



SOME RIGHTS RESERVED

Por Joan Ribas Lequerica

Cuando alguien te diga que algo es imposible, no discutas ni lo ignores, simplemente demuéstrale que está equivocado.

Dedicado a mi familia, a los que soportan día a día mis locuras y a todos aquellos a los que nunca les hayan dedicado nada.



Atribución-NoComercial-CompartirDerivadasIgual 2.5 Spain

Se puede:

- copiar, distribuir, exhibir, y ejecutar la obra
- para hacer obras derivadas

Bajo las siguientes condiciones:



Atribución. Usted debe atribuir la obra en la forma especificada por el autor o el licenciente.



No Comercial. Usted no puede usar esta obra con fines comerciales.



Compartir Obras Derivadas Igual. Si usted altera, transforma, o crea sobre esta obra, sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.

- Ante cualquier reutilización o distribución, usted debe dejar claro a los otros los términos de la licencia de esta obra.
- Cualquiera de estas condiciones puede dispensarse si usted obtiene permiso del titular de los derechos de autor.

Sus usos legítimos u otros derechos no son afectados de ninguna manera por lo dispuesto precedentemente.

Este es un resumen legible-por-humanos del Código Legal (la licencia completa) disponible en los siguientes lenguajes:

[Catalan](#) [Spanish](#) [Galician](#)

[Disclaimer](#)

Índice

Cómo usar este libro.....	7
Introducción.....	8
1.1. Evolución.....	8
1.2. Una primera visión sobre los web services.....	10
1.2.1. ¿Qué es un web service?.....	10
1.2.2. ¿Cómo se realiza todo esto?.....	12
1.2.3. Topología del web service.....	12
1.2.3.1. SOAP.....	14
1.2.3.2. WSDL.....	15
1.2.3.3. UDDI y WSIL.....	15
1.2.3.4. Encaminamiento.....	15
1.2.3.5. Seguridad.....	15
1.2.3.6. Coordinación y transacciones entre web services	16
1.2.4. Empaquetado de datos.....	16
1.3. Futuro de los web services.....	16
El lenguaje XML.....	18
2.1. Introducción.....	18
2.2. ¿Qué es exactamente XML?.....	19
2.3. Contenidos del documento XML.....	23
2.3.1. Comentarios.....	24
2.3.2. Elementos sin atributos con datos.....	24
2.3.3. Elementos con atributos y datos.....	25
2.3.4. Marcas con atributos sin datos.....	26
2.3.5. Elementos vacíos.....	27
2.3.6. Entidades.....	27
2.4. Cabeceras en XML.....	28
2.5. Validadores XML.....	29
2.5.1. El fichero de validación DTD.....	31
2.5.1.1. Los elementos.....	33
2.5.1.1.1. Elementos sin contenido.....	33
2.5.1.1.2. Elementos conteniendo datos.....	33
2.5.1.1.3. Elementos conteniendo otros elementos.....	34
2.5.1.1.4. Elementos conteniendo modificadores	35
2.5.1.2. Atributos:.....	37
2.5.1.3. Entidades.....	39
2.5.1.4. Relación entre DTD y documento XML.....	39
2.5.1.5. DTD AVANZADO.....	41
2.5.2. El XML Schema	42
2.5.2.1. Tipos de datos.....	45
2.5.2.1.1. Tipos de datos simples.....	45
2.5.2.1.2. Tipos de datos complejos.....	49
2.6. Los NAMESPACES.....	54
2.7. Analizadores.....	62
2.7.1. SAX.....	63
2.7.2. DOM.....	63
2.8. Conclusión.....	63
El lenguaje SOAP.....	65
3.1. Introducción.....	65
3.2. El mensaje SOAP.....	65
3.2.1. Tipos de mensajes.....	66
3.3. Constitución del mensaje SOAP.....	67
3.3.1. Envelope	68
3.3.1.1. Versión.....	69
3.3.2. Cabecera.....	70
3.3.2.1. mustUnderstand	71
3.3.2.2. actor o role.....	72
3.3.2.3. encodingStyle.....	73

3.3.3. Cuerpo.....	73
3.4. Empaquetado de datos.....	73
3.4.1. ¿Qué tipos se pueden serializar?.....	74
3.4.1.1. Tipos simples.....	74
3.4.1.1.1. Cadenas de texto.....	75
3.4.1.1.2. Enumeraciones.....	75
3.4.1.1.3. Arrays de bytes.....	76
3.4.1.1.4. Estructuras.....	76
3.4.1.1.5 Referencias.....	76
3.4.1.2. Arrays.....	78
3.4.1.2.1 Array de arrays.....	79
3.4.1.2.2. Arrays sin límites.....	79
3.4.1.2.3. Arrays parciales.....	81
3.4.1.2.4. Arrays referenciados elemento a elemento.....	81
3.4.1.3. Elementos nulos.....	82
3.4.1.4 Elementos defecto.....	82
3.5. Respuestas	82
3.6. Faults.....	84
3.6.1. Elementos faultcode estándar de SOAP.....	85
3.6.2. Elementos faultcode del usuario.....	86
3.7. SOAP con Attachments.....	87
El documento WSDL.....	91
4.1 El documento WDSL.....	91
4.2. Importancia del WSDL.....	92
4.3. Descripciones del WSDL.....	93
4.4. Morfología.....	94
4.4.1. Preámbulo.....	94
4.4.2. Descripción.....	95
4.4.3. Información de mensajes.....	96
4.4.4. Localización del servicio	97
4.4. Ejemplo práctico.....	98
Los lenguajes UDDI y WSIL.....	104
5.1 Introducción.....	104
5.2 UDDI.....	104
5.2.1 Localización de UDDI.....	107
5.2.2. Información almacenada en UDDI.....	108
5.2.2.1. BusinessEntity (Páginas blancas)	109
5.2.2.2. BusinessService (Páginas amarillas)	111
5.2.2.3. BindingTemplate (Páginas verdes)	113
5.2.2.4 tModel.....	115
5.2.2.5 PublisherAssertion.....	118
5.2.3. Mensajes de control del registro.....	120
5.2.3.1 Mensajes de búsqueda general.....	120
5.2.3.2 Mensajes de búsqueda específica.....	121
5.2.3.3 Mensajes de publicación.....	121
5.2.3.4 Mensajes de borrado.....	122
5.2.3.5 Otros mensajes.....	123
5.2.4 Identificadores.....	123
5.2.5. Exploradores UDDI.....	125
5.3 WSIL.....	126
5.3.1 Estructura de WSIL.....	126
5.3.2 Publicación del documento.....	128
5.3.2.1 Nombre fijo.....	128
5.3.2.2 Documento enlazado.....	129
5.3.4 Ejemplo de documento WSIL.....	130
Ejemplos prácticos.....	134
6.1. Introducción.....	134
6.2. Hola mundo Perl.....	134
6.2.1. Servidor Perl.....	137
6.2.3. Cliente Perl.....	139

