanismos del tercer grupo, los más propicios para la implementación de invocaciones asíncronas de una forma sencilla y eficaz, son anteriores a la creación de WS-Addressing, que representa el estado del

arte en el área en que nos movemos.

Así pues, el asincronismo es ofrecido dentro de nuestra propuesta de solución de dos maneras:

- Por una parte, mediante el uso de VWSDL se puede convertir, de forma muy sencilla, un servicio web síncrono en asíncrono.
- Por otra, y para permitir el asincronismo entre entidades sin utilizar componentes intermedios, el VWSEL permite la especificación de invocaciones asíncronas.

14.2.1. Asincronismo de servicios síncronos

Con las soluciones que hemos comentado hasta ahora, para convertir un servicio síncrono en uno asíncrono, la opción más estandarizada consistiría en integrarlo dentro de un proceso de negocio, desarrollado siguiendo las especificaciones de BTP, BPEL, etc., e invocar el servicio web a través de dicho proceso de forma asíncrona.

Esta forma de actuar, si bien resulta efectiva, puede resultar demasiado compleja y costosa, ya que la única funcionalidad que se requiere es el asincronismo, no la transaccionalidad o la construcción de *workflows*.

Para ofrecer una solución sencilla equiparable a los sistemas de asincronismo estándar, nuestro lenguaje VWSDL permite la construcción de servicios web asíncronos a partir de servicios web síncronos. Esto se consigue de forma muy sencilla añadiendo un atributo específico a la definición del servicio. En el ejemplo 14.1 podemos ver como el servicio VWSDL se define como asíncrono gracias a la inclusión del atributo `async=''true''` en el elemento `service`.

```
...
<method name="obtenerDatosCliente"
        type="equivalent"
        default="cos1">
...
<service name="DBAccess"
         group="VWSEngine"
         application="TestHA"
         async="true">
  <event type="connect" iterate="skip" iterations="1" />
  <define name="availability"
          expression="(successful/invocations)*100" />
</service>
```

Ejemplo 14.1: Fragmento de VWSDL con especificación de asincronismo.

Adicionalmente, nuestro lenguaje VWSDL soporta una mayor granularidad en la definición de servicios asíncronos, ya que permite definir **métodos asíncronos**. Es decir, dentro de un servicio web virtual, un conjunto de sus métodos podrá ser invocado de forma asíncrona y otros métodos de forma síncrona.

Para publicar un método asíncrono dentro de un servicio web virtual lo único que ha de hacerse es especificar el atributo `async=''true''` en la definición del método, en el elemento "`method`".

La posibilidad de especificar el asincronismo a dos niveles (método y servicio) abre una nueva discusión acerca del ámbito de los atributos dentro de un documento VWSDL. La precedencia de los atributos de tipo `async` deberá pues ser computada siguiendo las reglas de ámbito que hemos explicado anteriormente. Así, si especificamos distintos valores del atributo `async` en el elemento `service` y en un elemento `method`, el motor VWS que reciba una invocación para el servicio web virtual de este ejemplo deberá atender en primer lugar al valor del atributo `async` del método y, si éste no existe, al valor de dicho atributo que haya sido especificado en el elemento `service`. De esta forma, se plantean tres posibilidades a la hora de definir un método:

- Si un método ha sido definido como asíncrono (`async=''true''`), el valor del atributo `async` del servicio no tendrá efecto alguno sobre éste método, es decir, el método soportará asincronismo.
- Si un método ha sido definido como no asíncrono (`async=''false''`), el valor del atributo `async` del servicio tampoco tendrá efecto alguno sobre éste método, es decir, el método no podrá ser invocado de forma asíncrona.
- Por último, si un método no contiene un elemento `async`, el comportamiento del método ante una invocación asíncrona dependerá del valor del elemento `async` del servicio.

El atributo `async`, aunque pudiera parecerlo, no es en realidad un atributo de tipo booleano, ya que además de poder contener los valores `true` y `false` puede contener un tercer valor: `only`.

Cuando un método especifica en su definición el atributo `async=''only''`, lo que expresa realmente es que dicho método no podrá ser invocado **nunca** de forma síncrona, es decir, que sólo admite invocaciones asíncronas. De esta forma, el valor del atributo `async` permitirá especificar tres alternativas de invocación tal como se muestra en la tabla 14.1.

Desde el punto de vista del constructor de servicios web virtuales, el hecho de especificar al atributo `async` en un método o en un servicio es todo

Valor de `async`	Tipo de invocación
"true"	Síncrona y Asíncrona
"false"	Síncorna
"only"	Asíncrona

Cuadro 14.1. Tipos de invocación admitidos según el valor de `async`.

lo que se necesita para publicar un servicio web (o uno de sus métodos) de forma asíncrona. Pero en realidad, la implementación de un motor VWS será la encargada de hacer que dicha publicación se convierta en algo operativo para las entidades cliente.

En resumen, ya que un servicio web virtual se publica en última instancia como un servicio web estándar, será necesario desarrollar un mecanismo que permita el asincronismo entre entidades cliente y un servicio web estándar. Por este motivo, surge la implementación de asincronismo entre clientes delegados y proveedores delegados sin utilizar intermediarios. Esta implementación del asincronismo es la que se discute a continuación.

14.2.2. Asincronismo entre pares

Al igual que hemos hecho en otras ocasiones, nos basaremos en VWSEL para permitir la especificación de invocaciones asíncronas, ya que, sin duda, es el mejor mecanismo para enviar información de control entre entidad cliente y entidad proveedora.

En nuestra implementación de asincronismo entre pares (sin utilizar intermediarios) existen dos posibilidades básicas, como hemos explicado anteriormente:

- *Polling* o sondeo. Consiste en que el cliente realiza una invocación y, periódicamente, realiza invocaciones para comprobar el estado de la invocación inicial, es decir, realiza un sondeo hasta que el mensaje de respuesta a la primera invocación esté disponible.
- *Callback* o retrollamada. El cliente realiza una invocación y le indica al proveedor la dirección a la que debe enviar la respuesta cuando ésta esté disponible.

Por otra parte, a la hora de realizar una invocación, desde el punto de vista del sincronismo de la respuesta encontramos tres posibilidades:

- Invocación síncrona. Es la invocación tradicional en la que el mensaje de respuesta se devuelve al cliente en la misma conexión que el mensaje de petición.

- Invocación asíncrona con mensaje. Cuando éste está disponible es enviado por el proveedor al cliente inmediatamente.

- Invocación asíncrona con notificación. Cuando el mensaje de respuesta está disponible, el proveedor envía una notificación al cliente y, a continuación, éste deberá invocar al proveedor de nuevo para obtener el mensaje.

En el ejemplo 14.2 podemos ver como se realizaría una petición del primer tipo, síncrona. Este tipo de invocación existe en nuestra especificación por compatibilidad con una invocación estándar, donde se asume una invocación síncrona sin necesidad de especificar ningún tipo de parámetro. El mensaje de respuesta no contendrá ninguna información adicional relativa al sincronismo; contendrá tan solo el mensaje de respuesta a la invocación.

```
<SOAP:Header>
   <vwsel:extension>
     <vwsel:async type="none"/>
   </vwsel:extension>
</SOAP:Header>

<SOAP:Body>
   ...
</SOAP:Body>
```

Ejemplo 14.2: Invocación síncrona con VWSEL.

Si queremos realizar una invocación asíncrona podemos optar, como hemos comentado antes, por una de dos posibles opciones: sondeo o retrollamada.

En el ejemplo 14.3 podemos ver la secuencia de mensajes necesaria para realizar una invocación asíncrona con sondeo. La respuesta a la invocación inicial de la aplicación cliente será necesariamente un identificador de mensaje, que será el que la aplicación cliente utilizará en los sondeos posteriores a esta invocación inicial. En cuanto al cuerpo del mensaje de respuesta, típicamente será nulo, ya que dicho mensaje no estará disponible en el momento de enviar la respuesta.

En el ejemplo 14.4 podemos ver como la respuesta del servidor a una invocación de sondeo es negativa, es decir, el proveedor le indica al cliente, en la respuesta, que el mensaje que solicita todavía no está disponible.

Por el contrario, en el ejemplo 14.5 podemos ver una respuesta positiva a una invocación de sondeo. Una respuesta de este tipo finaliza el ciclo de invocación, es decir, cualquier invocación posterior con el mismo identificador de mensaje deberá dar como resultado una excepción indicando que no

Petición SOAP

```
<SOAP:Header>
   <vwsel:extension>
     <vwsel:async type="poll"/>
   </vwsel:extension>
</SOAP:Header>
<SOAP:Body>
...
</SOAP:Body>
```

Mensaje SOAP de respuesta

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async>
      <wsa:MessageID>
         uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
      </wsa:MessageID>
    </vwsel:async>
  <vwsel:extension>
</SOAP:Header>
<SOAP:Body/>
```

Ejemplo 14.3: Invocación y respuesta asíncrona con *polling*.

Petición SOAP

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async type="poll">
      <wsa:MessageID>
         uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
      </wsa:MessageID>
    </vwsel:async>
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body/>
```

Mensaje SOAP de respuesta

```
<SOAP:Header/>
<SOAP:Body/>
```

Ejemplo 14.4: Petición de *polling* y respuesta negativa.

existe el mensaje. Es importante distinguir la situación en la que el mensaje no existe de aquella en la que el mensaje no está disponible, ya que las aplicaciones cliente deben poder tratar de forma diferente cada tipo de situación.

Petición SOAP

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async type="poll">
      <wsa:MessageID>
         uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
      </wsa:MessageID>
    </vwsel:async>
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body/>
```

Mensaje SOAP de respuesta

```
<SOAP:Header/>
<SOAP:Body>
...
</SOAP:Body>
```

Ejemplo 14.5: Petición de *polling* y respuesta positiva.

Si queremos utilizar un verdadero enfoque asíncrono deberíamos utilizar la retrollamada en lugar del sondeo. Para ello, deberemos utilizar la invocación asíncrona de tipo `callback`, en lugar de la de tipo `poll` que hemos utilizado hasta ahora. Pero además, la invocación con retrollamada nos permite utilizar dos subtipos distintos de invocación asíncrona: la retrollamada con notificación y la retrollamada con mensaje.

En la retrollamada con notificación, el servicio web enviará al cliente que realizó la invocación original un aviso de que el mensaje ya está disponible cuando esto ocurra, pero no le enviará el mensaje propiamente dicho. Ha de ser la aplicación cliente quien se encargue de solicitar el mensaje de respuesta en un momento posterior. Como muestra, en el ejemplo 14.6 podemos ver una invocación de retrollamada de tipo `notify` con su respuesta.

En la mensajería basada en VWSEL se producen a menudo intercambios de mensajes en los que no se maneja información de aplicación, sino solamente información de control. También es posible que como respuesta a una invocación se produzca información de aplicación que ha de ser devuelta en un mensaje que no lleva información de control. Por lo tanto podrán existir mensajes SOAP que porten un `Body` o un `Header` vacíos.

Petición SOAP

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async type="callback" response="notify">
      <wsa:ReplyTo>
         <wsa:Address>http://host.com/client1</wsa:Address>
      </wsa:ReplyTo>
    </vwsel:async>
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body>
...
</SOAP:Body>
```

Mensaje SOAP de respuesta

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async response="notify">
      <wsa:MessageID>
         uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
      </wsa:MessageID>
    </vwsel:async>
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body/>
```

Ejemplo 14.6: Invocación con retrollamada de tipo `notify`.

Petición SOAP

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async type="notify">
      <wsa:MessageID>
         uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
      </wsa:MessageID>
    </vwsel:async>
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body/>
```

Mensaje SOAP de respuesta

```
<SOAP:Header/>
<SOAP:Body/>
```

Ejemplo 14.7: Notificación de mensaje de respuesta disponible.

Cuando una aplicación cliente recibe una invocación de un servidor con un mensaje SOAP que indica el uso de mensajería asíncrona con un identificador de mensaje (como se muestra en el ejemplo 14.7), debe solicitar inmediatamente la recepción del mensaje identificado con dicho `MessageID`. Es decir, la respuesta a una invocación anterior ya está disponible y la aplicación cliente que la originó debe invocar de nuevo al servicio web para obtener (*get*) el mensaje de respuesta. Para ello, debe utilizar un mensaje VWSEL de tipo `get`, que es la forma estandarizada de intercambio de mensajes dentro de nuestra propuesta basada en VWSEL. Un ejemplo de solicitud de mensaje lo podemos ver en el ejemplo 14.8

Dependiendo de la facilidad con que las aplicaciones cliente puedan ser invocadas desde las aplicaciones proveedoras (los servicios web), podremos optar por utilizar el sistema basado en notificaciones que acabamos de explicar, o bien utilizar un enfoque mucho más directo, basado en la entrega asíncrona de los mensajes de respuesta por parte de los servicios web. Este nuevo enfoque consiste en utilizar una invocación asíncrona del tipo `callback` pero de subtipo `message`, es decir, la aplicación cliente le indica al servicio web que desea realizar una invocación asíncrona y que cuando el mensaje de respuesta esté disponible le sea entregado en una dirección determinada. En el ejemplo 14.9 podemos ver una muestra de una invocación de la aplicación cliente de este tipo.

La respuesta inmediata a una invocación del tipo `callback/message` ha de ser necesariamente un mensaje SOAP conteniendo un identificador de mensaje (`MessageID`) que referencia a la petición de invocación que se

Petición SOAP

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:get>
      <wsa:MessageID>
         uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
      </wsa:MessageID>
    </vwsel:get>
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body/>
```

Mensaje SOAP de respuesta

```
<SOAP:Header/>
<SOAP:Body>
...
</SOAP:Body>
```

Ejemplo 14.8: Solicitud de un mensaje.

acaba de recibir en el servicio web. A diferencia de los tipos de invocación que hemos visto hasta ahora, este identificador de mensaje no será enviado nunca por la aplicación cliente, ya que será el servicio web el que realice una invocación hacia el cliente para enviarle el mensaje de respuesta a la invocación inicial. Pero esto no quiere decir que el identificador de mensaje no sea necesario, sino todo lo contrario. Las aplicaciones cliente necesitan los identificadores para poder saber a qué petición enviada con anterioridad corresponde cada mensaje de respuesta recibido, ya que en un momento dado, una aplicación cliente puede haber enviado más de una petición a un servicio web sin haber recibido ninguna respuesta.

Cuando un servicio web está preparado para enviar una respuesta asíncrona a una aplicación cliente, debe realizar una invocación hacia la dirección que le indicó el cliente en la invocación inicial y entregar el mensaje, tal como se muestra en la figura 14.10

14.3. Conclusiones

Como hemos constatado anteriormente, la implementación del asincronismo en nuestra plataforma de servicios web virtuales no aporta nada esencialmente nuevo en el sentido de que ya existen otras formas de realizar invocaciones asíncronas. Sin embargo nuestra propuesta permite:

Petición SOAP

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async type="callback" response="message">
      <wsa:ReplyTo>
         <wsa:Address>http://host.com/client1</wsa:Address>
      </wsa:ReplyTo>
    </vwsel:async>
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body/>
```

Mensaje SOAP de respuesta

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async type="message">
      <wsa:MessageID>
        uuid:6B29FC40-CA47-1067-B31D-00DD010662DA
      </wsa:MessageID>
    </vwsel:async>
  </vwsel:extension>
<SOAP:Header>
<SOAP:Body/>
```

Ejemplo 14.9: Invocación asíncrona del tipo `callback/message`.

Petición SOAP

```
<SOAP:Header>
  <vwsel:extension>
    <vwsel:async type="callback" response="message" />
  </vwsel:extension>
</SOAP:Header>
<SOAP:Body>
...
</SOAP:Body>
```

Mensaje SOAP de respuesta