6.2.4. Cliente en Java.....	140
6.3. euroConversor Perl.....	143
6.3.1. Servidor Perl.....	143
6.3.2. Cliente Perl.....	145
6.3.3. Cliente Java.....	147
6.3.4. Cliente Delphi.....	149
6.3.5. Cliente Visual Basic.....	153
6.4. Hola Mundo Java.....	154
6.4.1. Servidor Java.....	156
6.4.2. Cliente Java.....	161
6.4.3. Cliente Perl.....	163
6.4.4. El WSDL.....	164
6.4.5. Cliente Delphi.....	167
6.5. Attachments.....	173
6.5.1. Servidor.....	174
6.5.2. Cliente.....	176
6.6. Otras consideraciones.....	179
Encaminamiento.....	181
7.1. Introducción.....	181
7.2. WS-Routing.....	181
7.2.1. Mecanismos de rutado.....	182
7.2.2. Los elementos	183
7.2.3. Protocolos	188
7.2.4 Errores	189
Seguridad.....	192
8.1. Introducción.....	192
8.2. Protección del mensaje.....	192
8.3. Ejemplo.....	193
8.4. Cabecera de seguridad.....	195
8.4.1. Elementos de seguridad.....	197
8.4.1.1. Elementos de nombre de usuario.....	197
8.4.1.2. Elementos XML.....	199
8.4.1.3. Elementos binarios.....	199
8.4.2. Firmas digitales.....	199
8.4.3. Cifrado.....	203
8.4.3.1. Datos cifrados.....	205
8.4.3.2. Claves cifradas.....	207
8.5. Otras consideraciones.....	208
Cooperación entre Servicios Web.....	210
9.1. Introducción.....	210
9.2 WS-Coordination.....	210
9.3 WS-Transaction.....	211
9.3.1 Transacciones atómicas.....	211
9.3.2 Actividades empresariales.....	211
9.4 BPEL4WS.....	211
Apéndice A.....	213
A.0. Introducción.....	213
A.1. Instalación y configuración.....	213
A.2. Registry Browser.....	214
B.2. Ejemplos de mensajes.....	216
Apéndice B.....	229
B.0 Introducción.....	229
B.1 TcpTunnelGui.....	229
B.1 tcpTrace.....	232
Apéndice C.....	236
C.0 Glosario.....	236

Cómo usar este libro

Esta guía práctica está pensada para introducir al lector en el mundo de los web services, una tecnología emergente que desde su aparición, ha recopilado toda clase de elogios y alabanzas. Un ejemplo claro de lo que puede suponer para la tecnología informática, es lo rápido que las grandes empresas como SAP, Sun Microsystems, IBM o Microsoft entre otras muchas han adoptado dicha tecnología par incluirla en sus productos.

Gracias a esta guía, el lector conocerá el funcionamiento del conjunto de procesos que se dan lugar dentro de los web services, así como las entrañas de cada uno de dichos procesos y mensajes que se generan. Estos mensajes se analizarán y el lector aprenderá a diferenciar las distintas partes de los mismos y conocer su utilidad.

En el primer capítulo se centra en describir el entorno en el que se trabajará, problemas que se plantean y soluciones adoptadas. El lector podrá obtener una visión global de los web services y de su funcionamiento.

En el segundo capítulo se hace una incursión en el lenguaje XML, lenguaje estándar sobre el que se sustentan los mecanismos de comunicación entre las distintas partes que ínter operan en un entorno basado en web services. Este lenguaje es clave para los servicios web, por lo que aunque el lector esté familiarizado con él, es aconsejable su lectura.

En el tercer capítulo se conocerá el lenguaje de transmisión de mensajes entre dos web services o web service y aplicación. Se explicará el lenguaje SOAP.

El cuarto capítulo está dedicado a la interfaz del web service, su descripción y su morfología, incluyendo un pequeño ejemplo práctico. Es el capítulo dedicado al documento WSDL.

El quinto capítulo hace referencia a la publicación y descubrimiento de web services, se explicará la forma de dar publicidad al servicio y los mecanismos disponibles para buscar un web service (o una empresa) en concreto por todo Internet. Se verán los estándares WSIL y UDDI.

El sexto capítulo está dedicado a unos ejemplos prácticos de lo anteriormente visto, en él se desarrollarán tanto servicios, como clientes para éstos en varios lenguajes. Aunque la guía no está pensada para enseñar al lector a programar servicios web en un lenguaje determinado, en este capítulo se comentan ejemplos sencillos para mostrar el uso de lo explicado en capítulos anteriores. Los lenguajes utilizados serán Delphi, Visual Basic, Java y Perl. También se demuestra como sacarle partido al documento WSDL visto en el cuarto capítulo, para la generación de clientes automáticos.

En el capítulo séptimo se introducirá al lector en el encaminamiento de web services, que pese a que no es una característica estándar de las implementaciones SOAP, si es útil en algunas ocasiones donde se maneja información sensible.

En el octavo capítulo se verá la importancia que puede llegar a tener la seguridad en los web services, y se explicarán los métodos más comunes para conseguirla, tales como firmas digitales y sistemas de encriptación.

En el noveno capítulo se explica la cooperación entre distintos servicios web, y los mecanismos de coordinación existentes para realizar procesos distribuidos en los que entran en juego varios web services diferentes. Se presentan estándares nuevos como BPEL4WS.

Por último se añade un apéndice donde se podrá encontrar, un glosario de las abreviaturas usadas, que son muchas y, seguramente casi todas ellas, nuevas para el lector.

Para acabar con esta pequeña introducción del libro, me gustaría alentar al lector para que, sobre todo en el capítulo sexto, no se quede sólo con la información de la guía e investigue, ya que ésta es una pequeña parte del mundo de los servicios web, que podrá comprobar que es muy extenso y cambiante, apareciendo día a día nuevas aplicaciones y especificaciones, y variándose las ya existentes.

A lo largo de toda la obra se incluyen especificaciones y ejemplos, pero en algunos casos no ha podido ser por cuestiones de espacio, en su lugar se proporcionan referencias a los documentos de especificación, para que el lector pueda consultar si le surge alguna duda (direcciones de Internet y documentos RFC) o revisar si se ha modificado la especificación, ya que como se ha comentado anteriormente, al ser tecnologías recientes, están sufriendo muchas variaciones y apareciendo nuevas aplicaciones de éstas. Los documentos RFC pueden consultarse en las direcciones:

- <http://www.ietf.org/rfc/rfcXXXX.txt>
- <http://www.faqs.org/rfcs/rfcsXXXX.html>

Donde XXXX es el número de RFC, por ejemplo para la especificación RFC 2104 sería:
<http://www.ietf.org/rfc/rfc2104.txt> o <http://www.faqs.org/rfcs/rfcs2104.html>

Introducción

Hoy en día la información es una de las partes más importantes de nuestra vida cotidiana, mensajes como los anuncios de televisión, no son más que un flujo de datos con una finalidad propia: la información. Toda esta información necesita unos medios físicos para poder transmitirse, y como parte integradora de estos medios están las tecnologías informáticas, para las cuales la información es un elemento base. Las empresas comparten datos no solamente entre departamentos, sino que van más allá, compartiendo información entre empresas, cooperando en procesos o incluso ofreciendo servicios de tratamiento de datos, como validaciones, procesados o normalizados.

Hasta hace poco, todos estos flujos de información se debían realizar teniendo en cuenta, entre otros aspectos, los canales por los que circulaba, y sobre todo, las arquitecturas que entraban en juego durante el intercambio, ya que tenían que ser comunes, o como mínimo semejantes. Como puede suponer el lector esta circunstancia, es una traba importante a la hora de compartir datos, ya que obliga a tener tecnologías semejantes a nuestros clientes, o como mínimo métodos para pasar de una tecnología a otras y más si se tiene en cuenta el número de soluciones posibles que surgen de combinar sistemas operativos, bases de datos, procesador de textos, protocolos de red....

Un ejemplo muy gráfico es la fusión de dos empresas que trabajan con sistemas informáticos distintos; imagine que una trabaja todo con tecnología Microsoft y la otra con software libre. ¿Cómo compartirán ahora los datos?, ¿Habrà que codificar de nuevo las aplicaciones?, ¿Habrà que hacer migraciones?, etc.

El web service nos va a permitir inter-operar entre sistemas basados en tecnologías distintas o que se encuentran simplemente alejados físicamente, acortando sensiblemente los tiempos de adaptación entre proveedores de servicios y clientes, y por lo tanto ahorrando costes, recortando tiempos de recuperación de la inversión y dando mejor servicio.

Los web services se han convertido en uno de los pilares de la evolución informática actual, compañías como IBM, BEA, o Microsoft se han volcado en su desarrollo e integración, tanto en sus productos anteriores como en los nuevos, para dar la posibilidad de trabajar con ellos; abriendo así el abanico de integración entre distintos procesos empresariales y distintas tecnologías.

Los web services se nos presentan como la solución a los problemas de comunicación entre en un parque de sistemas informáticos, cada vez más heterogéneo y disperso físicamente, permitiendo una rápida adaptación a los sucesivos cambios de estrategias empresariales, gracias a su rápida integración e independencia de la plataforma y/o entorno.

1.1. Evolución

Con la llegada de Internet a la sociedad, el abanico de posibilidades en el mundo de los negocios se incrementa y el escaparate de las ofertas se mide a escala mundial. Los clientes potenciales no son ya simplemente los que nos rodean, sino que nuestros servicios pueden ofrecerse ahora a todo el mundo, pudiendo captar clientes con distintas necesidades; esto significa que si una empresa se dedica a la generación de ficheros CAD, es posible que unos clientes lo quieran en un formato y otros clientes lo quieran en otro formato distinto, o incluso en idiomas diferentes, por esto mismo se necesita, ahora más que nunca, que los procesos sean flexibles y capaces de adaptarse en el menor tiempo posible a cada una de las necesidades que se presenten.

Pero no todo son ventajas puesto que la competencia ha pasado de igual modo a ser mundial. ¿Cuál es la diferencia entre comprar por Internet a un proveedor francés y a un español suponiendo tiempos de entrega semejantes? La diferencia está en aportar algo más que la competencia, en agilizar los trámites y, como no, en facilitar el acceso al cliente a los productos.

Gracias a Internet, el intercambio de información entre distintas delegaciones de una misma empresa es algo trivial y rápido, incluso brinda la posibilidad de realizar procesos distribuidos, etc. Esto no siempre ha sido así, antes la información se debía transferir de ordenador a ordenador mediante discos o bien mediante conexiones lentas y/o limitadas en distancia.

En un entorno en el que el intercambio de datos es tan importante, se debe tratar de mejorar los flujos de información, superando todas las barreras que vayan apareciendo. Está claro que una acción tan

habitual como escribir un correo electrónico, hace veinte años era totalmente impensable, y ahora es una de las acciones más comunes dentro de las oficinas.

Uno de los aspectos que se debe de tener en cuenta en la evolución del mundo de la informática es la aparición de la programación orientada a objeto, ya que éste supuso un gran avance en este mundo. Con su aparición se van a poder realizar módulos independientes capaces de realizar tareas concretas, esto es, se van a crear y vender piezas de código compilado, que no tienen ejecución propia, pero que al ser llamadas por unos parámetros concretos, permitirán la recepción de una respuesta.

Mediante la utilización de estos objetos, es posible abstraer la lógica que éstos encierran, y tratarlos como piezas de código independientes, pudiendo ser llamadas desde cualquier parte del programa y permitiendo además su reutilización y un fácil mantenimiento e integración.

Suponga que una empresa genera un objeto capaz de contar las palabras de una cadena de texto enviada. Sería posible vender el objeto como tal, es decir, el cliente no tendrá que compilarlo, ni entender su funcionamiento interno, sólo habrá que decirle la manera de crear dicho objeto y la lista de llamadas que ofrece al usuario (API). En este caso, habría que proporcionar el nombre de la función que devuelve el número de palabras cuando se le pasa una cadena. La limitación que existe en este caso es que el programador que lo integre deberá usar una tecnología semejante a la que se usa en este objeto, ya que un programa de Visual Basic no puede llamar a un objeto Java (sin usar *bridges* o puentes de transformación de datos), por usar tecnologías totalmente distintas.