```
<SOAP:Header/>
<SOAP:Body/>
```

Ejemplo 14.10: Envío asíncrono de un mensaje de respuesta.

- La facilidad de convertir un servicio web asíncrono en uno síncrono, mediante el uso de un componente intermedio que realiza las funciones de "almacén" de mensajería.
- La posibilidad de que el asincronismo se realice a nivel de método (de `portType`) y no de servicio web, es decir, podemos construir servicios web en los que parte de sus métodos sean síncronos, parte sean asíncronos, otra parte pueda ser invocada de forma síncrona o asíncrona indistintamente, y otra parte pueda ser invocada **únicamente** de forma asíncrona.

Por otra parte, nuestra propuesta basada en el uso de VWSEL y VWSDL permite el uso de invocaciones asíncronas a dos niveles. En primer lugar, permite el asincronismo a través de componentes intermedios, que serán los encargados de permitir la conversión de un servicio síncrono en uno asíncrono sin necesidad de modificar nada. Además, VWSEL permite la realización de invocaciones asíncronas directas, de cliente delegado a proveedor delegado, sin necesidad de utilizar elementos intermedios, como ocurre con BTP o con BPEL.

Aunque no ha sido indicado aquí por no considerarse especialmente útil, la utilización de motores VWS como elementos intermedios en una invocación permite invocar de forma síncrona (desde una aplicación cliente) un servicio web asíncrono, gracias la intervención de un motor VWS. Esto no requiere nada especial en el VWSDL que describe al servicio web virtual. La facilidad de convertir un servicio web asíncrono en síncrono descansa sobre la implementación que se haga del motor VWS, no sobre la especificación del lenguaje VWSDL. Lo único que requiere un motor que implemente esta funcionalidad es la capacidad de poder actuar como cliente de una invocación asíncrona, es decir, que la parte cliente del motor sea capaz de mantener diálogos basados en VWSEL.

La utilidad de una funcionalidad como la que acabamos de describir puede ser dudosa, pero no nula, ya que si un cliente requiere un servicio web y sólo encuentra un proveedor que lo ofrezca en modo asíncrono, el mecanismo mencionado permitirá al cliente incorporar dicho servicio a sus aplicaciones cliente de forma síncrona. Otro aspecto a tener en cuenta es el relativo a los tiempos de invocación. Típicamente, los servicios publicados como asíncronos necesitan más tiempo para ejecutarse que aquellos que son publicados como síncronos, por una simple cuestión de diseño del servicio. Esto afectará a los *time-out* de los clientes síncronos, que deberían ser fijados a valores elevados, superiores a los tiempos de ejecución esperados para el servicio web asíncrono al que indirectamente se invoca.

CAPÍTULO 15

Roles del Motor

15.1. Introducción

A lo largo de este trabajo hemos planteado distintas formas en las que puede ser implementada la arquitectura de virtualización atendiendo a la ubicación de los motores VWS. Hemos analizado tres ubicaciones posibles para dichos motores: red pública, red privada del cliente y red privada del proveedor. Cada una de estas ubicaciones nos permitirá obtener distintas funcionalidades de un motor VWS, relacionadas todas ellas, fundamentalmente, con la seguridad.

El hecho de colocar un motor VWS en el extremo de una red privada de clientes delegados, en Internet, o en el extremo de una red privada de proveedores delegados, nos permitirá utilizar el motor VWS como un *proxy*, como un *broker*, o como un *firewall*, respectivamente. Cada uno de estos tres **roles** del motor VWS determina el conjunto de funcionalidades que el motor brindará.

Los distintos roles que puede adoptar un motor VWS no guardan relación alguna con su configuración, ni siquiera con los servicios virtuales que ofrecen. La relación se establece directamente entre ubicación y funcionalidad. Es decir, el mismo motor, con la misma configuración, ofrecerá funcionalidades distintas dependiendo de en cuál de las tres posibles ubicaciones se encuentre.

Un aspecto muy importante a tener en cuenta al implementar los roles que acabamos de ver es la seguridad. El motor VWS puede, independientemente del rol que desempeñe, actuar como elemento asegurador de la privacidad, tanto en el lado del cliente como en el lado del proveedor.

15.2. Propuestas

Analizaremos a continuación los distintos roles que puede interpretar un motor VWS en función de su ubicación: *proxy*, *firewall*, *broker*. También trataremos la construcción de arquitecturas más complejas basadas en la utilización de más de un motor VWS simultáneamente.

15.2.1. *Proxy*

El funcionamiento de un motor VWS como *proxy* no es algo novedoso. El acceso a Internet en las redes corporativas hace años que se realiza a través de *proxy*, debido, fundamentalmente, a las facilidades que ofrece para:

1. **Compartir recursos**, ya que múltiples usuarios con sus navegadores pueden acceder a Internet a través de un única línea de comunicación.

2. **Asegurar la red privada**, ya que no es necesario abrir puertos en los *firewalls* para todos los puestos que tengan acceso a la red pública. Únicamente el servidor *proxy* accederá al exterior.

3. **Simplificar la gestión**, ya que el acceso a Internet se realiza desde un único punto, y no desde cada puesto de la red privada.

4. **Mejorar el rendimiento** mediante el uso de técnicas de *caching*, que son aplicadas en el servidor *proxy*, reduciendo el tiempo de respuesta de todos los navegadores que hagan peticiones a dicho *proxy*.

Razones similares a estas son las que dan origen al rol de *proxy* de los motores VWS, ya que, situados en el borde exterior de una red privada, permiten a las aplicaciones cliente:

1. **Realizar invocaciones de forma segura**, ya que las aplicaciones no necesitan ser accesibles desde la red exterior, pudiendo enviar y recibir mensajería SOAP a través del motor VWS.

2. **Compartir recursos**.

3. **Simplificar la gestión**.

4. **Mejorar el rendimiento**, gracias al uso de VWSDL y VWSEL, las aplicaciones pueden beneficiarse de la mejora de los sistemas *caching* aplicadas a servicios web, como hemos explicado en el capítulo 13.

15.2.2. Cortafuegos (*Firewall*)

Todos los problemas de seguridad relacionados con redes privadas conectadas a redes públicas suelen solventarse utilizando tres técnicas muy comunes hoy en día: los cortafuegos (*firewall*), las zonas desmilitarizadas (*DeMilitarized Zone*, en adelante DMZ) y los *proxies* inversos (*Reverse Proxy*).

En una solución sencilla de aislamiento de red privada, un cortafuegos es una buena solución, ya que éste se encarga de permitir el paso hacia el interior de la red privada solamente a determinados puertos y determinadas direcciones IP. El resto de puertos y direcciones, que no se habilitan en el cortafuegos, serán inaccesibles desde la red pública.

Si se desea obtener un alto grado de protección, la combinación de DMZ y cortafuegos es la solución más habitual. La utilización de DMZ implica la creación de una zona intermedia entre la red privada y la red pública, en la que se instalan aquellas máquinas que deban ser accedidas desde el exterior, pero que redirigen las invocaciones hacia el interior. Por ejemplo, un servidor de aplicaciones se instalaría típicamente en la DMZ, mientras el servidor de bases de datos para dicho servidor de aplicaciones se instalaría en la red privada.

La seguridad mayor, actualmente, se puede lograr instalando un *proxy* inverso en la DMZ, de tal forma que cualquier petición entrante que llegue a la DMZ es redirigida hacia la red privada desde el *proxy* inverso. El *proxy* inverso no realiza ningún tipo de proceso, solamente redirecciona peticiones entrantes, de forma análoga al redireccionamiento de peticiones salientes que realiza un servidor *proxy* de web.

En el caso de los servicios web podemos alcanzar todavía un nivel más de seguridad. Si las técnicas de cortafuegos nos permiten obtener protección a nivel 3 (según el modelo ISO-OSI [37]), y el *proxy* inverso nos permite proteger la red privada a nivel 4 de dicho modelo, la utilización de servicios web virtuales nos permite proteger nuestro sistema a nivel 5 de dicho modelo. ¿Cómo? Utilizando un motor VWS (o más de uno) que actúe como cortafuegos a nivel de invocación SOAP.

La ubicación del motor VWS no es algo trivial. Típicamente, se podría instalar el motor VWS en la DMZ y funcionaría de un modo similar al de un *proxy* inverso: recibe peticiones de la red externa y las reencamina hacia la red privada.

De acuerdo a la arquitectura mostrada en la figura 15.1, las peticiones de invocaciones son recibidas en un motor VWS, que se encuentra situado en la DMZ del proveedor. Dicho motor generará, para cada petición de invocación virtual recibida, una invocación que pasará de la DMZ a la red privada. Esta última invocación será una invocación real, hacia un proveedor delegado.

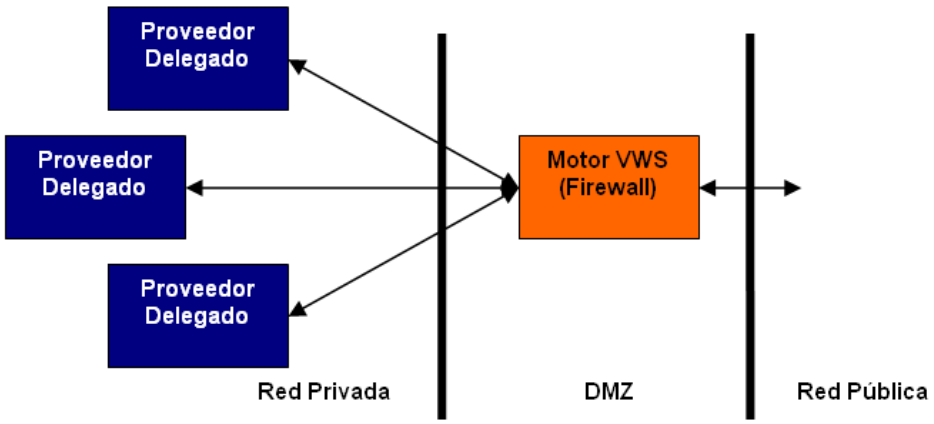

Figura 15.1. Rol de *firewall*.

15.2.3. *Broker*

El concepto de *broker* no es algo nuevo. Cuando hablamos de transacciones entre un cliente y un proveedor (ya sean comerciales o de cualquier otro tipo), es común la utilización de intermediarios, *brokers*, que facilitan la realización de operaciones entre cliente y proveedor.

En una transacción que implique la utilización de un *broker*, existe un intermediario que puede penalizar la transacción de alguna manera, pero que al mismo tiempo, ofrece un valor añadido. De no ser así sería inútil la utilización de intermediarios.

Un valor añadido muy apreciado por los clientes es la facilidad que tienen los *brokers* para tratar con múltiples proveedores. De esta forma, el cliente se beneficia porque puede realizar sus transacciones con el proveedor "más adecuado", es decir, es el más asequible, o el que ofrece el mejor producto, el de mayor calidad, etc.

Adicionalmente, para un proveedor, la realización de transacciones a través de *broker* conlleva la ventaja de extender los canales por los que llegar a nuevos clientes.

En medio de clientes y proveedores, los *brokers* obtienen beneficios por actuar como intermediarios.

Por tanto, la utilización de intermediarios representa a la vez una ventaja para los clientes y para los proveedores. Al mismo tiempo, representa una oportunidad de negocio para terceras entidades independientes.

Si trasladamos estas ideas al mundo de los servicios web, vemos que podemos aplicar perfectamente la tecnología de servicios web virtuales, basados en el uso de intermediarios (motores VWS).

A lo largo de este trabajo hemos mostrado ejemplos en los que la entidad cliente utiliza motores VWS para obtener distintas funcionalidades en su red privada. Asimismo, los proveedores de servicio los utilizan para obtener otro conjunto de facilidades.

Adicionalmente, es posible utilizar un motor VWS como intermediario independiente entre entidades cliente y proveedores de servicio. Se trata de definir una tercera entidad, el *broker*, que permite que las aplicaciones cliente invoquen servicios web de proveedores de servicio a la vez que ambos extremos se benefician de la utilización del *broker*.

El *broker* ofrecerá a las entidades cliente servicios web de valor añadido. Para un cliente, utilizar un servicio web ofrecido por un *broker* representa una gran ventaja, ya que no necesita preocuparse por encontrar los proveedores de servicios más adecuados; esa es la labor del *broker*. Por otra parte, los clientes delegados no necesitan tratar con múltiples interfaces de servicio web, ya que dichos clientes estarán enlazados con un único proveedor: su *broker*.

Al mismo tiempo, ofrecerá a los proveedores la posibilidad de llegar a muchos más clientes. Un eventual acuerdo entre un proveedor de servicio y un *broker*, simplificará su labor de búsqueda de clientes. Además, la existencia de un intermediario permite a los proveedores de servicio modificar los proveedores delegados según sus necesidades, ya que los *brokers* se encargarán de mantener el interfaz inalterado para los clientes.

¿Cómo implementamos un *broker* utilizando servicios web virtuales? Sin duda, el rol de *broker* es el que mejor se adapta a la arquitectura de servicios web virtuales, ya que esta arquitectura ha sido concebida como una estructura basada en el uso de intermediarios.

Típicamente, el rol de "*broker* independiente" debería ser desempeñado por una tercera parte que no guarde relación con las entidades cliente y proveedora más allá de lo contractual. Tal como vemos el la figura 15.2, el *broker* se implementa como un motor VWS independiente, que se situará normalmente en la red pública, para que pueda ser accedido por ambas partes (cliente y proveedor). Cuando un cliente desea realizar una invocación hacia un servicio web virtual, deberá enviar los mensajes SOAP al *broker*, quien se encargará de transformarlos convenientemente y reenviarlos al proveedor que corresponda, según los mecanismos descritos dentro del documento VWSDL que describe el servicio virtual.

Obviamente, como paso previo a la invocación, y en tiempo de desarrollo, la entidad cliente habrá realizado un proceso de *bind* entre sus aplicaciones y el servicio web virtual que le ofrece el *broker*. Y como paso previo a este proceso de *bind*, el *broker* habrá desarrollado los documentos VWSDL y publicado los documentos WSDL correspondientes, necesarios para la pu-

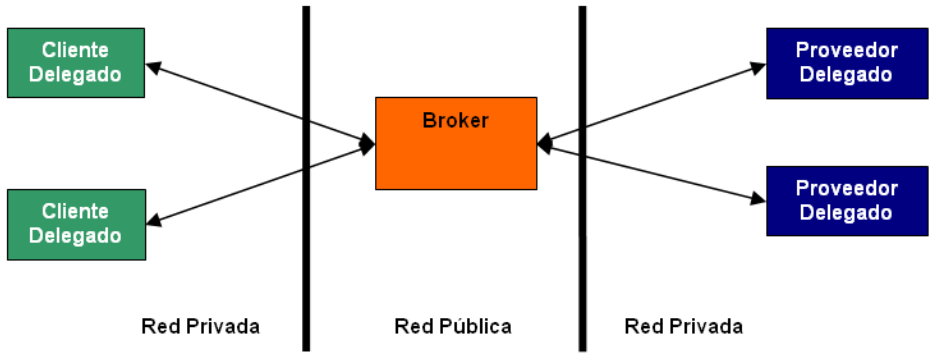

Figura 15.2. Rol de broker.

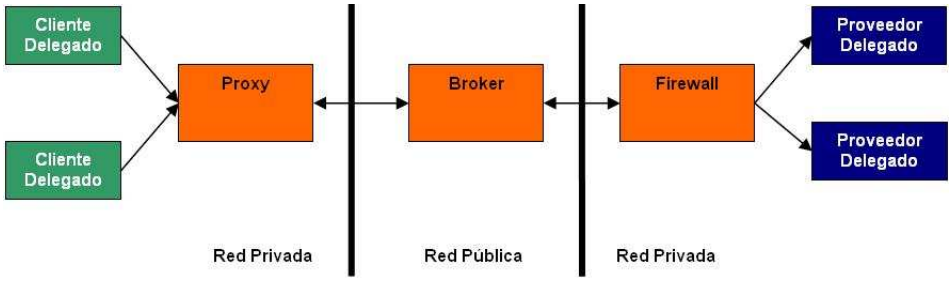

Figura 15.3. Diferentes roles en la arquitectura virtual.

blicación del nuevo servicio virtual.

15.2.4. Arquitecturas complejas

Los tres roles que hemos presentado hasta el momento se basan en una configuración típica de la arquitectura basada en virtualización, en la que intervienen tres elementos básicos: cliente delegado, motor VWS y proveedor delegado. Pero, como ya ha sido apuntado en capítulos anteriores, es posible construir arquitecturas en las que intervengan más de un motor VWS simultáneamente.

Es importante tener en cuenta que en las arquitecturas complejas cada entidad participante es responsable de su propio motor VWS, el cual utiliza para sus propios fines. La aplicación que cada entidad le da a su motor no guarda relación con las aplicaciones que el resto de entidades les den a los suyos. De hecho, la existencia de un motor VWS puede pasar inadvertida para un cliente delegado, ya que los servicios web virtuales son publicados como servicios web estándar.

Así, por ejemplo, en la figura 15.3 podemos ver una arquitectura en la

que participan 5 entidades:

1. En primer lugar encontramos los clientes delegados, que serán los que invoquen a los servicios web.
2. Dentro de la red privada del cliente encontramos también un motor VWS que realiza una función de *proxy*, concentrando todas las peticiones de invocación que se reciben de la red privada y canalizándolas hacia la red pública.
3. Entre clientes y proveedores delegados encontramos un motor VWS que actúa como *broker*, es decir, pone en contacto clientes delegados con proveedores delegados. Típicamente la figura de *broker* es representada por una entidad independiente que aglutina servicios web ofrecidos por distintos proveedores de servicio, y son ofrecidos a clientes con algún valor añadido.
4. En la red privada del proveedor nos encontramos en primer lugar con un motor VWS que actúa esta vez como *firewall*. Se encarga de recibir peticiones de invocación de la red pública, y redirigirlos a proveedores delegados instalados en la red privada. De esta forma, éstos últimos no necesitan estar accesibles desde la red pública, con la consiguiente mejora de seguridad.
5. En último lugar, nos encontramos con los proveedores delegados que ejecutan en última instancia los servicios solicitados por los clientes delegados.

Debe quedar claro que una arquitectura compleja articulada de esta forma no se utiliza simplemente como un mecanismo de redireccionamiento de mensajería SOAP. Todo lo contrario. Cada punto intermedio de la arquitectura es un punto con inteligencia en el proceso de mensajes, que puede aportar por tanto un valor añadido al proceso de la mensajería. Como hemos ido viendo a lo largo de este trabajo, los nodos intermedios pueden realizar labores de gestión de la calidad de servicio, disponibilidad, mejoras de rendimiento, etc.

Por ejemplo, el *broker*, lo compone un motor que trabaja con distintos proveedores y tendrá, por tanto, acuerdos firmados con ellos. Pero puede ofrecer a sus clientes un único servicio web y valores añadidos muy importantes, como la gestión de proveedores, o la alta disponibilidad.

15.3. Conclusiones

El mensaje más claro que extraemos del análisis de roles de los motores VWS es su simplicidad. El mismo motor VWS puede ser utilizado en cada

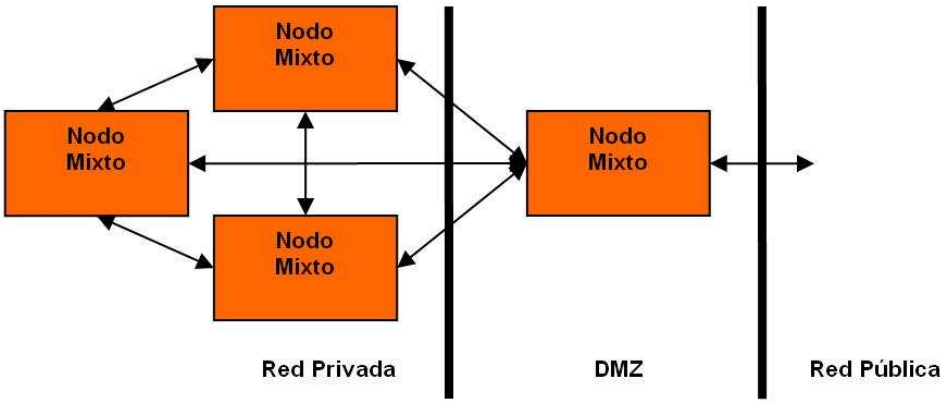

Figura 15.4. *Firewall* y *cluster* de servicios web.

uno de los roles que hemos analizado sin cambio alguno en su configuración.

Al mismo tiempo, las soluciones basadas en el uso de motores nos ofrecen un alto grado de homogeneidad entre sus componentes, ya que el mismo tipo de elemento puede ser utilizado para distintas funciones. Por ejemplo, podemos construir una entidad proveedora con un *cluster* de servicios web formado por nodos mixtos (son a la vez nodos controladores y nodos de servicios), y añadirle un firewall para proteger el *cluster* del exterior de la red privada del proveedor. Todos estos componentes (nodos mixtos y firewall) se implementan de la misma forma, utilizando motores VWS que publican los servicios que el proveedor de servicio desea ofrecer a sus clientes.

En realidad, si construimos un *cluster* de servicios web utilizando nodos mixtos (todos ellos con la misma implementación y la misma configuración), y situamos uno de los nodos en la DMZ de la entidad proveedora, habremos conseguido el mismo efecto: un firewall de servicios web más un *cluster* de servicios web (figura 15.4)

CAPÍTULO 16

Detección y Gestión de errores

16.1. Introducción

Disponer de un punto intermedio entre clientes delegados y proveedores delegados nos permite realizar ciertas labores de control sobre las invocaciones en curso. Un motor VWS puede ser utilizado para seleccionar el proveedor delegado y el proveedor de servicio más adecuados para cada invocación. Pero esta labor de control podría llegar aun más lejos. ¿Qué ocurre en circunstancias normales cuando una invocación falla? Sencillamente, se devuelve al cliente un mensaje SOAP conteniendo un `faultcode` y un `faultstring` indicando la causa del fallo.

A la hora de desarrollar software, el tratamiento de errores es siempre una labor difícil. Cuando además dentro de las aplicaciones realizamos invocaciones hacia servicios alojados en otras máquinas (un servicio web, por ejemplo), el tratamiento de errores debe ser todavía más estricto, puesto que las posibilidades de fallo aumentan debido al uso de componentes externos, como los proveedores de servicios web, la red de comunicaciones, etc.

Ahora, gracias a la introducción de los motores VWS, podremos reintentar la invocación antes de devolver un error al cliente. Un motor VWS puede gestionar los errores y, bajo ciertas condiciones, reintentar la invocación seleccionando un nuevo proveedor delegado (del mismo o distinto proveedor de servicio) después de haber detectado una invocación fallida. Esta forma de trabajar permite construir aplicaciones cliente más robustas, a la vez que representa una mejora en la disponibilidad desde el punto de vista del cliente delegado.

Cuando desarrollamos aplicaciones cliente que utilizan un servicio web, el tratamiento de errores debe ser realizado igual que se haría con cualquier otra invocación, es decir, se debe tratar dicha invocación como si fuera una

llamada a una subrutina, a una clase, a un componente, etc.

Supongamos la existencia de una aplicación cliente que utiliza un servicio web para consultar el valor de cambio de una divisa. Como hemos dicho, al desarrollar la aplicación debemos realizar un tratamiento de errores similar al que realizaríamos con cualquier otro tipo de invocación.

El servicio web que invocamos desde la aplicación, por ser un servicio de consulta, puede ser realizado con prestaciones equivalentes por otros muchos proveedores. Así que si queremos mejorar la disponibilidad de la aplicación, lo único que hemos de hacer es agregar más proveedores.

La agregación de proveedores exige una tarea de desarrollo adicional, ya que los procesos de *bind* deben ser realizados para cada proveedor. Adicionalmente, debemos realizar tratamiento de errores para cada uno de ellos y, para mejorar la disponibilidad de la aplicación cliente desde el punto de vista del usuario, deberemos implementar algoritmos que se encarguen de invocar proveedores, tratar los eventuales errores que puedan producirse y, en caso necesario, reintentar las invocaciones con proveedores alternativos.

Todo este tratamiento de errores, y los algoritmos de selección deben ser desarrollados *ad-hoc*, es decir, para cada nuevo proveedor que se añada se debe adaptar los algoritmos de selección para que lo tengan en cuenta.

16.2. Propuestas

Nuestra arquitectura basada en una capa de virtualización intermedia permite construir servicios web virtuales cuya ejecución se basa en la invocación de servicios web reales, ofrecidos por proveedores delegados.

Cada método de cada servicio virtual se construye mediante puertos ofrecidos por servicios web reales. Esto nos brinda la oportunidad de simplificarle a las aplicaciones cliente (los clientes delegados) ciertos trabajos relacionados con el control de errores. La utilización de servicios virtuales nos aporta, desde este punto de vista, las siguientes ventajas:

- Realizar el tratamiento de errores dentro del propio motor VWS.
- Reintentar la invocación con proveedores alternativos siempre que esto sea posible.

La capa de virtualización aporta inteligencia en todos esos procesos, es decir, no se limita a ser un mero retransmisor de peticiones de invocación, sino que puede tomar decisiones y actuar en consecuencia.

La forma en la que se implementa el tratamiento de errores es, de esta manera, mucho más sencilla para las aplicaciones cliente, ya que sólo nece-

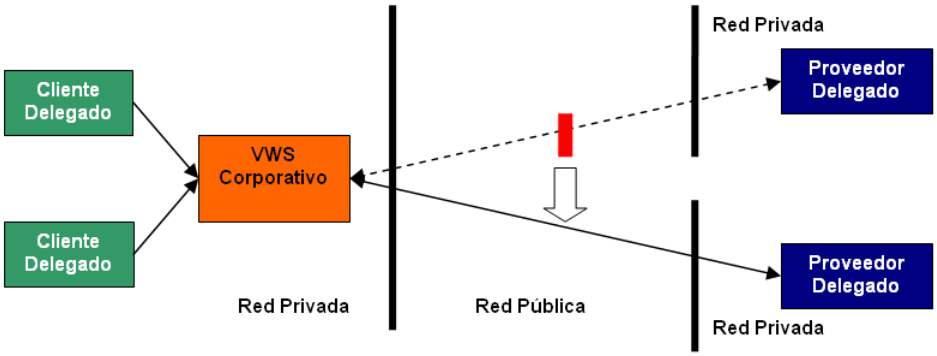

Figura 16.1. Tratamiento de errores con VWS corporativo.

sitan realizar los procesos de *bind* una única vez con el servicio virtual. Deja de ser necesaria la implementación de algoritmos de selección, o tratamientos de errores para múltiples proveedores, ya que toda está lógica estará implementada en los motores VWS.

Al mismo tiempo, cuando utilizamos motores VWS corporativos estamos obteniendo beneficios adicionales en relación al tratamiento de errores. Supongamos la existencia de un método virtual implementado con varios servicios web (figura 16.1). Si uno de los proveedores de dichos servicios deja de funcionar, el motor VWS dejará de seleccionarlo para la ejecución del servicio virtual, como ya hemos comentado en capítulos anteriores. Pero al mismo tiempo, el hecho de utilizar un servicio virtual evitará que se cometan más errores de los necesarios, ya que todas las aplicaciones cliente que invoquen el mismo servicio virtual se beneficiarán del conocimiento que el motor VWS tiene acerca de los proveedores. De esta forma, una vez se haya detectado un error en un proveedor delegado (a raíz de la invocación de un cliente delegado), las invocaciones subsiguientes realizadas por otros clientes delegados no sufrirán las consecuencias de trabajar con un proveedor erróneo, ya que el motor realizará las invocaciones seleccionando proveedores alternativos.

Por otra parte, el uso de motores VWS corporativos disminuye las posibilidades de error en las aplicaciones cliente, ya que éstas sólo realizarán invocaciones a servicios web locales, instalados dentro de la red corporativa, lo que evita problemas de seguridad (todo se ejecuta en un entorno controlado) a la vez que disminuye las posibilidades de error causadas por el uso de redes de comunicación externas (Internet) directamente desde las aplicaciones.

16.3. Validación de la propuesta

Para probar las funcionalidades de tratamiento de errores que proponemos, hemos utilizado nuestro simulador *Naoussa*, que permite invocar un servicio virtual desde varios clientes delegados.

Hemos realizado básicamente dos tipos de pruebas, encaminadas a demostrar la continuidad del sistema mediante la detección de errores:

1. Tiempos de respuesta elevados. Utilizando la parte cliente de *Naoussa* hemos iniciado un proceso de invocación continuada. Durante dicho proceso hemos utilizado el simulador de problemas de *Naoussa* para introducir retardos en las respuestas de uno de los proveedores delegados. Automáticamente, y de forma natural, el sistema empezó a utilizar el otro proveedor.

2. Caída de un proveedor. Para simular la caída de un proveedor delegado hemos puesto el tiempo de respuesta a infinito. Para ello, hemos utilizado el simulador de problemas para elevar el tiempo de respuesta a un número excesivamente elevado. De forma automática, el motor VWS al que invoca *Naoussa* ha dejado de seleccionar dicho proveedor para pasar a utilizar los otros disponibles.

Los resultados de estas pruebas pueden ser examinados en el apartado 11.5 del capítulo 11.

Utilizar nuestra propuesta para sistemas de alta disponibilidad y para tratamiento de errores a la vez es una aproximación perfectamente válida, ya que nuestra tecnología ofrece formas prácticamente iguales de resolver distintos problemas como los que acabamos de mencionar, basándose un servicios virtuales y motores VWS.

16.4. Conclusiones

Como hemos visto, el uso de servicios virtuales ofrece importantes ventajas en lo que a la detección y manejo de errores se refiere:

- Simplifica los procesos de desarrollo de aplicaciones, ya que gran parte de la lógica de tratamiento de errores se puede realizar en los motores VWS.

- Mejora la disponibilidad, ya que permite disponer de proveedores alternativos que cumplan la misma función. Y todo ello sin necesidad de alterar las aplicaciones cliente.

CAPÍTULO 17

Gestión de Proveedores

17.1. Introducción

A medida que la tecnología de servicios web se extienda y las empresas ofrezcan a sus clientes la posibilidad de interactuar directamente (B2B, B2C) y no a través de aplicaciones, nos encontraremos cada vez más con la necesidad de incorporar distintos proveedores a la misma aplicación. Ello se traduce en la necesidad de enlazar un mismo cliente delegado con varios proveedores delegados. Esta relación "1 a n" entre cliente delegado y proveedor delegado requerirá la existencia de una gestión sencilla de los proveedores delegados utilizados por cada cliente delegado. Esta gestión se traduce en la necesidad de agregar nuevos proveedores, eliminar proveedores en desuso, o adaptar los clientes delegados a nuevas versiones de los proveedores delegados que impliquen modificaciones en los interfaces.

Como hemos comentado en capítulos anteriores, en la arquitectura actual de los servicios web, es necesario desarrollar código específico para tratar cada proveedor delegado individualmente.

La utilización de servicios web virtuales, permite mejorar enormemente la gestión de los proveedores delegados, ya que utilizando VWS podemos enlazar una aplicación con un único proveedor delegado (que será el servicio virtual), a la vez que éste se encuentra enlazado (mediante un documento VWSDL) con un conjunto de proveedores delegados (servicios web reales) que son equivalentes en cuanto a su funcionalidad.

El enlace directo (*direct binding*) es la forma adoptada hoy en día para enlazar proveedores delegados con clientes delegados. Dicha forma de enlace se traduce en la generación de *proxies* en tiempo de desarrollo de las aplicaciones, no en tiempo de ejecución. De esta forma de enlazar las aplicaciones surgen las dificultades para realizar modificaciones a la lista de proveedo-

res delegados en uso. Como hemos comentado anteriormente, además de tener que realizar los procesos de *bind*, es necesario adaptar la lógica de los clientes delegados para que tengan en cuenta los nuevos *proxies*, los *proxies* modificados, o que dejen de utilizar *proxies* eliminados.

Parte de estos problemas se debe a la utilización del enlace directo entre clientes delegados y proveedores delegados, es decir, al ya mencionado problema del acoplamiento entre clientes delegados y proveedores delegados desde el punto de vista del interfaz.

17.2. Propuestas

Al igual que en otros problemas que hemos analizado anteriormente, la gestión de proveedores puede ser aliviada enormemente introduciendo un componente intermedio que nos permita la construcción de enlaces indirectos, es decir, que los clientes delegados no se enlacen directamente con los proveedores delegados, sino con algún elemento intermedio. Dicho elemento será, de acuerdo a nuestra propuesta, un servicio web virtual.

Como hemos explicado anteriormente, los servicios web virtuales encapsulan a los servicios web reales, lo que permite dividir el *binding* en dos partes:

- Un proceso de enlace entre aplicaciones cliente y los servicios web virtuales. Este proceso es exactamente igual al que se realizaría entre aplicaciones cliente y proveedores de servicios web. La única diferencia es que se hace entre una aplicación cliente y un servicio web que se publica a partir de un servicio web virtual, y que ambos residen en un mismo motor VWS.

- Otro proceso de *binding* entre motores VWS y proveedores delegados. Este proceso de enlace, que no existe cuando desarrollamos aplicaciones con enlace directo, no es en realidad un enlace tal como lo entendemos habitualmente, ya que la construcción de los mensajes que se han de enviar a los proveedores delegados se hace utilizando transformaciones sobre los mensajes SOAP recibidos de los clientes delegados. Para realizar las transformaciones no es necesario el uso de *proxies*, con lo que podríamos decir que el enlace entre motores VWS y proveedores delegados es dinámico. Hay que tener en cuenta que el motor VWS maneja metadatos para realizar estas transformaciones, con lo que aplicar una solución como la que proponemos (basada en transformaciones), al enlace directo en aplicaciones cliente, complicaría enormemente el desarrollo de las aplicaciones cliente.

Como vemos, convertimos un *binding* directo en dos enlaces, uno directo a un motor VWS y uno indirecto desde dicho motor a los proveedores delegados.

Tal como hemos estudiado en capítulos anteriores, la definición de un método virtual se realiza, incluyendo elementos `invoke` dentro de dicha definición además de otras cosas. Cada uno de estos elementos representa en realidad la definición de un enlace dinámico entre un motor VWS y un proveedor delegado. Así pues, la gestión de proveedores se reduce básicamente a:

- Definir los elementos `invoke` que necesitemos, de acuerdo a los datos que tengamos de los WSDL correspondientes (los que implementan el servicio).
- Definir las transformaciones necesarias para cada proveedor delegado que hayamos especificado dentro de cada `invoke`.

Ambos pasos se realizan sobre un documento VWSDL, es decir, sobre una definición de un servicio web virtual, con lo que no está involucrado en este proceso ninguna modificación de aplicaciones: **son sólo definiciones**.

Por lo tanto, los procesos de adición, modificación o eliminación de proveedores delegados se realizarán, respectivamente, como sigue:

1. Añadir un elemento `invoke` (y la transformación que corresponda, si es necesaria) a la definición del método virtual para agregar un proveedor.
2. Modificar la definición del elemento `invoke` y/o la transformación asociada para modificar el enlace con un proveedor.
3. Borrar un elemento `invoke` de un documento VWSDL para dejar de utilizar un proveedor.

Es importante tener en cuenta dos cuestiones importantes:

1. Sigue habiendo un enlace directo entre las aplicaciones cliente y el motor VWS. La aplicabilidad de esta solución se basa en el hecho de que las entidades cliente utilicen motores VWS para concentrar las invocaciones desde la red corporativa hacia el exterior, es decir, múltiples clientes corporativos se beneficiarán de disponer de un único interfaz para distintos proveedores, quienes, presumiblemente, utilizan distintos interfaces.

2. Como las modificaciones a las listas de proveedores que implementan los servicios se hacen en las definiciones de los servicios web virtuales, y no hay que modificar ninguna aplicación, es perfectamente válida la modificación de servicios web virtuales (desde esta óptica de multiproveedor) en tiempo real. Es decir, las aplicaciones pueden estar ejecutándose, realizando invocaciones, al mismo tiempo que se modifica el documento VWSDL. Esto se debe a que el enlace entre motores y proveedores delegados es completamente dinámico (de hecho el enlace se crea, como hemos comentado, en tiempo de ejecución). Esto, finalmente, representa también una mejora en la disponibilidad, ya que permite reducir las paradas planificadas debidas a cambios en la configuración de las aplicaciones.

17.3. Validación de la propuesta

Para verificar la validez de nuestra propuesta desde el punto de vista de la gestión simplificada de proveedores, hemos utilizado nuestro simulador de carga *Naoussa*, y lo hemos ejecutado lanzando, desde 3 aplicaciones cliente (3 *threads* de *Naoussa*), peticiones de invocación contra dos proveedores por tiempo indefinido.

Durante la prueba hemos modificado el documento VWSDL que define el servicio virtual utilizado por los clientes simulados realizando los tres procesos mencionados previamente (agregación, modificación y eliminación).

El resultado ha sido completamente exitoso. Tal como hemos comentado, no hemos necesitado modificar las aplicaciones cliente. Además, el cambio lo hemos hecho sin hacer ninguna parada en *Naoussa*, es decir, los tres clientes simulados estuvieron enviando peticiones de invocación ininterrumpidamente durante todo el período de duración de la prueba.

17.4. Conclusiones

Desde nuestro punto de vista, la introducción del motor VWS entre clientes delegados y proveedores delegados representa una forma sencilla de garantizar la independencia entre ambos extremos. Aunque sí es cierto que agrega un componente intermedio, los beneficios desde el punto de vista del desarrollo de aplicaciones (e incluso desde la explotación, como hemos visto) son considerables. Evidentemente, la introducción de la capa de virtualización es algo completamente opcional, las aplicaciones cliente pueden decidir cuando invocar a un servicio web virtual (que para ellas se muestra como un servicio estándar más) o a uno tradicional.

Si no se esperan grandes cambios en los interfaces de los proveedores utilizados, o bien, si no se espera cambiar de proveedor frecuentemente (o agregar otro, o dar de baja alguno), puede no compensar la utilización de la capa virtual. Es el arquitecto de software quien debe decidir en qué ocasiones es conveniente y en cuáles no lo es.

En cualquier caso, su utilización siempre será beneficiosa, ya que permite a los desarrolladores de aplicaciones cliente anticipar los cambios que se produzcan en el mercado, puesto que disponen de un mecanismo sencillo para elegir los proveedores más adecuados en cada momento, ya sea por criterios de coste, de rendimiento, o de cualquier otro tipo. Esta facilidad para anticiparse, se traduce en tres ventajas importantes:

- Disminuye el *Time-To-Market*, es decir, acorta el tiempo que las empresas necesitan para adaptar su infraestructura tecnológica (aplicaciones que invocan a servicios web, en nuestro caso) a los cambios que se produzcan en el mercado.
- Permite aprovechar las oportunidades de negocio, ya que las empresas son más ágiles a la hora de desarrollar software.
- Simplifica los procesos de mantenimiento de aplicaciones, ya que no será necesario modificar las aplicaciones cliente si se producen cambios en el proveedor (interfaz o implementación), o si se decide cambiar de proveedor, o agregar nuevos proveedores.

CAPÍTULO 18

Reducción del Nivel de Acoplamiento

18.1. Introducción

La tecnología de servicios web está cautivando la atención de los desarrolladores. Las tradicionales aplicaciones distribuidas basadas en DCOM o en EJB están dando paso, poco a poco, a aplicaciones diseñadas bajo las ideas de las arquitecturas SOA. Como parte de estas ideas, la utilización de servicios web es un punto caliente en el desarrollo de aplicaciones. Sin duda, con el paso del tiempo, cada vez encontraremos más aplicaciones basadas en servicios web y menos basadas en protocolos o arquitecturas propietarias.

Todo esto acabará por traducirse en un mundo de aplicaciones conectadas, ya sea dentro del entorno corporativo o a través de Internet. Esta tendencia a desarrollar software basado en servicios web acabará por traducirse en una gran ventaja para los desarrolladores de aplicaciones: más proveedores para un mismo servicio, con las ventajas derivadas de la competencia (abaratamiento de precios, mejora de la calidad, etc.).

Pero no todo son ventajas. En el lado negativo, los desarrolladores de aplicaciones cliente se encontrarán con que para un mismo servicio web podrán elegir entre un conjunto de proveedores cuyos **interfaces serán distintos**, independientemente de cual sea su implementación. Esto limitará las posibilidades de las aplicaciones, puesto que se verán obligadas a gestionar servicios web que manejarán distintos tipos de datos para las mismas funciones, o distinto número de parámetros, o distintas estructuras de los mismos, etc.

De manera similar a la utilizada en la gestión de múltiples proveedores (capítulo 17), podemos enfrentarnos muy fácilmente a los problemas de acoplamiento que se producen entre clientes delegados y proveedores delegados. Aunque una de las mayores virtudes de los servicios web es su bajo acopla-

miento, este no es nulo. Y existen situaciones, que serán más frecuentes cuanto más se extienda la tecnología de servicios web, en las que un cambio en la estructura de los datos que envía o recibe un proveedor delegado (el interfaz del servicio web), implique la modificación de los clientes delegados para adaptarlos al nuevo interfaz.

La utilización de motores VWS mejora la disminución del nivel de acoplamiento, ya que permite que los clientes delegados dialoguen con un interfaz (el que les presenta el servicio web virtual), mientras los proveedores delegados pueden dialogar con el motor VWS utilizando un interfaz completamente distinto. Es el motor VWS el responsable de hacer compatibles dichos interfaces (esquemas XSD en realidad), gracias al uso de las especificaciones contenidas en los documentos VWSDL.

Además, una vez que se ha desarrollado una aplicación cliente que ha sido enlazada con un conjunto de servicios web, la adopción de un nuevo proveedor es un proceso traumático, ya que exige la realización de, al menos, las siguientes tareas:

- Realizar el proceso de *binding* para el nuevo servicio.
- Añadir lógica de tratamiento de errores y combinarla con la ya existente para los servicios web en uso.
- Modificar las aplicaciones cliente para tratar las nuevas estructuras de datos que espera recibir y devolver el nuevo servicio web.
- Despliegue de las aplicaciones modificadas.

La simplificación de los procesos de *binding* y tratamiento de errores han sido comentados en el capítulo 16. Por otra parte, cuando queremos tener posibilidad de invocar múltiples proveedores desde una misma aplicación cliente debemos también prestar especial atención a las estructuras de los datos.

Los espacios de nombres, los tipos de datos y los nombres de los campos de las estructuras son tres factores a considerar a la hora de integrar servicios web creados por distintas entidades, y que se relacionan directamente con los procesos de enlace. Estos problemas son causados en realidad por el acoplamiento existente entre las aplicaciones cliente y el interfaz de los servicios web que utilizan. Aunque es cierto que la utilización de servicios web disminuye enormemente el acoplamiento entre cliente y proveedor, en realidad ésto sólo es aplicable a la relación existente entre aplicación cliente e implementación del servicio web, ya que el acoplamiento entre dicha aplicación y el interfaz es tan fuerte como el que podemos encontrar entre una clase llamada desde otra, o una función (con sus parámetros) llamada desde alguna otra parte de un programa.

Aunque es posible realizar invocaciones utilizando sistemas de enlace dinámico ([29] o las *Endpoint Referentes* del BPEL [69]), dichos sistemas se basan en el uso de metadatos, que deben ser manejados por las aplicaciones y que, además de exigir una programación especial, impiden el uso natural de los servicios web, que es a través de *proxies* que se traducen en objetos nativos del lenguaje de programación utilizado para desarrollar las aplicaciones cliente.

18.2. Propuestas

Para solucionar los problemas derivados del elevado acoplamiento entre aplicaciones e interfaz de servicios web proponemos el uso de nuestra arquitectura basada en virtualización. El verdadero valor de nuestra propuesta, para los problemas de acoplamiento, reside en la introducción de una capa intermedia, que es la que permite dividir el problema de acoplamiento en dos partes:

- Acoplamiento entre aplicaciones cliente y la capa de virtualización.
- Acoplamiento entre la capa de virtualización y los proveedores de servicios web.

La introducción de la capa de virtualización permite aislar los interfaces de los proveedores, desde el punto de vista de las aplicaciones cliente. La capa virtual, que debe ser implementada utilizando elementos activos e inteligentes, será la responsable de dialogar con el interfaz que los proveedores presentan.

Al mismo tiempo, la capa virtual debe presentar a las aplicaciones un nuevo interfaz. Evidentemente, existe acoplamiento entre aplicación cliente e interfaz, pero es completamente distinto del que existiría entre aplicaciones cliente y proveedores de servicio web, ya que la introducción de un elemento activo en la arquitectura permitirá realizar modificaciones a los interfaces en tiempo de ejecución, evitando de esta forma los procesos de enlace y otros procesos derivados de éste.

Por tanto, la utilización de la capa virtual permite que los clientes delegados realicen invocaciones hacia los motores VWS (elementos activos de la capa virtual) utilizando un interfaz determinado, mientras los motores VWS pueden invocar a los proveedores delegados utilizando un interfaz distinto. El motor VWS será el responsable de realizar las modificaciones, para adaptar la mensajería recibida por un interfaz al formato que se espera recibir en el otro interfaz. Lo que es lo mismo: el motor VWS se encarga de

realizar cambios de esquema en la estructura de los mensajes SOAP recibidos, creando nuevos mensajes SOAP que siguen el esquema predefinido por el destinatario final de dichos mensajes.

Traduciendo esta propuesta a nuestra arquitectura, la capa virtual estaría formada por servicios web virtuales, que presentan un interfaz concreto a los clientes delegados. Cada método del servicio web virtual podrá invocar a uno o más proveedores delegados, y cada uno de éstos utilizará su propio esquema. El motor VWS, basándose en las especificaciones contenidas en los documentos VWSDL que describen los servicios virtuales, deberá realizar las transformaciones oportunas para obtener, a partir de los mensajes SOAP recibidos de los clientes delegados, nuevos mensajes SOAP que sigan los esquemas que definen los proveedores delegados y viceversa.

Es importante destacar que cada método virtual se puede implementar con varios servicios web reales, cada uno de los cuales puede seguir su propio esquema. Por ello será necesario disponer de transformaciones para cada uno de los proveedores virtuales por cada método que se publique en un servicio web virtual dentro de un motor VWS.

Estas transformaciones, como ya hemos comentado en capítulos anteriores, deben ser realizadas por el motor VWS en función del contexto en el que se esté ejecutando (servicio virtual, método y proveedor delegado), y pueden ser llevadas a cabo mediante dos mecanismos:

- Aplicando una hoja de estilo XSL.
- Aplicando una transformación simple STF (apéndice A).

El proceso completo es el que se muestra en la figura 18.1. Como podemos ver en dicha figura, las peticiones de invocación emitidas por la entidad cliente, pueden llegar al proveedor A o al proveedor B, dependiendo de las decisiones tomadas en el motor VWS. Para que esto funcione, y teniendo en cuenta que el proveedor A ofrece un Interfaz A, mientras que el proveedor B ofrece un Interfaz B (distinta del primero), es necesario aplicar transformaciones (STF en la figura) entre los formatos de los mensajes recibidos en el motor VWS (con el Interfaz A') y los formatos de los distintos proveedores (A y B).

18.3. Conclusiones

La introducción de la capa virtual ofrece una serie de ventajas importantes desde el punto de vista del acoplamiento:

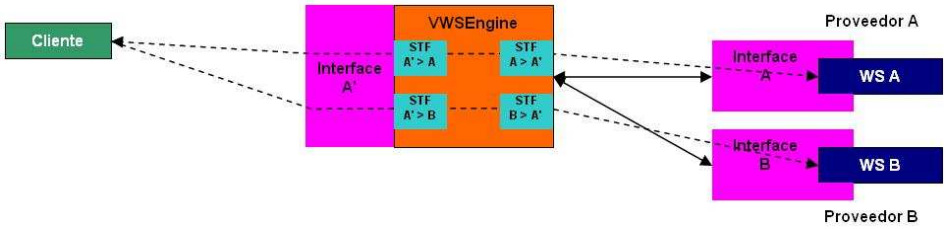

Figura 18.1. Transformación de mensajes entre cliente y proveedor.

- Mejora la adaptación a la tecnología de servicios web, ya que permite introducir modificaciones (nuevos proveedores) a medida que vayan estando disponibles sin necesidad de realizar ningún cambio en las aplicaciones cliente, ni siquiera el enlace. Los únicos cambios que se deben hacer han de ser realizados en la descripción del servicio web virtual, el documento VWSDL.

- Independiza los interfaces. Gracias a la introducción de los motores VWS entre clientes delegados y proveedores delegados, podemos obtener un alto grado de independencia de los interfaces, permitiendo que los proveedores realicen cambios en ellos sin necesidad de modificar las aplicaciones cliente; tan sólo adaptando los documentos VWSDL.

- Reduce el nivel de acoplamiento, lo que facilita la evolución de clientes y proveedores por separado. Las aplicaciones cliente pueden ser modificadas, mientras los proveedores delegados se mantienen intactos. El único cambio que ha de hacerse es en el interfaz del servicio web virtual. Una vez más, nos estamos refiriendo a modificaciones en los documentos VWSDL.

Como hemos comentado en otros capítulos, la capa virtual introduce un pequeño *overhead* que debe ser tenido en cuenta si las aplicaciones son elevadamente dependientes del tiempo de respuesta.

CAPÍTULO 19

Otras Aplicaciones

19.1. Acuerdos de Nivel de Servicio

19.1.1. Introducción

Derivado del uso de sistemas de calidad de servicio, nuestra arquitectura permite utilizar acuerdos de nivel de servicio (*Service Level Agreements*, SLA). Gracias al control que los motores VWS tienen sobre todas las invocaciones virtuales y reales, dichos motores permiten controlar la evolución de cada SLA firmada entre clientes y proveedores. Es decir, el responsable de velar por el cumplimiento de los objetivos especificados dentro de una SLA será el motor VWS. Asimismo, podrá ser el responsable de tomar las acciones oportunas cuando un objetivo haya dejado de cumplirse, o de notificar tal situación a la entidad responsable de emprender dichas acciones.

La capacidad de control que ofrecen los motores VWS hace que sean un punto muy propicio para el control del cumplimiento de los acuerdos de nivel de servicio y, al mismo tiempo, el lugar más adecuado para iniciar las acciones pertinentes derivadas de un ocasional incumplimiento de algún objetivo del acuerdo de nivel de servicio.

Como hemos visto en capítulos anteriores, la introducción de un motor VWS entre clientes delegados y proveedores delegados ofrece la posibilidad de disponer de un punto de control sobre todas las invocaciones que se realizan. Ya hemos visto la utilidad que esto tiene a la hora de, por ejemplo, construir un *cluster*, ya que permite disponer de un nodo controlador encargado de distribuir la carga y gestionar los errores en los nodos.

De hecho, disponer de un nodo central es la base de nuestra propuesta para la implementación de técnicas de calidad de servicio. El uso de expresiones en base a métricas predefinidas son las herramientas que nos permiten

implementar dichas técnicas.

Al mismo tiempo, la amplia gama de métricas utilizadas por los motores en el cálculo de expresiones, podría también ser utilizada para controlar la evolución de los proveedores delegados desde el punto de vista del control del cumplimiento de un determinado nivel de servicio.

Como vemos, la implantación de técnicas de utilización de acuerdos de nivel de servicio es muy sencilla partiendo de la base existente para la gestión de la calidad de servicio.

19.1.2. Propuestas

Una SLA es un contrato entre dos entidades por el cual una de ellas se compromete a ofrecer unos servicios a la otra cumpliendo un conjunto de requisitos, especificados también en el propio contrato. Esos requisitos son lo que denominamos objetivos. Del eventual incumplimiento de alguno de los objetivos especificados en un acuerdo, se pueden derivar acciones encaminadas a la disolución del contrato, a penalizaciones, o cualquier otra acción acordada entre las partes firmantes del contrato.

Un motor VWS será un punto muy adecuado para efectuar el control de los acuerdos de nivel de servicio que puedan establecerse entre clientes delegados y proveedores delegados. El proceso por el cual nuestra propuesta sería utilizada como un mecanismo de control de SLA sería el siguiente:

- **Establecimiento de la SLA**. Este proceso es ajeno al funcionamiento de nuestra arquitectura y consiste en la determinación de los elementos a medir y las acciones que han de ser llevadas a cabo en caso de incumplimiento. Los intervinientes son las partes firmantes, es decir, clientes del servicio que gestiona la SLA y proveedores de dicho servicio.

- **Despliegue de la SLA**. Una vez establecidos los puntos del acuerdo, la SLA debe ser instalada o desplegada en los motores VWS que corresponda. Como parte del funcionamiento de la SLA, será necesario especificar las fechas y horas de inicio y terminación de dicha SLA, así como los intervalos de tiempo (dentro de dichas fechas) en los que la SLA será aplicable. En los intervalos de no aplicabilidad la SLA estará vigente pero no activa, y al llegar la fecha de finalización, la SLA dejará de estar activa y vigente.

- **Explotación de la SLA**. Una vez que la SLA ha sido desplegada, los motores VWS que la hayan recibido deberán velar por el cumplimiento de los objetivos (SLO, *Service Level Objcetives*) especificados en ella. Los motores deberán tener en cuenta los periodos de vigencia

y actividad, y llevar a cabo las acciones necesarias cuando se detecte el incumplimiento de algún SLO.

Para especificar el uso de una SLA en un documento VWSDL, se puede utilizar el elemento `sla` contenido dentro de cada elemento `invoke`. El elemento `sla` hace referencia a una URL en la que un documento especificando los acuerdos de nivel de servicio debe ser tenido en cuenta a la hora de realizar una invocación hacia el proveedor especificado en el elemento `invoke` (ver ejemplo 19.1).