Un paso más en la carrera de la flexibilidad en los procesos es la aparición de objetos distribuidos. Esta tecnología permite guardar los objetos llamados por la aplicación en una máquina diferente de la que realizará las llamadas. Imagine que se ha vendido el objeto capaz de contar palabras a una empresa con doscientos oficinistas que hacen uso del mismo. Hasta la aparición de los objetos distribuidos, el programa se debía de instalar en todos y cada uno de los ordenadores, pero suponga ahora que se desarrolla una versión más rápida del objeto que cuenta palabras. Para actualizar las instalaciones de los ordenadores a la nueva versión, se debía instalar de nuevo en los doscientos ordenadores. Con la aparición del objeto distribuido sólo es necesario instalar el elemento en un ordenador central y, configurando la red de manera adecuada, todas las aplicaciones harán uso de este nuevo objeto. Actualizando solamente este último, las doscientas instalaciones utilizarán la última versión.

Aún así el mayor inconveniente que se sigue planteando es que las tecnologías usadas deben seguir siendo semejantes; no es posible llamar a un objeto con tecnología CORBA mediante una llamada COM+ (sin usar elementos de transformación).

Otro problema que se plantea es la posibilidad de hacer llamadas remotas también a través de Internet, y no limitarse solamente a la intranet de la empresa. Pero no debe dejar a cualquiera usar los servicios de la empresa, ni se debe abrir el cortafuegos para poder realizar llamadas remotas, por lo que se necesita buscar soluciones que ofrezcan la posibilidad de usar los objetos desde cualquier punto, sin olvidar la seguridad informática de la empresa, y a ser posible usando protocolos ya existentes (para facilitar la adaptación del entorno).

El método utilizado para superar este escollo, es el uso de los puertos que ya se encuentran abiertos, para “colar” la información necesaria para llamar a los objetos. Como uno de los puertos más usados por su interactividad, es el del protocolo HTTP (Hyper Text Transfer Protocol), que se usa para el servicio de páginas web (puerto 80), fueron este puerto y protocolo, los elegidos por la mayoría de las soluciones, entre ellas la más extendida, llamada HTTP Tunneling, que se basa en el “camuflaje” de los datos de llamada del objeto, dentro de una llamada normal HTTP.

La pega de esta técnica, es que existen varias implementaciones, con distintos métodos de codificación de datos, por lo que al no existir un estándar, si queremos usar los servicios de otras empresas usando HTTP Tunneling, primero debemos llegar a un acuerdo con ellos, sobre la manera en la que trabajaremos.

Así pues necesitamos un lenguaje que sea capaz de transmitir información de forma estructurada, con tipo (que soporte la especificación de tipos en los datos), de manera descentralizada y que sea común a todos los sistemas, fácil de implementar e integrar, además debe ser ligero, escalable y modular.

Aunque parezca mentira, actualmente existe un protocolo que permite todo esto: SOAP (Simple Object Access Protocol). Aunque SOAP no es el único protocolo de codificación de mensajes para el uso de web services, sí es el más extendido y el que mayor número de implementaciones tiene.

SOAP no es un protocolo de transmisión, por lo que se debe apoyar en protocolos de envío y recepción de datos ya existentes en el mundo de Internet, tales como el HTTP, SMTP, FTP, etc., o incluso en protocolos no estándar como el Jabber.

SOAP ofrece más de lo mencionado, ya que no obliga a usar ningún modelo de programación, ni ninguna semántica específica, ni ningún lenguaje, simplemente define los mecanismos para representar la información a transmitir, ofreciendo los medios para codificarla e introducirla de manera modular.

SOAP es el estándar buscado en este tipo de transmisiones ya que, además, ofrece posibilidades de usar distintos protocolos de envío de la información tales como HTTP, SMTP, Jabber... por lo que cada empresa puede elegir el que más le convenga.

1.2. Una primera visión sobre los web services

1.2.1. ¿Qué es un web service?

Cada vez se lee más que compañías de software ofrecen sus productos con la posibilidad de usar web services, pero ¿qué es un web service?

Rigurosamente se diría que un web service es un objeto que tiene todas o parte de sus funciones accesibles mediante protocolos de red.

Un web service no es nada “nuevo” en conjunto, sino que es simplemente un nuevo uso de las herramientas que ya existían, para conseguir que los servicios informáticos puedan abrir su abanico de clientes, su facilidad y versatilidad de implantación, aunando varias tecnologías mediante estándares acogidos por las grandes empresas de la informática.

El hecho de anunciar a bombo y platillo el uso o el soporte de los web services por parte de una empresa o un producto, tiene su parte de razón; puesto que ayuda a la computación entre sistemas heterogéneos, pudiendo compartir información y código, y acortando aún más el camino entre el cliente y el proveedor. Además proporciona herramientas que ayudan a un desarrollo más rápido de aplicaciones, a una mayor facilidad de mantenimiento de la aplicación y, sobre todo, evita el uso de las plataformas semejantes por las dos partes que intervienen en el uso del servicio.

Si se piensa en una empresa de servicios, es posible que tenga un software realizado por módulos independientes, pero que a su vez necesiten llamarse entre ellos para formar un único programa. El problema es que los clientes quieren los programas hechos a medida. Si el programa está realizado usando programación orientada a objeto, las modificaciones pueden ser mínimas, pero aún así, obligamos al cliente a usar la tecnología utilizada por nuestros objetos; obligándole en muchos casos a usar unos programas cliente realizados en un lenguaje específico o incluso, un sistema operativo en concreto, porque, por ejemplo, no se pueden usar unas DLL (Dynamic link library, librerías de enlazado dinámico) desde un sistema Ultrix.

A partir de ahora se puede tener un servidor con la lógica empresarial codificada en Java, y usar clientes móviles realizados en Visual Basic, y una modificación en el servidor no implicará ningún cambio en el cliente. Todo este ahorro de tiempo permitirá invertir esfuerzos en otras áreas que no sean la implantación del producto, como mejoras del código de las rutinas o en el test del software. Además, debido a la rápida implantación, no será necesario parar los procesos durante largos periodos puesto que inmediatamente después de la publicación del servicio éste ya sería utilizable.

Un web service es una interfaz, accesible por protocolos (estándar o no) usados en Internet, que permite acceder a las funcionalidades de un objeto concreto, sin importar las tecnologías ni plataformas implicadas en la petición.

Un web service es una parte de lógica de negocio, capaz de procesar y accesible desde cualquier lugar, por cualquier persona (con permisos para ello), a través de cualquier medio. Más explícitamente, un web service es una interface hacia una aplicación o proceso accesible vía red informática mediante cualquier tipo de tecnología orientada a Internet, tales como FTP, HTTP, SMTP, Jabber, etc.