```
...
<method name="obtenerDatosCliente" type="equivalent">
  ...
  <invoke name="1" type="provider" id="p1"
          operation="getData"
          sla="http://sample.company.net/sla/agr-1.sla>

    <mapin  type="copy" />
    <mapout type="copy" />
  </invoke>
  ...
</method>
...
```

Ejemplo 19.1: Ejemplo de SLA.

19.1.3. Conclusiones

VWSDL no ha sido concebido como un lenguaje para la especificación de SLAs, simplemente ofrece la posibilidad de relacionar un documento VWSDL con una especificación de SLA.

Queda fuera del alcance de este trabajo analizar la estructura de un documento que recoja acuerdos entre partes dentro de una SLA. Como posibles vías de trabajo apuntamos la posibilidad de utilizar WSLA [106] como especificación a incluir dentro de los documentos VWSDL, mediante la referencia externa provista por el elemento `sla`.

19.2. Composición

19.2.1. Introducción

Es una tendencia de mercado, y una futura necesidad reconocida por todas las entidades implicadas en la construcción de la tecnología de servicios

web, alcanzar la madurez suficiente en la tecnología de tal forma que permita la construcción de nuevos servicios web mediante técnicas de agregación de servicios web más simples.

Estas tendencias han dado lugar a la aparición de distintas propuestas, como BPML [45], WSCI [46], XLANG [134], etc. Todas ellas son especificaciones que permiten la descripción de procesos de negocio basada en servicios web, habilitando de esta forma la construcción de nuevos servicios a partir de otros servicios web existentes. Los procesos de negocio también pueden ser analizados desde una óptica de interacción entre partes, por lo que algunas de las iniciativas mencionadas (y alguna más), permiten especificar procesos de negocio de forma abstracta, es decir, representando la interacción entre partes, independientemente de si estas partes están o no implementadas como servicios web.

19.2.2. Propuestas

El motor VWS, como elemento central de la arquitectura propuesta, es un gran candidato para permitir la construcción de flujos de trabajo (*workflow*). Debido a su interacción con la parte cliente y con la parte proveedora, y al hecho de que es una entidad que funciona en modo desatendido, puede dar soporte a la implementación de flujos, en los que el motor es el encargado de coordinar las invocaciones de un flujo de control.

La composición de servicios es un asunto suficientemente complejo como para dar lugar a un nuevo trabajo a partir de éste. Nosotros hemos reconocido las posibilidades de nuestra arquitectura como habilitadora de nuevas formas de construir servicios web.

Se construye servicios web virtuales por agregación de otros servicios web, reales o virtuales. El mecanismo utilizado para construir dicha agregación es la descripción de un flujo.

Creemos que la composición por agregación puede ser realizada mediante técnicas de programación sencillas, y es por eso que planteamos, a partir de nuestra propuesta, dos líneas de trabajo bien diferenciadas, aunque puedan confluir en los resultados obtenidos: programación y procesos de negocio.

Programación

Entendemos que nuestra propuesta basada en un elemento central, el motor VWS, nos brinda una gran oportunidad para construir un motor de **flujo de control** sobre el motor de servicios web virtuales. Es decir, mediante las especificaciones adecuadas, podemos lograr que el motor ejecute un conjunto de pasos, mediante lógica sencilla de programación, basada en los tradicionales recursos programáticos como pueden ser las estructuras: secuencial, `if-then-else` y `do-while`.

Estas estructuras básicas, junto con la ejecución secuencial de pasos, nos permitiría obtener un conjunto mínimo de pseudo-instrucciones de un lenguaje de programación basado en servicios web.

Las estructuras básicas pueden ser complementadas de forma sencilla mediante estructuras más complejas que permitan:

- La ejecución en paralelo de distintas tareas.
- La sincronización de procesos, es decir, especificar puntos del flujo de control en los que se debe esperar a que se produzca un conjunto de eventos.

Por último, un leguaje de programación como el que proponemos (WSPL, *Web Services Programming Language*) debe permitir realizar una gestión de errores. Nuestra propuesta en este sentido es una arquitectura de programación basada en excepciones, dada la potencia que ofrece sobre una gestión de errores clásica basada en códigos o condiciones.

Procesos de negocio

Los procesos de negocio están siendo objeto de continuos trabajos en el mundo comercial. Las dos grandes catalizadoras de los procesos de negocio basados en servicios web (Microsoft e IBM) han aunado sus esfuerzos, unificando sus distintos enfoques (XLANG y WSFL, respectivamente) para dar lugar al ya mencionado BPEL4WS.

BPEL4WS no se limita a la simple definición de procesos ejecutables de negocio (describe el comportamiento de cada participante). Va más allá, y permite definir procesos abstractos, es decir, lo que se denominan protocolos de negocio: el intercambio de mensajes entre las distintas partes involucradas en los procesos del negocio.

Desde esta óptica, BPEL va mucho más allá que nuestra propuesta de WSPL, ya que permite modelar procesos de negocio en los que, por ejemplo, haya intervención humana, no sólo servicios web.

19.2.3. Conclusiones

Una posible vía de extensión de nuestra arquitectura sería aquella implementada vía BPEL4WS. De manera similar a la propuesta que hemos mostrado para extender VWSDL con WSLA, se puede extender VWSDL con BPEL4WS para dotar a los servicios web virtuales de la facilidad de ser construidos mediante agregación.

La extensión de VWSDL con BPEL4WS no invalida WSPL. WSPL representa en realidad un subconjunto de BPEL4WS, una implementación

mucho más sencilla y menos ambiciosa que permite solventar problemas simples mediante técnicas de programación tradicionales.

19.3. Publicación y Suscripción con VWS

19.3.1. Introducción

El control de la mensajería que tiene un motor VWS permite dotar a la arquitectura de servicios web de capacidades de replicación de mensajes basadas en técnicas de publicación/suscripción. Para ello, los creadores de servicios web virtuales pueden autorizar la suscripción (por parte de un tercero) de un tipo de mensajes determinado. El motor actúa como publicador y ese tercero como suscriptor.

Como hemos apuntado en secciones anteriores, la utilización de un motor que convierta las invocaciones en indirectas tiene como resultado inmediato el hecho de que *todo* el tráfico entre clientes y proveedores viaja a través de los motores VWS. Esto otorga una posibilidad de control sobre la mensajería, pudiendo llegar a implementar una gestión inteligente de los mensajes. Esta gestión inteligente la entendemos como la posibilidad de obtener o generar valor añadido a partir del simple tráfico de mensajes. ¿Cómo? Pues a través de mecanismos de publicación/suscripción, utilizados con éxito en otros entornos como mensajería (JMS [11], MQSeries [15]) o bases de datos (DB2 [6], SQLServer [22], Oracle [17]).

El modelo de invocación tradicional (*request/reply*, petición/respuesta, R/R) es el que se aplica tradicionalmente a todo el desarrollo de aplicaciones distribuidas (componentes, EJB, DCOM, etc.) y es, además, el modelo de invocación estándar de los servicios web.

Cuando una aplicación cliente está mostrando un dato en pantalla, que es obtenido como resultado de una invocación a un agente externo (léase servicio web, componente, o lo que sea), nos encontramos con un problema: el *polling* o sondeo. Es decir, la aplicación se verá obligada a invocar periódicamente al proveedor del dato para que éste sea actualizado convenientemente. Si el sondeo es muy frecuente, probablemente se producirán muchas más invocaciones de las necesarias. Si el sondeo es poco frecuente, el número de invocaciones se reducirá, pero la "frescura" del dato podría no ser adecuada.

Para solventar problemas como éstos, surgen otros modelos de invocación, como el modelo *publish/subscribe* (publicación/suscripción, P/S). En un modelo P/S, existe una entidad que ofrece un servicio de información, el publicador; existe otra entidad que requiere dicho servicios, el suscriptor; y existe una tercera entidad en el modelo, que es el tema o asunto (*topic*).

El funcionamiento de un modelo P/S es bien sencillo. En primer lugar, un publicador ofrece un conjunto de temas sobre los que ofrece información. Por otra parte, los suscriptores buscan una información concreta. Cuando un suscriptor encuentra el publicador adecuado, ambos deben establecer un acuerdo y comenzar una relación del tipo P/S, en la que el publicador informará al suscriptor de cualquier cambio que se produzca en el tema.

19.3.2. Propuestas

Como parte de nuestra solución hemos aportado dos leguajes bien diferentes con los que implementar los trabajos de descripción de servicio web virtual (usando VWSDL), y los diálogos extendidos entre clientes delegados y proveedores delegados que permitan refinar los procesos relativos a invocaciones de servicios web virtuales (usando VWSEL).

Nuestra propuesta, que no será tratada aquí por razones que se explicarán a continuación, se basa en el uso de mecanismos similares a los explicados para el uso de sistemas *cache*. Es decir, intercambio de mensajes de control a través de las cabeceras SOAP para especificar las necesidades de suscripción. Las propuestas de publicación se ofrecen a través de VWSDL, mientras que la información de suscripción viaja a través de VWSEL (elemento `subscribe`) dentro de los mensajes SOAP.

Paralelamente a nuestro desarrollo, han surgido nuevas propuestas de estándares para dar soporte a los problemas de publicación/suscripción. Está ya en marcha un conjunto de trabajos encaminados a ofrecer soluciones a la implementación del modelo P/S en los servicios web. En [87] se introduce un conjunto de especificaciones agrupadas bajo la denominación WS-Notification:

- **WS-BaseNotification** [88]. Define la forma sencilla de intercambio de mensajes entre partes.
- **WS-BrokeredNotification** [89]. Define la forma de intercambio de mensajes a través de un intermediario. Esta especificación es, presumiblemente, la que se relacione con nuestro VWSDL.
- **WS-Topics** [90]. Define los mecanismos para organizar los asuntos susceptibles de ser publicados y suscritos.

19.4. Conclusiones

En general, los sistemas de comunicación basados en el patrón P/S son ampliamente utilizados para intercambio de información entre distintos

componentes conectados a través de una red y, como tal modelo de comunicación, será necesario implantarlo también en la tecnología de servicios web.

En general, el modelo P/S no es especialmente útil en aplicaciones cliente con usuarios, ya que el modelo más común en estos casos será un modelo R/R.

Los beneficios esperados de la implementación de un modelo P/S en las tecnologías de servicios web son bastantes, sobre todo en aplicaciones desatendidas, donde un origen de información comunica cambios de estado a un conjunto de destinos, receptores de dicha información, típicamente aplicaciones desatendidas.

La implementación de las técnicas P/S en nuestra propuesta dependerán mucho de la evolución del conjunto de estándares acogidos bajo la denominación común WS-Notificacion. Pero, al igual que ocurre con los acuerdos de nivel de servicio, la forma esperada de interactuar entre nuestra propuesta y WS-Notification será a través de recursos externos.

PARTE VI

Conclusiones

CAPÍTULO 20

Conclusiones y líneas futuras

20.1. Conclusiones

En el momento de comenzar este trabajo nos planteamos como objetivo fundamental la mejora de la arquitectura de servicios web actual, con el fin último de dotarla de dos funcionalidades principales:

- Alta disponibilidad.
- Calidad de servicio.

Para lograr estos dos objetivos establecimos como factor crítico de éxito la consecución de ambos a partir de una solución única. Esto fue lo que nos llevó a desarrollar nuestra teoría de la virtualización aplicada a los servicios web. Mediante el uso de servicios web virtuales, extendemos la arquitectura actual, dotándola de nuevas funcionalidades, que permiten obtener beneficios tan simples como necesarios en entornos de trabajo basados en servicios web. Algunos ejemplos son:

- Alta disponibilidad.
- Calidad de Servicio.
- Mejora de rendimiento.
- Asincronismo.
- Gestión de proveedores.
- Gestión simplificada de errores.

- *Brokering.*

La utilidad de los servicios virtuales para alcanzar estos objetivos ha sido publicada en artículos en conferencias especializadas en el campo de los *Web Services* ([77], [78], [79], [80]).

Es muy importante tener en cuenta que la solución propuesta ha sido desarrollada teniendo en mente siempre una condición muy importante: mantener **la compatibilidad**. Ésta ha sido cuidadosamente estudiada y conservada desde dos puntos de vista bien diferentes:

- **Compatibilidad externa**. Se refiere a la compatibilidad de nuestra solución con el resto de componentes de la tecnología de servicios web. En este caso hemos tenido en cuenta, fundamentalmente, que nuestra solución no fuese disruptiva, que pudiese implementarse conviviendo con la actual arquitectura de servicios web.

- **Compatibilidad interna**. Se refiere a la compatibilidad de nuestra solución cuando es aplicada a distintos problemas de los servicios web que tratamos de resolver. Esto ha sido tenido en cuenta, garantizando la existencia de una solución única para todos los problemas considerados.

Aunque nosotros hemos desarrollado nuestro trabajo orientado a solventar nuestros objetivos de partida (alta disponibilidad y calidad de servicio), nuestra solución es extensible y, sobre todo, aplicable a otros entornos, como hemos demostrado en [79] y en [80].

La compatibilidad a la que nos hemos estado refiriendo no es sólo arquitectural. Es decir, además de ser compatible con la arquitectura actual, debe serlo también con las entidades que forman dicha arquitectura. Así, un proveedor de la arquitectura actual puede serlo a la vez de nuestra arquitectura virtual. Paralelamente, un cliente que invoca un servicio web estándar, debe poder invocar un servicio web virtual de la misma forma. La compatibilidad con ambos extremos (cliente y proveedor) está garantizada gracias a la introducción de la capa intermedia de la arquitectura.

Una segunda restricción fue tenida en cuenta a la hora de desarrollar la solución: **la extensibilidad**. Al igual que ocurre con todas las técnicas y métodos que rodean a los servicios web, la extensibilidad de los lenguajes, protocolos o arquitecturas utilizados es algo crítico para garantizar la continuidad de los servicios web como tecnología. Nosotros también hemos tenido en cuenta este hecho inherente a los propios servicios web, y nuestra solución es perfectamente extensible, al basarse en estándares ya existentes, como el XML, el SOAP o el WSDL.

En realidad el desarrollo de nuestra propuesta se basa en la extensibilidad de los servicios web y, por tanto, no deja de ser extensible, ya que incorpora esquemas abiertos y públicos que pueden ser extendidos por cualquiera que desee mejorar nuestra arquitectura.

El término servicio web virtual (*Virtual Web Service*) ha sido utilizado en distintos trabajos, pero con significados distintos al que le hemos dado aquí. A menudo es utilizado para representar servicios que son ofrecidos a través de la web (*Word Wide Web*), nada que ver con el concepto software de servicio web. Sin embargo, existen otros trabajos en los que el término y el concepto de VWS que se manejan son muy similares a los que nosotros proponemos.

En [72], los servicios web virtuales son entendidos como una agregación de servicios web mediante operadores semánticos de agregación (*Semantic Aggregation Operators*). Dichos operadores permiten realizar distintas operaciones sobre los documentos XML obtenidos como resultado de invocar a todos los servicios web que forman un servicio web virtual, como puede ser una intersección, una unión, etc., dando como resultado un nuevo documento XML, que será el resultado de una invocación al servicio web virtual. Es decir, el objetivo que se persigue es el tratamiento de la información representada con documentos XML, como en el ya mencionado XMLMediator [7], por ejemplo.

En [118] se construyen servicios web virtuales mediante el uso de agentes. Se propone el uso de dos tipos de agentes: agentes de servicio (*Service Agents*, SAs) y agentes privados (*Private Agents*, PAs). Los SAs son agentes que invocan a servicios web y exponen sus funcionalidades (desarrollados como componentes EJB, por ejemplo). En la parte cliente, se construyen PAs, agentes privados, mediante llamadas a los agentes de servicio. Los PAs serán la base para construir los servicios web virtuales que serán luego consumidos como servicios web estándar. La construcción de servicios web virtuales mediante SAs y PAs persigue como objetivo la personalización: construir servicios web virtuales mediante agregación (utilizando agentes) de servicios web virtuales existentes que realicen la labor que interese a la entidad cliente. La personalización de servicios web ha sido especialmente concebida para su uso en dispositivos con capacidad de proceso limitada, como teléfonos móviles o PDAs.

La construcción de servicios por agregación está actualmente resuelta de un modo estándar con BPEL4WS [69]. Además, la utilización de agentes programados introduce una capa de acoplamiento fuerte entre servicios web reales y virtuales (contraria a los fundamentos de la tecnología de servicios web), que ha de ser convenientemente programada para alcanzar el objetivo de la personalización.

20.2. Líneas futuras

A lo largo del desarrollo de este trabajo han ido surgiendo oportunidades de mejora que no han sido desarrolladas aquí. Estas oportunidades, o líneas de trabajo, permitirán mejorar la arquitectura en red y nuestra propuesta de servicios web virtuales. Algunas posibles líneas de trabajo son:

- **Expresiones**. Como hemos explicado, una buena parte de las funcionalidades de nuestro trabajo se basa en el uso de expresiones. Éstas han sido desarrolladas dentro de este trabajo de una forma elemental, introduciendo los conceptos necesarios para establecer la base de su uso. Evidentemente, la complejidad de las cargas de trabajo gestionadas por un motor VWS pueden no ser modeladas adecuadamente con las funciones básicas que nosotros hemos construido. Funciones del tipo media ponderada, trabajar con subconjuntos de valores (para las facetas), media deslizante, etc., y otras muchas, pueden ser muy útiles a la hora de construir expresiones.

- **Integración**. Paralelamente al desarrollo de este trabajo, han ido surgiendo soluciones a otros problemas o carencias de los servicios web. Los mayores avances se han producido en el campo de la composición de servicios web, como ya hemos comentado. Pero también ha habido notables progresos en otras áreas, como en la mensajería asíncrona o en las arquitecturas de publicación y suscripción. Un trabajo importante que servirá para enriquecer las propuestas explicadas a lo largo de este trabajo, consistirá en la integración de nuestra propuesta de servicios virtuales con los nuevos estándares que han ido surgiendo.

- **Workflow**. Como ya hemos apuntado anteriormente, nuestra propuesta basada en el uso de un componente intermedio da origen a la vez a una oportunidad de control sobre las invocaciones. Dicho control puede ser utilizado para el desarrollo de flujos de ejecución de servicios web. Es decir, se plantea la posibilidad de construir nuevos servicios web por agregación de servicios existentes, mediante el uso de un lenguaje de programación, WSPL, que debería permitir la especificación de flujos de control.

- **Service Level Agreement**. La utilización de motores VWS como puntos intermedios de una invocación ofrece una oportunidad de control sobre dicha invocación. Esta oportunidad puede ser aprovechada para implementar técnicas de gestión de niveles de servicio. Trabajos paralelos a éste han dado como resultado un lenguaje de especificación de acuerdos de nivel de servicio, el WSLA [106]. Una posible vía de trabajo en el área de los acuerdos de nivel de servicio consistiría en estudiar y mejorar la actual integración de VWSDL y WSLA.

- **Sistemas Multi-Agente**. La utilización de múltiples proveedores para implementar un método de un servicio web virtual encaja perfectamente en las teorías que se aplican a los MAS (*Multi-Agent Systems*), los sistemas multi-agente. En nuestra propuesta, el motor VWS es el responsable de seleccionar un proveedor de servicio web (un agente), invocarlo, y pasarle el resultado al cliente delegado. Esta lógica de invocación podría modificarse para hacer funcionar nuestro motor VWS como un centro de control de un sistema multi-agente, permitiéndole invocar a varios proveedores a la vez y enviar la respuesta más adecuada a la entidad cliente según unos criterios que habrán sido fijados de antemano.

PARTE VII

Apéndices

CAPÍTULO A

SimpleTxForm

A.1. SimpleTxForm (STF)

El lenguaje de transformación simple (*Simple Transform* o *SimpleTxForm*, en adelante STF) ha sido diseñado como parte de este trabajo con los siguientes objetivos:

- Ofrecer a los creadores de documentos VWSDL una forma de adaptar los esquemas de datos que envían y reciben los clientes delegados a los que esperan recibir y enviar los proveedores delegados.

- Las transformaciones entre ambos esquemas serán presumiblemente sencillas, ya que es de esperar que no existan grandes diferencias entre el esquema de los clientes delegados y el de los proveedores delegados a la hora de representar un mismo tipo de objeto. Por ello, y debido a la sobrecarga de proceso que supone la ejecución de una transformación XSL, se ofrece un lenguaje mucho más limitado, pero a la vez mucho más sencillo.

- La mensajería sobre la que trabaja VWSDL es mensajería SOAP. Los objetos SOAP no requieren, en general, la utilización de atributos en los documentos XML más allá de los que corresponden a los espacios de nombres y a los tipos de datos. Así pues, el lenguaje propuesto sólo permite realizar modificaciones en los elementos de los esquemas origen y destino, nunca en los atributos si los hubiese.

El objeto principal de un documento STF es el elemento `transform`. A partir de un elemento de este tipo, se define una estructura jerárquica de

elementos `element`, que serán los que determinen la estructura del documento XML resultante. Cada elemento `element` debe especificar cual es el origen del elemento, así como la forma en la que ha de obtenerse.

```
<client>
  <name>
    <given>julio</given>
    <family>fernandez</family>
    <treat>Mr</treat>
  </name>
  <address>
    <street>Sol</street>
    <number>3</number>
    <zip>36200</zip>
    <location>Vigo</location>
    <province>Pontevedra</province>
  </address>
  <contact>
    <email>jfvilas@det.uvigo.es</email>
    <telephone>
      <work1>567898</work1>
      <work2>534899</work2>
      <personal>123456</personal>
    </telephone>
    <fax>
      <work>442365</work>
      <personal>372234</personal>
    </fax>
  </contact>
</client>
```

Ejemplo A.1: Documento XML de orgien.

Para explicar el funcionamiento de STF tomaremos como ejemplo base el documento XML mostrado en el ejemplo A.1. En dicho documento se muestran los datos de un cliente (`client`), estructurados de la siguiente forma: nombre, dirección e información de contacto.

A modo de ejemplo, empezaremos por crear un documento STF (ejemplo A.2) que nos permita transformar un documento del tipo `client` en uno del tipo `persona`. El resultado de la ejecución del documento STF del ejemplo A.2 se muestra en el ejemplo A.3.

La transformación del documento del ejemplo A.1 en el documento que se muestra en el ejemplo A.3 se basa en un documento STF que contiene (ejemplo A.2):

- A nivel superior un elemento del tipo `element` que contiene un atributo (`name`) cuyo valor es `persona`. Es decir, en el documento resultante,