Si bien es cierto que los web services pueden ser accedidos usando múltiples protocolos, quizá sea el HTTP el más usado (Microsoft hace uso intensivo del protocolo de mensajes instantáneos), por la

facilidad de implementación, por ser interactivo y por el hecho de estar ampliamente extendido, por ello a lo largo del libro nos centraremos principalmente en él.

El web service se colocará entre el usuario y el código usado por éste, y se encargará de abstraer las especificaciones técnicas del programa que atenderá la llamada, para que cualquier lenguaje de programación que tenga soporte web service (realmente todo el que tenga posibilidad de trabajar con Internet), tenga acceso a nuestro programa. Esta abstracción nos permite usar los web services en transacciones B2B (Bussiness to Bussiness, negocio a negocio), B2C(Bussiness to Client, de negocio a cliente), C2B (cliente to Bussiness, de cliente a negocio), P2P (Peer to Peer, de igual a igual)..., pudiendo ser el elemento cliente un usuario humano, otro web service, o un programa.

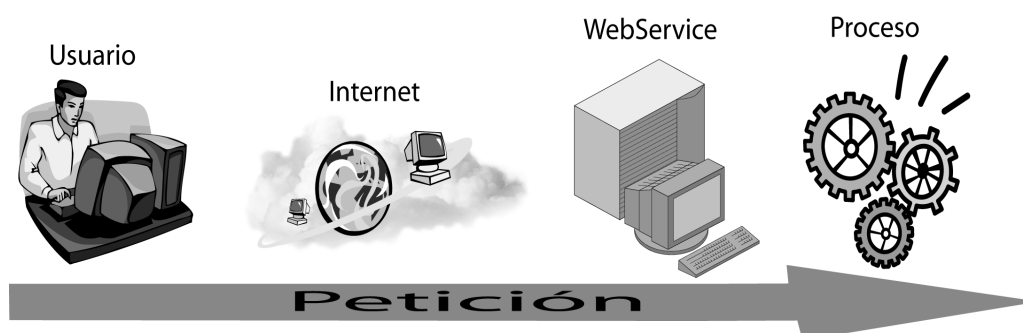


Figura 1: Esquema del web service.

Hay que pensar en el web service como cualquier otro tipo de objeto. No hay que creer que un web service sirve sólo para devolver la temperatura que hace en estos momentos en Chicago, o un código postal de una población. Si bien la mayoría de ejemplos que hay en la red son de este estilo, se verá más adelante que es posible usarlos para CUALQUIER aplicación, desde normalizaciones de datos, recuperación de datos, peticiones desatendidas, manejo de ficheros, etc... puesto que puede funcionar con cualquier tipo de datos, de manera atendida (interactiva) o desatendida (procesos batch o procesos por lotes), de manera síncrona o asíncrona...

Podemos pensar en el web service como una entidad (autónoma o acoplable) accesible desde cualquier nodo de Internet, capaz de reaccionar a peticiones del usuario, sin implicar ningún tipo de lenguaje de programación, ningún tipo de transporte concreto, ni ninguna restricción que se pueda dar en objetos convencionales, si bien hay que tener presente que trabaja mediante mensajes en modo texto, que deberán viajar por la red, y ser analizados, por lo que no se le puede pedir la misma velocidad que a objetos con llamadas nativas, y que estén alojados en la misma máquina que en la que se está ejecutando la aplicación.

La versatilidad y diversidad de modos de acceso al web service, permitirá que el servicio que atiende las peticiones, pueda ser desde pequeños objetos alojados en un servidor HTTP montado en un PDA, hasta Enterprise Java Beans funcionando en un gran servidor de aplicaciones de varios miles de euros.

Se podría decir que los web services son para el software como el Plug & Play para el hardware.

Por último, como ejemplo de lo útil que puede ser un web service, y de cómo puede dar ventaja sobre los competidores por su rápida integración, se presenta el siguiente ejemplo.

Suponga que tiene una empresa que se dedica a realizar un programa de diseño en tres dimensiones, pero la competencia tiene otro programa, que, como es lógico, guarda los datos en otro formato. Para poder importar o transformar los datos del formato de la competencia al nuestro, puede implementar unas rutinas en el programa que vende. Pero si en un momento dado la competencia cambia el formato (por cambiar por ejemplo de versión), los usuarios de su programa, deberán esperar a que se desarrolle una

nueva versión para poder transformar ficheros de la competencia en ficheros accesibles por su aplicación. Aquí podría entrar en juego un web service. Se podría desarrollar rápidamente un objeto al que se le pasa como parámetro un fichero del nuevo formato de la competencia, y devuelve un formato compatible con su programa, si ahora lo hace accesible transformándolo en un web service, su cliente sólo deberá hacer una petición a este web service con el fichero a transformar, y podrá obtener el fichero compatible con su aplicación. ¿No cree que este cliente estará contento con su compañía por haberle ofrecido de forma tan rápida una solución?

1.2.2. ¿Cómo se realiza todo esto?

Está claro que para poder mantener una coherencia entre datos recibidos y datos emitidos teniendo en cuenta que pueden ser dos sistemas totalmente distintos, se deben seguir unas normas tanto de codificación como de transporte, perfectamente definidas antes del comienzo de la transmisión. Todos los intercambios de mensajes se realizan de forma que sea compatible a todos los sistemas, mediante mensajes de tipo texto. Para hacer la petición a un servicio, se utiliza un mensaje, y la respuesta también se dará de la misma forma, mediante otro mensaje.

Así, con estas condiciones, se puede tratar Internet como un entorno de trabajo orientado a mensajes, no importando la localización física del destinatario, y lo que es más importante, tampoco debe importar la tecnología usada en el elemento que responda a la petición.

La comunicación entre los distintos elementos del entorno, se realizará utilizando un lenguaje común, éste deberá ser un estándar, como lo es XML (en siguiente capítulo se detalla dicho lenguaje).

En cuanto a los protocolos de red utilizados, no existe ninguna limitación física que prohíba usar alguno de ellos, si bien unos se adaptarán mejor que otros a las necesidades de cada aplicación en cada momento, lo que está claro es que cliente y servicio deben utilizar el mismo protocolo para el paso de mensajes, ya que si no, no se encontraría ningún tipo de respuesta.

A la hora de confeccionar el mensaje que se emitirá, hay que tener en cuenta el método de codificación de los datos, ya que el mensaje ha de ser de tipo texto, y no todos los datos utilizados en la aplicación son texto. Si se piensa que existen lenguajes para los cuales no existe el tipo *boolean*, o que a lo mejor el tipo *float* es de 32 bits, mientras que en otros es sólo de 16 bits, existe un problema a la hora de pasar datos entre sistemas heterogéneos. De alguna manera se debe controlar la transformación entre el dato del mensaje a transmitir o recibir y el código de la aplicación que lo gestione. Para ello está SOAP, en él siempre se representa todo en modo texto; cada una de las partes se encargará entonces de realizar la conversión hacia el dato que sea preciso en cada instante. Se verán ejemplos en capítulos posteriores.