```
<transform>
  <element name="persona">
    <element name="nombre"
             type="copy"
             source="/client/name/given"/>
    <element name="apellido1"
             type="copy"
             source="/client/name/family"/>
    <element name="direccion">
        <element name="postal"
                 type="copy"
                 source="/client/address" />
        <element name="electronica"
                 type="copy"
                 source="/client/contact/email" />
    </element>
    <element name="telefono"
             type="copy"
             source="/client/contact/telephone/personal"/>
  </element>
</transform>
```

Ejemplo A.2: Documento STF.

```
<persona>
  <nombre>julio</nombre>
  <apellido1>fernandez<apellido1>
  <direccion>
    <postal>
      <street>Sol</street>
      <number>3</number>
      <zip>36200</zip>
      <location>Vigo</location>
      <province>Pontevedra</province>
    </postal>
    <electronica>jfvilas@det.uvigo.es</electronica>
  </direccion>
  <telefono>123456</telefono>
</persona>
```

Ejemplo A.3: Documento resultante.

el elemento raíz será `persona`.

- A segundo nivel (jerárquicamente hablando), encontramos otros cuatro elementos del tipo `element`, que se corresponden con la creación de los elementos `nombre`, `apellido1`, `direccion` y `telefono` del documento resultante de la transformación.

- Los elementos resultantes `nombre`, `apellido1` y `telefono` se obtienen como una copia directa de un elemento origen: `/client/name/given`, `/client/name/family` y `/client/contact/telephone/personal`, respectivamente.

- El elemento resultante `direccion` se obtiene creando un elemento con dicho nombre y aplicando las reglas contenidas en el elemento origen identificado con el mismo nombre.

- Los elementos `postal` y `electronica`, contenidos en el elemento resultante `direccion`, se obtienen de la misma forma que los elementos explicados previamente.

- La copia de un elemento de un documento origen a uno destino se realiza tal cual, es decir, no se hace ningún tratamiento especial para diferenciar un elemento que sólo contiene un nodo texto de un elemento que contiene otros nodos del tipo elemento.

Como vemos, el lenguaje es muy sencillo, y ésta es precisamente una de las bases de su diseño. Si las transformaciones que se necesitan para la ejecución de un servicio web virtual no se pueden especificar utilizando STF, el diseñador del servicio deberá utilizar XSL.

Finalmente, hemos incluido en esta memoria el documento XSD que describe el lenguaje STF (figura A.1), así como un diagrama del mismo (figura A.2).

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema targetNamespace="http://schemas.uvigo.es/stf/1.0"
  xmlns="http://schemas.uvigo.es/stf/1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <xsd:element name="transform">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="element" type="elType" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="version" use="required" fixed="1.0"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="elType">
    <xsd:sequence>
      <xsd:element name="element" type="elType"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>

    <xsd:attribute name="name" type="xsd:NMTOKEN" use="required"/>
    <xsd:attribute name="source" type="xsd:NMTOKEN"/>
    <xsd:attribute name="type" type="xsd:string"/>

  </xsd:complexType>

</xsd:schema>
```

Figura A.1. Esquema del lenguaje STF.

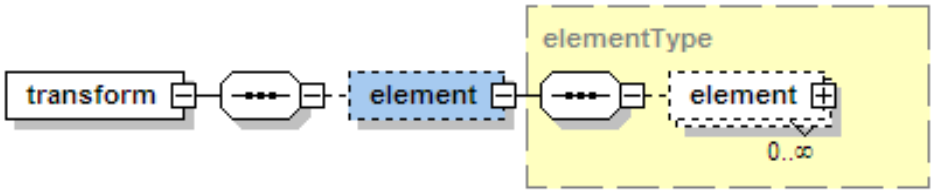

Figura A.2. Diagrama del lenguaje SimpleTxForm.

CAPÍTULO B

Lenguaje VWSDL

B.1. Introducción al lenguaje

El lenguaje VWSDL surge como una necesidad para crear descripciones de servicios virtuales. No se plantea en ningún momento como una extensión de WSDL, ya que la forma en la que se define un servicio web estándar y uno virtual es completamente distinta. La mayor parte de esta distinción proviene del hecho de que el servicio virtual es en realidad una abstracción de un servicio web real. Por lo tanto, no son necesarios, y no deberían existir, detalles relativos a la implementación, como el *binding* o la URL del servicio, por ejemplo.

Como hemos comentado en capítulos anteriores, VWSDL es un lenguaje independiente, y las razones para ello, entre otras, son las que acabamos de exponer. Otra razón fundamental para no crear VWSDL como un lenguaje *stand-alone* es que no tiene que ser entendido por los clientes delegados (aplicaciones cliente de un arquitectura de servicios web estándar), ni por los proveedores delegados (servicios web de dicha arquitectura).

El lenguaje VWSDL ha sido diseñado para permitir la descripción de un servicio web virtual que posteriormente será publicado como un servicio web real. Es decir, mediante mecanismos de transformación se podrá obtener un documento WSDL estándar que describa un servicio web real en términos compatibles con la arquitectura actual. De esta forma, las aplicaciones cliente pueden enlazarse con un servicio real, sin apercibirse del hecho de que dicho servicio esté implementado en realidad como un servicio virtual.

El lenguaje VWSDL es la base para la creación de los documentos VWSDL. Estos documentos serán consumidos por un motor VWS y nunca por una entidad cliente o una entidad proveedora. Cada documento VWSDL

describe un servicio web virtual que puede contener varios métodos.

¿Cómo se crea un documento VWSDL? Para crear un documento VWSDL no es necesario disponer de ningún software que implemente una lógica de negocio, como ocurre con el WSDL. Los documentos VWSDL se apoyan en documentos WSDL existentes para ofrecer nuevos servicios virtuales. De esta forma, para crear un documento VWSDL deberemos seguir los pasos principales que a continuación detallamos:

1. Búsqueda de proveedores. Lo primero que hemos de hacer es buscar proveedores delegados (otros servicios web y sus WSDL correspondientes) que puedan encargarse de la ejecución de la lógica de negocio que nosotros queremos publicar como servicio web virtual.

2. Definir los métodos. Deberemos, dentro de nuestro documento VWSDL, definir tantos métodos como deseemos, y asociar cada uno de ellos con el WSDL y `portType` correspondiente a la operación o método que se publica.

3. Definir las transformaciones. Para cada método, es posible que sea necesario realizar una transformación sobre la mensajería SOAP de entrada y/o salida. Utilizando XSL o SimpleTxForm deberemos especificar las reglas que permiten transformar los mensajes adecuadamente.

4. *Deploy*. Una vez finalizado, debemos enviar nuestro documento VWSDL a un motor VWS que será el encargado de ejecutarlo cuando sea invocado.

5. Publicación. El último paso consiste en publicar el servicio virtual para que pueda ser utilizado por aplicaciones cliente. Para ello, debemos obtener un documento WSDL a partir de nuestro VWSDL y publicarlo. Típicamente (y así lo hemos implementado en nuestro prototipo) esta es una operación automática que no conlleva trabajo de desarrollo alguno (solamente transformaciones de documentos).

B.2. Esquema del lenguaje

Como documentación adicional, las figura B.1, B.2, B.3, B.4 y B.5 muestran el esquema XSD que define el lenguaje VWSDL. Adicionalmente, un diagrama de cada uno de los elementos principales se muestra en la figura B.6.

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:stf="http://schemas.uvigo.es/stf/1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <xs:complexType name="cacheType">
    <xs:attribute name="from" type="xs:time" use="required"/>
    <xs:attribute name="to" type="xs:time" use="required"/>
    <xs:attribute name="check" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="data"/>
          <xs:enumeration value="method"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="interval" type="xs:byte"/>
  </xs:complexType>

  <xs:complexType name="defineType">
    <xs:attribute name="name" type="xs:NMTOKEN"
                  use="required"/>
    <xs:attribute name="expression" type="xs:string"
                  use="required"/>
    <xs:attribute name="type" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="typesType">
    <xs:sequence>
      <xs:any namespace="http://www.w3.org/2001/XMLSchema"
              processContents="strict"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="stylesheetType">
    <xs:sequence>
      <xs:any namespace="http://www.w3.org/1999/XSL/Transform"
              processContents="strict"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
  </xs:complexType>
  <xs:complexType name="transformType">
    <xs:sequence>
      <xs:any namespace="http://schemas.uvigo.es/stf/1.0"
              processContents="strict"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
  </xs:complexType>
```

Figura B.1. Esquema del lenguaje VWSDL (I).

```
<xs:complexType name="eventType">
  <xs:attribute name="type" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="fault"/>
        <xs:enumeration value="noConnect"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="iterate" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="retry"/>
        <xs:enumeration value="skip"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="iterations" type="xs:positiveInteger"
                use="required"/>
</xs:complexType>

<xs:complexType name="inputType">
  <xs:sequence>
    <xs:element name="parm" type="parmType"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
</xs:complexType>

<xs:complexType name="invokeType">
  <xs:sequence>
    <xs:element name="mapin" type="mapType"/>
    <xs:element name="mapout" type="mapType"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
  <xs:attribute name="type" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="method"/>
        <xs:enumeration value="provider"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="id" type="xs:string" use="required"/>
  <xs:attribute name="operation" type="xs:string"/>
  <xs:attribute name="next" type="xs:string"/>
</xs:complexType>
```

Figura B.2. Esquema del lenguaje VWSDL (II).

```
<xs:complexType name="mapType">
  <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
  <xs:attribute name="type" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="copy"/>
        <xs:enumeration value="stylesheet"/>
        <xs:enumeration value="transform"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>

<xs:complexType name="methodType">
  <xs:sequence>
    <xs:element name="event" type="eventType"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="select" type="selectType"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="cache" type="cacheType"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="invoke" type="invokeType"
                maxOccurs="unbounded"/>
    <xs:element name="input" type="inputType"/>
    <xs:element name="output" type="outputType"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
  <xs:attribute name="type" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="conditional"/>
        <xs:enumeration value="equivalent"/>
        <xs:enumeration value="iterative"/>
        <xs:enumeration value="sequential"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="default" type="xs:string"/>
  <xs:attribute name="extensionClient" type="xs:boolean"/>
  <xs:attribute name="extensionProvider" type="xs:boolean"/>
  <xs:attribute name="extensionRestrict" type="xs:boolean"/>
  <xs:attribute name="async">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="true"/>
        <xs:enumeration value="false"/>
        <xs:enumeration value="only"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
```

Figura B.3. Esquema del lenguaje VWSDL (III).

```
<xs:complexType name="outputType">
  <xs:sequence>
    <xs:element name="parm" type="parmType"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
</xs:complexType>

<xs:complexType name="parmType">
  <xs:attribute name="name" type="xs:NMTOKEN"
                use="required"/>
  <xs:attribute name="element" type="xs:string"
                use="required"/>
</xs:complexType>

<xs:complexType name="providerType">
  <xs:sequence>
    <xs:element name="define" type="defineType"
                minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NMTOKEN"
                use="required"/>
  <xs:attribute name="location" type="xs:anyURI"
                use="required"/>
  <xs:attribute name="check" type="xs:boolean"
                use="required"/>
  <xs:attribute name="service" type="xs:string"
                use="required"/>
  <xs:attribute name="port" type="xs:string"
                use="required"/>
</xs:complexType>

<xs:complexType name="selectType">
  <xs:sequence>
    <xs:element name="define" type="defineType"
                minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN"/>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="type" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="boolean"/>
        <xs:enumeration value="numeric"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="expression" type="xs:string"
                use="required"/>
  <xs:attribute name="default" type="xs:string"/>
  <xs:attribute name="alternate" type="xs:string"/>
</xs:complexType>
```

Figura B.4. Esquema del lenguaje VWSDL (IV).

```
  <xs:complexType name="serviceType">
    <xs:sequence>
      <xs:element name="cache" type="cacheType"
                  minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="define" type="defineType"
                  minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="event" type="eventType"
                  minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NMTOKEN" use="required"/>
    <xs:attribute name="group" type="xs:string" use="required"/>
    <xs:attribute name="application" type="xs:string"
                  use="required"/>
    <xs:attribute name="async">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="true"/>
          <xs:enumeration value="false"/>
          <xs:enumeration value="only"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>

  <xs:element name="definition">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="types" type="typesType"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="stylesheet" type="stylesheetType"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="transform" type="transformType"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="provider" type="providerType"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="method" type="methodType"
                    maxOccurs="unbounded"/>
        <xs:element name="service" type="serviceType"/>
      </xs:choice>
      <xs:attribute name="targetNamespace" type="xs:anyURI"
                    use="required"/>
      <xsd:attribute name="version" use="required" fixed="1.0">
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Figura B.5. Esquema del lenguaje VWSDL (V).

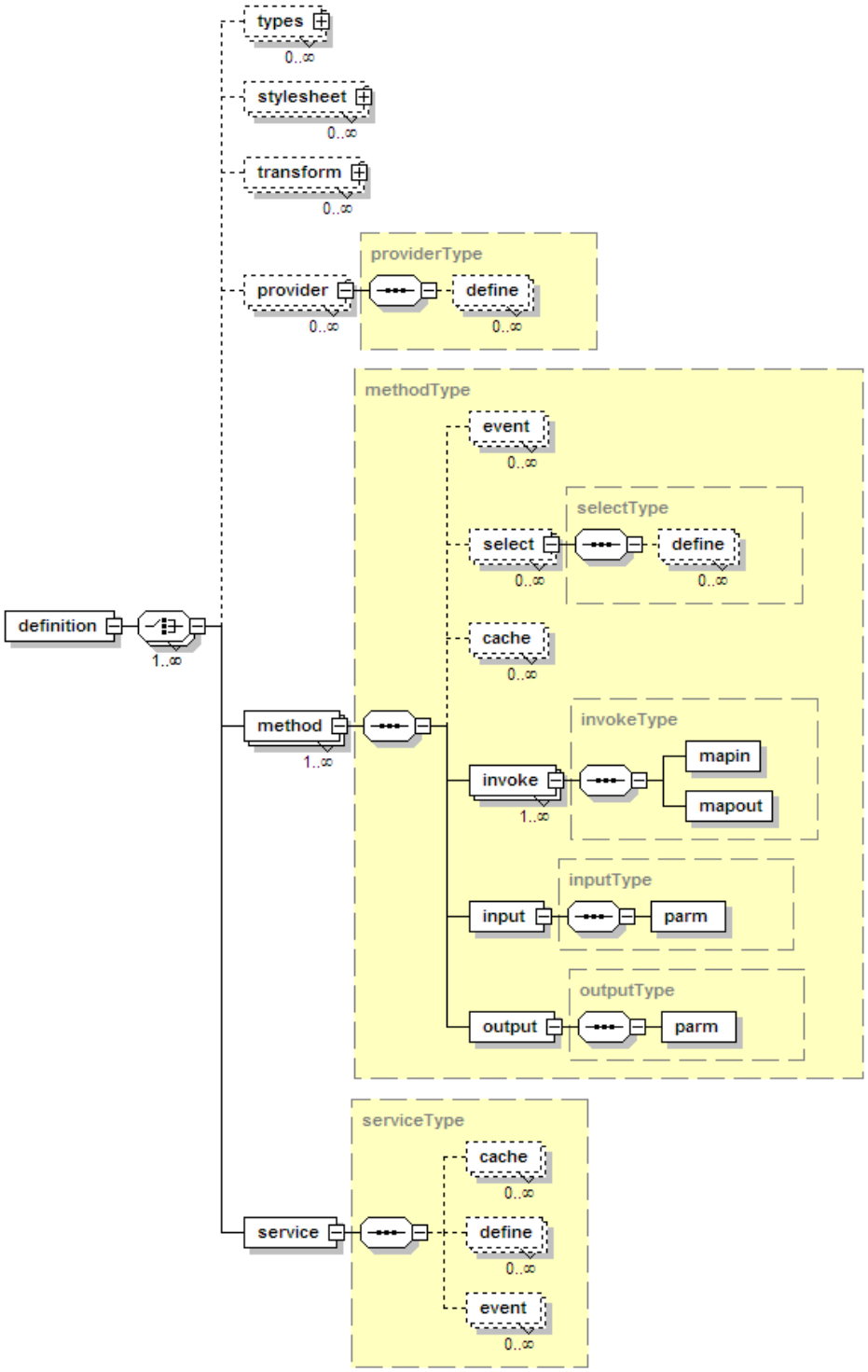

Figura B.6. Diagrama del lenguaje VWSDL.

CAPÍTULO C

Lenguaje VWSEL

C.1. Introducción al lenguaje

Ya hemos comentado en capítulos anteriores la necesidad que se plantea, tanto dentro como fuera de nuestra arquitectura, de intercambiar información de control entre aplicaciones cliente y servicios web. En el caso particular de nuestra propuesta dicha necesidad viene marcada por tres motivos principales:

1. En los sistemas de QoS es necesario dar al cliente la oportunidad de seleccionar o, al menos, influir en el proceso de selección de los proveedores delegados que ejecutarán un servicio virtual concreto.

2. En los sistemas de mejora basados en *caching* es necesario que clientes delegados y proveedores delegados intercambien información de control relativa a la validez y durabilidad de los mensajes.

3. Para dotar al sistema de invocación de los servicios web de una mayor flexibilidad se propone el uso de mensajería asíncrona. Para dar soporte a esta necesidad, clientes delegados y proveedores delegados necesitarán intercambiar información relativa a la disponibilidad de los mensajes de respuesta dentro de una modelo de invocaciones asíncronas.

El lenguaje VWSEL, aunque concebido inicialmente como una extensión de VWSDL para mensajería SOAP, puede en realidad utilizarse como lenguaje independiente en el caso de mejoras basadas en *cache*, ya que el sistema que proponemos no depende en realidad (para arquitecturas 2LCA) de la existencia de un componente intermedio, como puede ser un motor VWS.

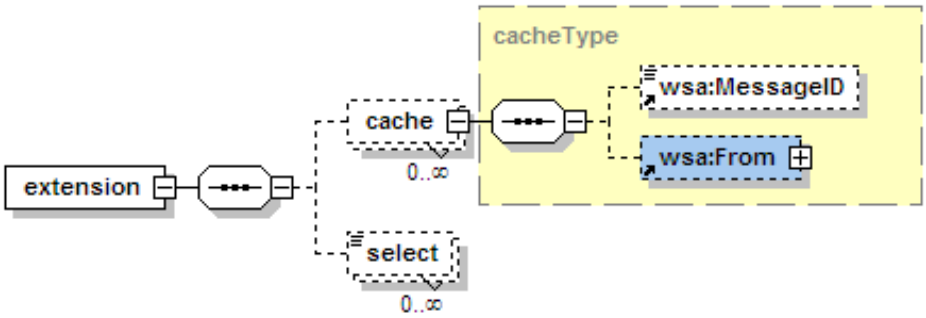

Figura C.1. Diagrama del lenguaje VWSEL.

Análogamente, VWSEL puede ser utilizado para dar soporte a la implementación del asincronismo entre clientes delegados y proveedores delegados, independientemente de si se utilizan o no servicios web virtuales.

VWSEL es un lenguaje sencillo, y esta sencillez fue una de las prioridades en el diseño del lenguaje, ya que cada mensaje SOAP que llegue a un proveedor o a un motor VWS debe ser analizado con el objetivo de extraer los documentos VWSEL que pueda contener e interpretarlos adecuadamente.

C.2. Esquema del lenguaje

Como documentación adicional, las figuras C.2 y C.3 muestran el esquema XSD que define el lenguaje VWSEL, así como un diagrama de cada uno de los elementos principales (figura C.1).

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema targetNamespace="http://schemas.uvigo.es/vwsel/1.0"
  xmlns="http://schemas.uvigo.es/vwsel/1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  elementFormDefault="qualified">

  <!-- Elemento raiz del documento de extension -->
  <xsd:element name="extension">

    <xsd:complexType>

      <xsd:sequence>

        <xsd:element name="async" minOccurs="0">

          <xsd:complexType>

            <xsd:sequence>
              <xsd:element ref="wsa:MessageID" minOccurs="0"/>
              <xsd:element ref="wsa:ReplyTo" minOccurs="0"/>
            </xsd:sequence>

            <xsd:attribute name="type" use="required">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="none"/>
                  <xsd:enumeration value="poll"/>
                  <xsd:enumeration value="callback"/>
                  <xsd:enumeration value="notification"/>
                  <xsd:enumeration value="message"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>

            <xsd:attribute name="response" use="required">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="notification"/>
                  <xsd:enumeration value="message"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>

          </xsd:complexType>

        </xsd:element>
```

Figura C.2. Esquema del lenguaje VWSEL (I).

```
        <xsd:element name="get" minOccurs="0">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="wsa:MessageID" minOccurs="0"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
         <xsd:element name="select" minOccurs="0">
          <xsd:complexType>
            <xsd:attribute name="type" use="required">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="client"/>
                  <xsd:enumeration value="engine"/>
                  <xsd:enumeration value="restrict"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
          </xsd:complexType>
        </xsd:element>


        <xsd:element name="cache" minOccurs="0">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="wsa:MessageID" minOccurs="0"/>
              <xsd:element ref="wsa:From" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="type" type="xsd:string"
                           use="required"/>
            <xsd:attribute name="enabled" type="xsd:boolean"/>
            <xsd:attribute name="interval" type="xsd:short"/>
          </xsd:complexType>
        </xsd:element>

      </xsd:sequence>

      <xsd:attribute name="version" use="required" fixed="1.0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="1.0"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>