1.2.3. Topología del web service

Ya se ha dicho que un entorno en el que se usan web services, es un entorno gestionado por mensajes, por lo que habrá acciones de envío y recibo de éstos.

En una acción de petición – respuesta de un web service, pueden intervenir desde una sola máquina hasta N, siendo N tan grande como el número de máquinas conectadas a la red. Esto es porque un web service no tiene por que ser invocado directamente, es decir, el servicio que atenderá la petición puede estar en nuestra propia máquina, como puede estar en Laponia; el mensaje viajará de máquina a máquina hasta llegar al destino. Además existen métodos de encaminamiento pudiendo obligar al mensaje a seguir caminos específicos.

Si hablamos de las máquinas que realizan acciones que afectan al funcionamiento del web service, podemos hablar de tres:

- 1.- la que realiza la búsqueda del servicio y petición del mismo
- 2.- la que responde a la búsqueda
- 3.- la que responde al servicio.

A medida que se avance en el contenido de la guía, se explicará la función y funcionamiento de cada una de ellas.

La arquitectura así formada es la siguiente:

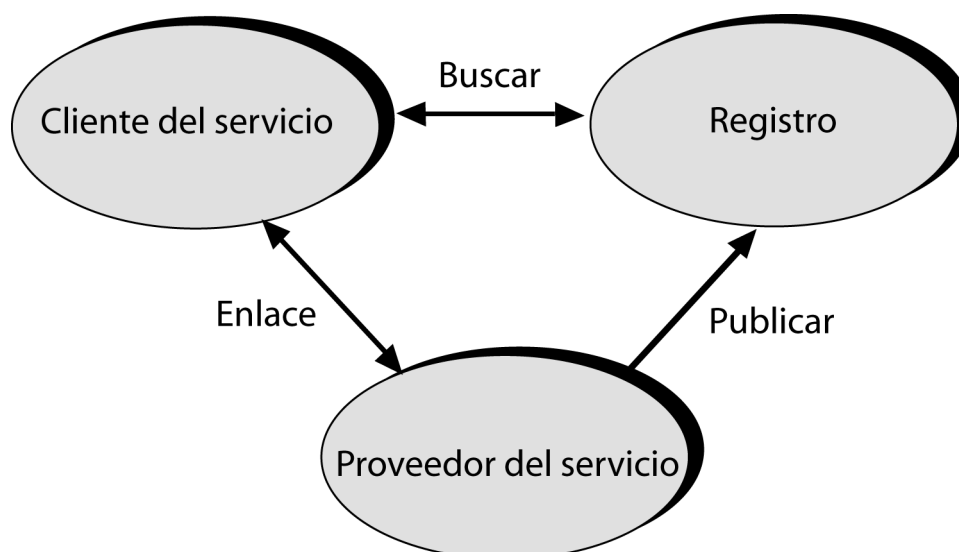


Figura 2: Topología de un sistema basado en web services.

Como puede suponer el lector, estas tres máquinas pueden ser en realidad la misma, sobre todo si se está en la fase de desarrollo del producto, más concretamente, en esta fase, no se usa registro alguno, ya que se conoce perfectamente donde está alojado el servicio (en normalmente en el propio ordenador donde se realiza el desarrollo) y cómo es su interfaz, pero se verá que en producción, no es que sea aconsejable la existencia de las tres, sino que es casi obligatorio.

Para hacerse una idea del funcionamiento de esta arquitectura, piense en lo que hace, por ejemplo, cuando necesita, un carpintero:

- 1.- Va a las páginas amarillas y busca por carpinteros.
- 2.- Obtiene una lista que procesa buscando el que más le convenga, bien por proximidad, bien por que le guste el nombre de la empresa o por lo que sea. Elige uno.
- 3.- Llama al carpintero para reclamar sus servicios.
- 4.- El carpintero le ofrece sus servicios.

A grandes rasgos, con los web services funciona de la misma manera:

- 1.- La máquina hace una petición al registro para conseguir una lista de web services que cumplen las condiciones de búsqueda proporcionadas.
- 2.- La lista que se obtiene, contiene información a partir de la cual es posible determinar la petición que más se acomode a las necesidades.
- 3.- Realiza la petición.
- 4.- Obtiene la respuesta (si la hubiera) por parte del servicio.

El funcionamiento no es tan sencillo como lo presentado anteriormente, pero es algo que iremos viendo a lo largo de los sucesivos capítulos del libro, donde se irán completando los aspectos que envuelven cada una de las peticiones.

Si se piensa en la cantidad de expectativas que debe cumplir y procesos que se deben realizar en la transmisión y proceso de un web service, está claro que el XML por si solo, no es suficiente para cumplir todos los requerimientos. Para asegurar que todas las transacciones que se dan entre las partes, se realizan de forma correcta, se necesitan otro tipo de tecnologías, en este caso, tecnologías basadas directamente en lenguaje XML.

Los lenguajes, protocolos o especificaciones usados principalmente serán:

- SOAP (Simple Object Access Protocol)
- WSDL (Web Services Description Language)
- UDDI (Universal Description Discovery and Integration) o WSIL (Web Service Inspection Language)
- WS-Routing, encaminamiento
- WS-Security, seguridad
- Coordinación y transacciones entre web services

1.2.3.1. SOAP

Como primera aproximación, se puede decir que SOAP se usará como plantilla para generar los mensajes entre cliente y web service, para codificar las peticiones y las respuestas, además marcará la manera en la que cliente y servicio hablarán entre ellos.

Va a ser el lenguaje utilizado para generar el mensaje que será enviado como petición y más tarde servirá para la generación de la respuesta (en el caso de que exista). No es la única manera de generar estos mensajes, pero si la más extendida y por consiguiente la que se explicará en el libro.

Para abrir un poco el apetito del lector presentamos lo que podría ser una respuesta SOAP a una petición del famoso programa Hola Mundo.

```
HTTP/1.1 200 OK
```

```
Content-Type: text/xml; charset=utf-8
```

```
Content-Length: 496
```

```
Date: Wed, 01 Jan 2002 19:09:14 GMT
```

```
Server: Apache Tomcat/4.0.6 (HTTP/1.1 Connector)
```

```
Set-Cookie: JSESSIONID=09A3D3E15FFC3D0C9FEE0CE5DF72FC4A;Path=/soap
```

```
<?xml version='1.0' encoding='UTF-8'?>
```

```
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<SOAP-ENV:Body>
```

```
<ns1:getHolaResponse xmlns:ns1="urn:holawebsevice" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

```
<return xsi:type="xsd:string">Hola Joan , soy tu servicio
web.</return>
```

```
</ns1:getHolaResponse>
```

```
</SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

1.2.3.2. WSDL

WSDL (Web Service Description Language, lenguaje de descripción del web service) es el documento que servirá para realizar la descripción funcional y técnica del web service.

En este documento se incluyen, entre otros datos, la declaración de los tipos y nombres de funciones del web service, y además servirá como anuncio publicitario para que otras personas lo puedan utilizar, y generar sus clientes de manera casi automática.

Al generar el WSDL, se puede indicar para qué sirve el servicio, cómo se usa, qué se debe esperar de él, con qué datos se llama, cómo se responde e incluso dónde se debe realizar la petición del servicio, es decir en que localización física se halla el objeto que atenderá la petición del usuario.

1.2.3.3. UDDI y WSIL

Se ha mencionado anteriormente que para buscar un web service, se acudía a una especie de páginas amarillas. Este registro es el llamado registro UDDI (Universal Description Discovery and Integration), y contiene información a escala mundial de los servicios prestados por cada una de las empresas registradas. UDDI será concretamente el protocolo usado para la identificación del WSDL del servicio que buscamos.

WSIL es una nueva forma distribuida de mantener listas de enlaces a direcciones en las que se halla presente un WSDL.

Tanto UDDI como WSIL se usarán para la publicación y localización de web services, la diferencia entre ellos está en la forma que tienen de almacenar los datos, ya que el UDDI es un registro centralizado y WSIL son listas distribuidas.

1.2.3.4. Encaminamiento

Pese a que en las primeras versiones de SOAP no se tuvo presente la importancia de un sistema de encaminamiento de mensajes, actualmente existen varias soluciones al problema que se puede presentar si se transmiten datos sensibles.

El hecho de enviar datos a través de Internet, los deja totalmente indefensos frente a ataques, no importa que se tenga una alta seguridad en los ordenadores de la empresa, si los ordenadores nodo por los que pase el mensaje hasta llegar a su destino, son vulnerables.

Es por lo que se pensó en poder marcar los ordenadores por los que se debe transmitir el mensaje, creando una red segura de nodos fiables, surgiendo así varias implementaciones para cubrir este pequeño “fallo” de SOAP.

Una de las más usadas es la de Microsoft, llamada Web service-Routing (WS-Routing). El funcionamiento es simple. En la cabecera se indicarán los ordenadores por los que deberá pasar el mensaje. Estas direcciones, normalmente correspondientes a ordenadores fiables, tienen que estar en el mismo orden que el que se quiere que siga el mensaje.

1.2.3.5. Seguridad

Los web services pueden actuar en Intranet o en Internet. Si son web services privados, es decir que sólo lo se usarán en una red interna, no hay que preocuparse tanto por la seguridad, ya que lo normal es que los empleados de una empresa no se dediquen a intentar extraer información de las conexiones. El panorama cambia cuando los servicios se exportan hacia Internet, ya que entonces, el mensaje se encuentra en un entorno “hostil”, en el que cualquier ordenador puede ser diana de algún ataque.

El lector no debe pensar que cualquier persona se colará en su sistema informático simplemente por ofrecer web services, ni mucho menos. Hay que recordar que al ser transmitido por protocolos estándar, se puede incluso utilizar el puerto 80 para atender las peticiones, por lo que no se tendrá ni que cambiar la configuración del cortafuegos (en caso de tenerlo).

Realmente la seguridad en los servicios, no está pensada para los pequeños o simples, sino más bien está pensada para transacciones en las que se necesite conocer la identidad de la persona que las está efectuando, o reconocer la veracidad de algunos datos para evitar usurpaciones de identidad, por ejemplo en una petición de crédito o transacciones monetarias.

Además, se podrán utilizar firmas digitales como PGP (Pretty Good Privacy, privacidad bastante buena) y otras basadas tanto en claves asimétricas como en simétricas.

Pero una vez más, las grandes empresas de la informática salen a la palestra con servicios de autenticación, como los tres más conocidos:

- Sun's liberty project de Sun Microsystems
- Magic carpet de AOL
- Passport de Microsoft

Es lógico pensar que estas tres empresas (y otras muchas que también dan servicios de este tipo) trabajan cada una independientemente, sin tener en cuenta la manera de trabajar del resto.

Debido a la necesidad de estándares para mantener compatibilidades entre web services, surge el sistema de firmas XML Digital Signature (firma digital XML) y el sistema de encriptación XML Encryption, esta vez sostenidos desde el World Wide Web Consortium.

1.2.3.6. Coordinación y transacciones entre web services

Pese a que es relativamente fácil hacer que dos servicios web se comuniquen, cuando son varios los que deben proceder a intercambiar datos, la situación puede complicarse. Además es posible que se quiera que los servicios se llamen en un orden determinado, y permitir deshacer tareas ya realizadas si alguna de las actividades que forman parte del proceso no se realiza de forma correcta (transacciones).

Mediante las especificaciones WS-Coordination y WS-Transaction, se ofrece la posibilidad al programador de realizar estas y otras muchas acciones que facilitan la interconexión de procesos basados en servicios web.

Además existe otro lenguaje llamado BPEL4WS, Business Process Execution Language for Web Services (Lenguaje de ejecución de procesos empresariales para servicios web), que permite definir procesos empresariales basados en web services, mediante una serie de actividades (como llamadas a servicios, esperas, etc) y unas etiquetas de control (como bifurcaciones, bucles, etc..) que ayudan a generar el flujo del proceso.

1.2.4 Empaquetado de datos

Aunque XML está pensado para trabajar con mensajes en modo texto (no binario), esto no impide que se utilicen datos de cualquier tipo, pero no sólo datos de tipo simple, sino que permite usar objetos creados por el propio programador, imágenes o archivos ejecutables. Esto es posible gracias al empaquetado de los datos, (también conocido como marshalling o serialización).

El empaquetado de datos se debe realizar a la hora de generar el mensaje a transmitir, y debe hacerse de manera que sea comprensible por todas las partes que intervengan en la transacción, y pese a que el estándar de empaquetado SOAP es una de las soluciones más ampliamente adoptada, no la es única, ni quizá la mejor, por lo que se tiene que fijar del tipo de empaquetado usado.