    </xsd:complexType>

  </xsd:element>

</xsd:schema>
```

Figura C.3. Esquema del lenguaje VWSEL (II).

CAPÍTULO D

Prototipo

D.1. Arquitectura

Con el objetivo de validar la arquitectura propuesta hemos desarrollado un prototipo que implementa algunas de las principales funciones propuestas en este trabajo. Este prototipo, completamente funcional, no ha sido concebido como un producto final, ni servirá como una prueba de validación del rendimiento, ya que ha sido construido con el mero objetivo de comprobar la validez de la solución propuesta. Por ello, y aunque el desarrollo ha sido llevado a cabo con el mayor esmero desde el punto de vista arquitectural, es una implementación claramente mejorable desde el punto de vista del rendimiento. El prototipo cumple dos objetivos básicos:

- Por una parte, como ya hemos comentado, sirve como elemento para la validación inicial de nuestras propuestas.
- Por otra, sirve como implementación de referencia a la hora de construir motores VWS que implementen nuestra solución basada en servicios web virtuales.

D.2. Características Técnicas

El motor ha sido desarrollado sobre plataforma .Net. El lenguaje seleccionado para dicha implementación ha sido C#, por ser éste considerado como el estado del arte en materia de lenguajes de programación. C# no representa, desde nuestro punto de vista, ninguna mejora considerable sobre Java (ambos son claramente orientados a objetos y multiplataforma), y ha sido elegido por una cuestión de familiaridad con el entorno de desarrollo.

La aplicación desarrollada en C#.Net fue compilada como una aplicación web, y desplegada para ser ejecutada sobre Internet Information Server [16], en el que se pueden instalar sin mayor problema las aplicaciones ASP.Net (en la plataforma seleccionada, los servicios web son en realidad aplicaciones ASP.Net).

La aplicación ha sido desarrollada para ser ejecutada sobre .Net versión 1.1, con lo que puede ser compilada y ejecutada en cualquier plataforma Linux, utilizando Mono [25] y Apache [2] como base para la ejecución de la aplicación ASP.Net sin cambio alguno.

D.3. Características Funcionales

El prototipo, en su versión actual, implementa las siguientes funciones elementales de nuestra propuesta:

- Es capaz de entender y representar con objetos propios del lenguaje cualquier documento VWSDL.
- Puede evaluar expresiones que incluyan variables y constantes, así como expandir y computar expresiones complejas creadas a partir de facetas de usuario o facetas complejas. No soporta facetas temporales, ya que no está incluido el soporte para el acceso a un repositorio de *rating* externo.
- Es capaz de seleccionar un proveedor entre un conjunto de candidatos en función de los resultados obtenidos en el proceso de evaluación.
- Dispone de mecanismos de *caching*.
- Es capaz de seleccionar proveedores alternativos en función de ciertas condiciones de error.
- Soporta el uso de ámbitos (*scope*) en la definición de las variables, y, por tanto, los tiene en cuenta a la hora de evaluar expresiones, gestionar eventos, o realizar *caching* de los mensajes recibidos y/o enviados.

A partir de las funciones primitivas que hemos detallado anteriormente, el prototipo ha sido utilizado para realizar pruebas de las siguientes funcionalidades de nuestra propuesta:

- Alta Disponibilidad. Hemos construido un *cluster* de servicios web con el prototipo, demostrando de esta forma la funcionalidad de alta disponibilidad de que hemos hablado en capítulos anteriores. Para

facilitar la creación de *clusters* de este tipo, la aplicación dispone de un diálogo interactivo en el que, a partir de un documento WSDL, se puede crear **automáticamente** un documento VWSDL que permita la publicación y ejecución de dicho servicio como servicio web virtual, implementado en un conjunto de servidores que han de ser especificados en el momento de la generación del servicio virtual.

- Mejora de rendimiento. Hemos utilizado con éxito nuestro prototipo para optimizar el acceso a un servicio web, mediante el uso de técnicas de *caching*.

- Calidad de servicio. De forma similar a cómo hemos realizado pruebas de alta disponibilidad, hemos desarrollado y probado distintos tipos de expresiones que nos permiten incluir técnicas de calidad de servicio en los procesos de selección de servicios web.

- Gestión simplificada de proveedores. Hemos realizado pruebas de agregación, modificación y eliminación de proveedores en los servicios web virtuales, sin que ello afecte a los clientes delegados. Más aún, dichas pruebas fueron realizadas mientras un simulador lanzaba peticiones de invocación ininterrumpidamente y, como resultado de la prueba, todas las modificaciones fueron realizadas sin que el cliente delegado se apercibiese de ello.

- Gestión de errores. Utilizando los simuladores que desarrollamos para las pruebas de alta disponibilidad, introducimos errores forzados en los nodos del *cluster*, comprobando como los clientes del simulador seguían realizando invocaciones completamente ajenos a los errores en dichos nodos.

D.4. Experiencias

Hemos realizado distintos experimentos con el prototipo, obteniendo resultados importantes y claramente beneficiosos en los entornos de producción de servicios web. Hemos construido dos simuladores de carga que nos han permitido verificar las bondades de nuestro sistema cuando se encuentra en un entorno de producción de carga elevada.

El primer simulador, *Naoussa*, emula un conjunto de clientes lanzando peticiones contra un motor VWS, quien se encarga de invocar a dos servicios web reales.

El segundo simulador, *Rothenburg*, ha sido utilizado para comprobar la capacidad de nuestro sistema como una forma de mejorar el rendimiento en aplicaciones en base a sistemas de *caching*.

Ambos simuladores, y los resultados de los experimentos, han sido tratados específicamente en los correspondientes capítulos relativos a alta disponibilidad, mejora de rendimiento, gestión de proveedores, etc.

Bibliografía

[1] Apache Axis. In *http://ws.apache.org/axis/*.

[2] Apache Software Foundation. In *http://www.apache.org/*.

[3] Business Process Management Initiative. In *http://www.bpmi.org*.

[4] CISCO Content Services. In *http://www.cisco.com/en/US/products/hw /contnetw/ps792/index.html*.

[5] Customer Information Control System. In *http://www-306.ibm.com/ cics*.

[6] DataBase 2 (DB2). In *http://www-306.ibm.com/software/data/db2*.

[7] EntireX XML Mediator. In *http://www1.softwareag.com/Corporate/ products/entirex/default.asp*.

[8] GDPS. In *http://www-03.ibm.com/servers/eserver/zseries/gdps/*.

[9] Gentran Integration Suite. In *www.sterlingcommerce.com/Products/ AllProducts/Gentran*.

[10] Internet Engineering Task Force. In *http://www.ietf.org*.

[11] Java Message Service (JMS). In *http://java.sun.com/jms*.

[12] Microsoft Biztalk Server. In *www.microsoft.com/biztalk*.

[13] Microsoft Custer Services. In *http://www.microsoft.com/ windows-server2003/technologies/clustering/default.mspx*.

[14] Microsoft Virtual Server. In *www.microsoft.com/virtualserver*.

[15] MQ Series. In *http://www.ibm.com/mqseries*.

[16] MS Internet Information Services. In *www.microsoft.com/iis*.

[17] Oracle Database. In *http://www.oracle.com/database*.

[18] Organization for the Advancement of Strcutured Information Standards (OASIS). In *http://www.oasis-open.org*.

[19] Security Assertion Markup Language (SAML). In *http://www.oasis-open.org/ committees/tc_ home.php?wg_ abbrev=security*.

[20] Simple Workflow Access Protocol (SWAP). In *www.ics.uci.edu /ietf-swap/*.

[21] Software AG. In *http://www.softwareag.com*.

[22] SQL Server. In *http://www.microsoft.com/sql*.

[23] Stryon Software. In *http://www.stryon.com*.

[24] Systinet Server. In *http://www.systinet.com/products/systinet_ - server*.

[25] The Mono Project. In *http://www.go-mono.com/rationale.html*.

[26] Universal Description Discovery and Integration (UDDI). In *http://www.uddi.org*.

[27] Veritas Software. In *http://www.veritas.com/vcs*.

[28] VMWare. In *http://www.vmware.com*.

[29] Web Services Invocation Framework. In *http://ws.apache.org/wsif*.

[30] Webmethods Enterprise Services Platform. In *http://www.webmethods.com/Products/ESP*.

[31] WebSphere Application Server. In *http://www.ibm.com/websphere*.

[32] World Wide Web Consortium. In *http://www.w3c.org*.

[33] WS-Choreography Working Group. In *http://www.w3.org/2002 /ws/chor*.

[34] WS-I. In *http://www.ws-i.org/*.

[35] XML Protocol. In *http://www.w3.org/2000/xp/Group*.

[36] Xtradyne. In *http://www.xtradyne.com*.

[37] *Information Technology - Open Systems Interconnection - Basic Refernce Model: The Basic Model*. ISO/IEC, 2 edition, 1994.

[38] Web Services Transactions Specifications. *developerWorks (http://www-128.ibm.com/developerworks/library/specification/ws-tx)*, Nov 2004.

[39] F. A. Schreiber A. Maurino, B. Pernici. Adaptive Channel Behavior in Financial Information Systems. In *Proceedings of Information Systems for a Connected Society Ubiquitous Mobile Information and Collaboration Systems (CAiSE'03)*, pages 77–89, Jun 2003.

[40] Holt Adams. Asynchronous operations and Web services (Part 1): A primer on asynchronous transactions. *developerWorks (http://www-128.ibm.com/developerworks/webservices/library/ws-asynch1.html)*, Apr 2002.

[41] Holt Adams. Asynchronous operations and Web services (Part 2). *http://www-128.ibm.com/developerworks/webservices/library/ws-asynch2*, Jun 2002.

[42] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. Extensible stylesheet language (XSL) 1.0. *http://www.w3.org/TR/xsl*, 2001.

[43] Peter G. Aitken. *XML The Microsoft Way*. Pearson Education, 2001.

[44] J.P.A. Almeida, L.F. Pires, and M.J. van Sinderen. Web services and seamless interoperability. European Workshop on Object Orientation and Web Services (EOOWS), July 2003.

[45] Asaaf Arkin. Business Process Modeling Language (BPML) specification. *http://www.bpmi.org/bpml-spec.htm*, Jun 2002.

[46] Asaaf Arkin. Web Service Choreography Interface 1.0 Specification. *http://ftpna2.bea.com/pub/downloads/wsci-spec-10.pdf*, Jun 2002.

[47] Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach, John Manferdelli, Hiroshi Maruyama, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, John Shewchuk, and Dan Simon. Web Services Security (WS-Security). *developerWorks (http://www.ibm.com/developerworks/library/ws-secure)*, Apr 2002.

[48] Paola Bari, Pierre Cassier, Annamaria Defendi, John Hutchinson, Alain Maneville, Gianfranco Membrini, and Caleb Ong. *Programmers Guide to Workload Manager*. IBM, 2004.

[49] Douglas K. Barry. *Web Services and Service-Oriented Architectures: The Savvy Manager's Guide*. Ed. Morgan Kauffman, 2003.

[50] Mark Bartel, John Boyer, Barb Fox, Brian LaMacchia, and Ed Simon. XML-Signature Syntax and Processing. *http://www.w3.org/TR/xmldsig-core/*, Feb 2002.

[51] V. Batos, K. Fertalj, and D. Kalpic. Application of e-commerce in banking industry. In *Proceedings of the 25th International Conference on Information Technology Interfaces ITI*, pages 107–112, Jun 2003.

[52] David Bau and David Orchard. SOAP Conversation Protocol (SOAP Conversation) 1.0. *http://dev2dev.bea.com/pub/a/2002/06 /SOAP-Conversation.html*, Jun 2002.

[53] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. *http://www.w3.org/TR/ws-arch*, Feb 2004.

[54] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Marc Nottingham, David Orchard, John Shewchuk, Eugene Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. Web Services Addressing. *http://www.w3.org/Submission/ws-addressing/*, 2004.

[55] Don Box, Aaron Skonnard, and John Lam. *Essential XML:Beyond Markup*. Addison-Wesley, 2000.

[56] Neil Bradley. *The XML Companion*. Addison-Wesley, 2001.

[57] Paul A. Buhler and Jose M. Vidal. Enacting BPEL4WS Specified Workflows with Multiagent Systems. In *Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

[58] M. Burstein, C. Bussler, T. Fininand M.N. Huhns, M. Paolucci, A.P. Sheth, S. Williams, and M. Zaremba. A semantic Web services architecture. *IEEE Internet Computing*, 9(5):72–81, Sep-Oct 2005.

[59] M. H. Burstein. The many faces of mapping and translation for semantic Web services. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering WISE 2003*, pages 261–268. IEEE, Dec 2003.

[60] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. *IEEE Transactions on Computers*, 47(4):445–457, Apr 1998.

[61] Ethan Cerami. *Web Services Essentials*. O'Reilly and Associates, 2002.

[62] Shenze Chen and D. Towsley. A performance evaluation of RAID architectures. *IEEE Transactions on Computers*, 45(10):1116–1130, Oct 1996.

[63] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 1.2. http://www.w3.org/TR/wsdl12, 2002.

[64] Ka-Po Chow and Yu-Kwong Kwok. On load balancing for distributed multiagent computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(8):787–801, Aug 2002.

[65] E. Cohen and H. Kaplan. Refreshment policies for web content caches. In *Proceedings of IEEE INFOCOM*, pages 1398–1406, Apr 2001.

[66] Frank Cohen. Debunking SAML myths and misunderstandings. *developerWorks (http://www-128.ibm.com/developerworks/xml/library/x-samlmyth.html)*, Jul 2003.

[67] Marco Conti, Mohan Kumar, Sajal K. Das, and Behrooz A. Shirazi. Quality of Service Issues in Internet Web Services. *IEEE Transactions on Computers*, 51(6):593–594, June 2002.

[68] Microsoft Corporation. *Microsoft Windows 2000 Server Resource Kit*. Microsoft Press, 2000.

[69] Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, and Sanjiva Weerawarana. Business Process Execution Language for Web Services. *developerWorks (http://www-106.ibm.com/developerworks/webservices/library/ws-bpel)*, Jul 2002.

[70] Asit Dan, Heiko Ludwig, and Giovanni Pacifici. Web Services Differentiation with Service Level Agreements. *ftp://ftp.software.ibm.com/software/websphere/webservices/ webserviceswithservicelevelsupport.pdf*, 2003.

[71] S/390 Division. High Availability Web Services. *http://www-1.ibm.com/servers/eserver/zseries/networking/haws.html*, 2000.

[72] A. Dogac, G. Laceli, S. Kirbas, Y. Kabak, S. Sinir, A. Yildiz, and Y. Gurcan. Artemis: Deploying Semantically Enriched Web Services in the Healthcare Domain. *Information Systems Journal (Elsevier)*, 31(4-5):321–339, Jun-Jul 2006.

[73] S. Subramaniamand D.L. Eager. Affinity scheduling of unbalanced workloads. In *Proceedings of Supercomputer 94*, pages 214–226, Nov 1994.

[74] S. A. Ehikioya and B. Ola. A comparative study of specification methods for electronic commerce systems. In *Proc. of 3rd ACS/IEEE International Conference on Computer Systems and Applications*, page 108, 2005.

[75] A. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[76] D. Kuoand A. Fekete, P. Greenfield, J. Jang, and D. Palmer. Just what could possibly go wrong in B2B integration? In *Proceedings. 27th Annual International Computer Software and Applications Conference COMPSAC*, pages 544–549, Nov 2003.

[77] Julio Fernandez, Jose Pazos, and Ana Fernandez. An architecture for building web services with Quality-of-Service features. In *Proceedings of the Fiveth Web-Age Information Management (WAIM)*, Jul 2004.

[78] Julio Fernandez, Jose Pazos, and Ana Fernandez. High availability with clusters of web services. In *Proceedings of the sixth Asia-Pacific Web Conference (APWeb)*, pages 644–653, Apr 2004.

[79] Julio Fernandez, Jose Pazos, and Ana Fernandez. Optimizing Web Services performance using Caching. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP)*, pages 157–162, Aug 2005.

[80] Julio Fernandez, Jose Pazos, and Ana Fernandez. Optimizing Web Services performance using Caching. In *Journal of Advanced Computational Intelligence and Intelligent Informatics*, page Por publicar, Aug 2006.

[81] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster Based scalable network services. In ACM, editor, *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Oct 1997.

[82] E. Frank. *DCOM: Microsoft Distributed Component Object Model*. Hungry Minds Inc, 1999.

[83] John Fuller, Mayilraj Krishnan, Keith Swenson, and Jeffrey Ricker. Asynchronous Service Access Protocol (ASAP) Version 1.0. *http://www.oasis-open.org/committees/download.php/14210/wd-asap-spec-02e.doc*, May 2005.

[84] Nick Gall. SOAP Bubbles B2B through the Firewall. 802, Dec 1999.

[85] Yaron Goland, Mark Nottingham, and David Orchard. WS-CallBack Protocol 0.91. *http://dev2dev.bea.com/ webservices/WS-CallBack-0_ 9.html*, Feb 2003.

[86] Brian D. Goodman. Accelerate your Web services with caching. *developerWorks journal (http://www-128.ibm.com/developerworks/webservices/library/ws-cach1 /?ca=degrL19wscaching)*, Dec 2002.

[87] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedhukin, D. Snelling, S. Tuecke, W. Vambenepe, and B. Weihl. Publish-Subscribe Notifications for Web Services. *developerWorks (http://www-128.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf)*, May 2004.

[88] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedhukin, D. Snelling, S. Tuecke, W. Vambenepe, and B. Weihl. Web Services Base Notification (WS-BaseNotificaction). *ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf*, May 2004.

[89] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedhukin, D. Snelling, S. Tuecke, W. Vambenepe, and B. Weihl. Web Services Brokered Notification (WS-BrokeredNotification). *ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BrokeredN.pdf*, May 2004.

[90] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Sedhukin, D. Snelling, S. Tuecke, W. Vambenepe, and B. Weihl. Web Services Topics. *ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-Topics.pdf*, May 2004.

[91] J. Gray and D. P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, Sep 1991.

[92] P.D. Groom. The IT security model. *IEEE Potentials*, 22(4):6–8, Oct-Nov 2003.

[93] IEEE 802.1Q Working Group. *802.1q: Virtual LANs*. IEEE, 2001.

[94] XML Protocol Working Group. Simple Object Access Protocol (SOAP) 1.2. *http://www.w3.org/TR/SOAP*, Jun 2003.

[95] Rosa Laura Zavala Gutiérrez and Michael N. Huhns. *On Building Robust Web Service-Based Applications*, chapter 14, pages 293–310. Kluwer Academic Publishing, 2004.

[96] J. Gwertzman and M. Seltzer. World-Wide Web cache consistency. In *Proceedings of USENIX Annual Technical Conference*, pages 141–151, 1996.

[97] J.B. Hundehege. Evolutionary development of multichannel-banking architecture. In *Proceedings of International Conference on Advances*

in Infrastructure for e-Business, e-Education, eScience, e-Medicine, and Mobile Technologies on the Internet, 2001.

[98] IBM. *S/390 virtual image facility for Linux*. IBM, 3 edition, 2001.

[99] Louis W. Storms IV and Scott J. Peterson. *Microsoft Windows DNA Exposed*. Sams, 1999.

[100] Li jie Jin, Vijay Machiraju, and Akhil Sahai. Analysis on Service Level Agreement of Web Services. *HP Software Technology Laboratory*, Technical Report HPL-2002-180, Jun 2002.

[101] R. J. Kauffman and H. Mohtadi. Information technology in B2B e-procurement: open vs. proprietary systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences HICSS*, pages 2129–2138, Jan 2002.

[102] Michael Kay. *XSLT Programmer's Reference*. Wrox Press, 2001.

[103] Farnk Leymann. Web Services Flow Language (WSFL) 1.0. *http://www-4.ibm.com/software/solutions/webservices/pdf/wsfl.pdf*, May 2001.

[104] D. Li and D.R. Cheriton. Scalable Web Caching of Frequently Updated Objects using Reliable Multicast. In *Proceedings of 2nd USENIX Symposium on Internet Technologies and Systems*, pages 1–12, Oct 1999.

[105] D. Linthicum. *Enterprise Application Integration*. Addison-Wesley, 1999.

[106] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richar Franck. Web Service Level Agreement (WSLA) Language Specification. Aug 2002.

[107] A. Mani and A. Nagarajan. Understanding quality of service for Web services. *developerWorks journal (http://www-106.ibm.com/developerworks/library/ws-quality.html)*, Jan 2002.

[108] S. McIlraith, T. Son, and H. Zeng. Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53, Apr 2001.

[109] D. A. Menasce. QoS Issues in Web Services. *IEEE Internet Computing*, 6(6):72–75, Nov/Dec 2002.

[110] J. Mendling and M. Miller. A Comparison of BPML and BPEL4WS. In *1st Conference "Berliner XML-Tage"*, pages 305–316, 2003.

[111] Microsoft. Perform Output Caching with Web Services in Visual C# .NET. (Art. Id. 318299), Sep 2003.

[112] Eric Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI.* Addison-Wesley, 2002.

[113] Eric Newcomer, Mark Little, and Greg Pavlik. Web Services Coordination Framework Specification. *http://www.oasis-open.org/committees/download.php/15042/WS-CF.zip*, Oct 2005.

[114] Jeffrey M. Nick, Jen-Yao Chung, and Nicholas S. Bowen. Overview of IBM System/390 Parallel Sysplex. In *Proceedings of 10th International Parallel Processing Symposium (IPPS)*, pages 488–495. IEEE Computer Society, Apr 1996.

[115] K. V. Pandya, A. Diamond, and R. Bhogal. Research issues in e-business. In *Portland International Conference on Management of Engineering and Technology PICMET '01*, pages 29–37, Aug 2001.

[116] D. Rogerson. *Inside COM.* Microsoft Press, 1997.

[117] E. Roman, S. Ambler, T. Jewell, and F. Marinescu. *Mastering EJB.* John Wiley & Sons, 2001.

[118] Jarogniew Rykowski and Wojciech Cellary. Virtual Web Services - Application of Software Agents to Personalization of Web Services. In *Proceedings of the 6th international conference on Electronic Commerce*, pages 409–418. ACM Press, 2004.

[119] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Li Jie Jin, and Fabio Casati. Automated SLA Monitoring for Web Services. *13th IFIP/IEEE International Wokshop on Distributed Systems*, pages 28–41, 2002.

[120] D. A. Sanders, L. M. Cremaldi, V. Eschenburg, C. N. Lawrence, C. Rileyand D. J. Summers, and D L. Petravick. Redundant Arrays of IDE Drives. *IEEE Transactions on Nuclear Science*, 49(4):1834–1840, Aug 2002.

[121] Ron Schmelzer, Travis Vandersypen, Jason Bloomberg, Madhu Siddalingaiah, Sam Hunting, Michael Qualls, Chad Darby, David Houlding, and Diane Kennedy. *XML and Web Services Unleashed.* Ed. Sams, 2002.

[122] Daniel Scholler. Enterprise Application Integration Design: Part 1 - The medium is the message. *Ed. Metagroup*, (Delta File 858), Apr 2000.

[123] Daniel Scholler. Enterprise Application Integration Design: Part 2 - Practical Principles of EAI Design. *Ed. Metagroup*, (Delta File 910), Oct 2000.

[124] T. Schroeder, S. Goddard, and B.Ramamurthy. Scalable Web server clustering technologies. *IEEE Network*, 41(3):38–45, Jun 2000.

[125] Scott Seely and Kent Sharkey. *SOAP: Cross Platform Web Service Development Using XML*. Prentice Hall, 2001.

[126] S.S. Y. Shim, V. S. Pendyala, M.Sundaram, and J. Z. Gao. Business-to-business e-commerce frameworks. *IEEE Computer*, 33(10):40–47, Oct 2000.

[127] K. Siau, Ee-P Lim, and Z. Shen. Mobile commerce: promises, challenges, and research. *Journal of Database Management*, 12(3):4–13, Jul-Sep 2001.

[128] Aaron Skonnard and Martin Gudgin. *Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema*. Addison-Wesley, 2001.

[129] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly and Associates, 2001.

[130] A. K. Somani and T.R T. R. Sarnaik. Reliability analysis techniques for complex multiple fault tolerant computer architectures. *IEEE Reliability*, 39(5):547–556, Dec 1990.

[131] Keith Swenson and Jeffrey Ricker. Asynchronous Web Services Protocol (AWSP). *http://xml.coverpages.org/AWSP-Draft20020405.pdf*, Apr 2002.

[132] Santra Technologies. Quality of Service for Web Services Defined. *http://www.santra.com/knowledge/?id=qos*, 2002.

[133] F. Teraoka and M. Tokoro. Host migration transparency in IP Networks: The VIP approach. *Proceedings of SIGCOMM*, 23(1):45–65, Jan 1993.

[134] Stish Thatte. XLANG: Web Services for Business Process Design. *http://www.gotdotnet.com/team/xml_wsspecs/xlang-c /default.htm*, 2001.

[135] D. R. Tojib, L. F. Sugianto, and M. M. Rahim. A new framework for B2E portal development. In *Proc. of IEEE International Conference on e-Technology e-Commerce and e-Service EEE*, pages 711–714. IEEE, Apr 2005.

[136] Larry Velez. Intelligent Network Routing Infrastructure: Part 2 - Service Design. Delta File 939, Dec 2001.

[137] C. Venkatapathy and S. Holdsworth. An introduction to Web Services Gateway. *developerWorks (http://www-106.ibm.com /developerworks/webservices/library/wsgateway/)*, 2002.

[138] C. Vinosky. Web services interaction models. Current practice. *IEEE Internet Computing*, 6(3):89–91, May-June 2002.

[139] W3C. XML Schema Part 0: Primer Second Edition. *http://www.w3.org/TR/xmlschema-0/*, Oct 2004.

[140] Eden Remme Watt, Don Denoncourt, Scott Lee, Bob Cancilla, Rick Stevens, and Lynda Stickelmaier. *Understanding e-business Application Integration.* MC Press, 2002.

[141] Scott Weller. Web services qualification. *developerWorks (http://www-106.ibm.com/developerworks/library/ws-qual)*, April 2002.

[142] François Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/2004/REC-xml-20040204/*, 2004.

[143] L. Zadeh. Knowledge Representation in Fuzzy Logic. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):89–100, 1989.