El empaquetado de datos es un aspecto muy importante, y se debe tener mucho cuidado en la manera en la que se hace, por que pese a que como hemos dicho, se trata de conseguir el mensaje en modo texto sin formato, se tiene que tener en cuenta que no todos los países tienen disponibles los mismos caracteres en su idioma. Los españoles tenemos letras que no están disponibles en la codificación UTF-8, por ejemplo nuestra apreciada "ñ" (¿alguien ha encontrado la ñ en un teclado americano?), o los signos "¿" y "¡" y las tildes (la diéresis no da problemas por ser usada en más idiomas), además los sistemas operativos no guardan el texto plano de la misma manera, como ejemplo, indicar que Windows lo almacena marcando cada final línea como carácter ASCII 10 + ASCII 13 (nueva línea + retorno de carro), mientras que los sistemas Unix lo hacen tan sólo mediante ASCII 10, otros sistemas usan EBCDIC (Código Extendido de Intercambio Decimal Codificado en Binario, de IBM) para representar caracteres, etc...

1.3 Futuro de los web services

Pese a que el lector pueda pensar que los web services tienen como objetivo hacer olvidar tecnologías como CORBA o COM+, no se puede pensar por ejemplo en comparar las velocidades de proceso que se dan a la hora de utilizar una y otra tecnología, ya que los servicios web son mucho mas lentos que cualquiera de ellas, debido sobre todo al empaquetado y desempaquetado de datos. Los servicios web no

están para suplantar estas tecnologías, sino para completarlas e integrarlas bajo mismos procesos. Está claro que ahora se tiene la herramienta “perfecta” (hasta que no se invente otra) para poder integrar sistemas distintos bajo un mismo proceso. La flexibilidad que esto reporta, da a suponer que el incremento del uso de los web services será importante en los próximos años.

Además teniendo en cuenta la rapidez de adaptación que proporcionan, pueden llegar a ser un elemento diferenciador de la calidad de un software y por lo tanto de la empresa que lo comercializa.

Al estar tan ligados los web services a otras tecnologías como XML, WSDL o UDDI, el futuro también dependerá de la evolución que sufran éstas, que teniendo en cuenta su juventud, no cabe duda que cambiarán, implementarán más funcionalidades y se les encontrarán nuevas utilidades.

Los web services son pues una herramienta potente para el desarrollo distribuido en plataformas heterogéneas, con un presente y un futuro prometedor por la facilidad de adaptación a las nuevas tecnologías informáticas que vayan surgiendo, así como a nuevos y viejos lenguajes de programación.

El lenguaje XML

2.1. Introducción

El lenguaje XML (eXtensible Markup Language, lenguaje extensible de etiquetas) es de vital importancia en los web services, además se está implantando como estándar en muchas aplicaciones por ser fácilmente interpretable e integrable. Esta es la razón por la que se debe conocer (aunque sea de manera superficial) cómo se utiliza y cuáles son sus posibilidades.

Este capítulo introducirá al lector al lenguaje XML, lenguaje sobre el que están basados los web services y que, como se verá más adelante, se utiliza de forma profusa. La intención no es convertir al lector en un experto modelador de documentos XML, sino crear una base de conocimiento sobre este lenguaje, ya que se utilizará bastante a lo largo del libro; además podrá servir para futuras consultas sobre sintaxis o principios de diseño de documentos XML.

El XML surgió como una derivación del lenguaje SGML (Standard Generalized Markup Language, Lenguaje generalizado estándar de marcas), también conocido como ISO 8879 y fue diseñado para ser utilizado como estándar en el ámbito internacional, para la definición de los contenidos y estructuras en documentos electrónicos. El lenguaje SGML es un lenguaje bastante complicado y extenso en cuanto normativa, por lo que se suelen usar derivaciones de éste como el HTML (Hyper Text Markup Language, lenguaje de marcas de hipertexto) o el XML.

El número de aplicaciones que utilizan documentos en formato XML ha ido creciendo a lo largo de este último año, utilizándose no sólo como mensajes entre aplicaciones, sino como archivos físicos. El XML se usa para acciones tan diversas como configuraciones de aplicaciones, plantillas de estilo o descripción de datos o procesos. Aplicaciones como el *OpenOffice* guardan todos sus documentos (Textos, presentaciones, gráficos...) como combinaciones de documentos XML (lo que les convierte el documentos fácilmente portables).

Para comprobar que la aplicación *OpenOffice* guarda sus archivos en XML, se debe renombrar un archivo guardado por ésta y darle la extensión zip (ya que está comprimido). Una vez descomprimido se podrá acceder a estos documentos XML

El XML se ha extendido de tal forma, que ha llegado incluso a las bases de datos. Por ejemplo, es posible recuperar consultas directamente en XML e incluso, existen bases de datos pensadas específicamente para XML como Xindice (www.xindice.org), haciéndolas especialmente útiles y versátiles para las tecnologías emergentes.

Una de las mayores ventajas de XML, es que permite realizar un intercambio eficaz de mensajes entre tecnologías distintas, no importa que cliente y servidor sean totalmente heterogéneos, ya que sólo con analizar correctamente (se debe saber el significado de cada elemento leído) el documento XML se tiene suficiente para actuar acorde con la especificación del mensaje: ejecutar algún programa, configurar una aplicación, etc...

Hay muchas razones por las que el lenguaje XML ha alcanzado una popularidad y un uso amplio en las tecnologías informáticas. Entre otras destacan:

- **Facilidad:** Es un lenguaje fácil de entender por los humanos y fácil de usar por programas informáticos, por lo que se puede incorporar rápidamente a cualquier tipo de aplicación.
- **Universal:** no importa el sistema operativo, o el lenguaje de programación usado, ya que es totalmente independiente de ellos, además con la posibilidad de añadir el tipo de codificación utilizado, es posible representar caracteres específicos de cualquier idioma, como puede ser la ñ o cualquier carácter cirílico.
- **Versátil:** permite representar cualquier tipo de dato de manera sencilla, además sirve tanto para datos simples como para datos jerárquicos o estructurados.
- **Texto plano:** Ya se había mencionado varias veces esta cualidad anteriormente. Al no ser binario, es totalmente independiente de la plataforma utilizada en su proceso, si bien puede

interpretarse de manera diferente dependiendo de la página de códigos cargada, pero ya veremos como solucionar esto.

- **Estructuración:** Es un documento jerárquico totalmente estructurado, formando árboles de datos, por lo que el proceso del documento es muy sencillo mediante programas informáticos. Esto permitirá la automatización de innumerables tareas, que anteriormente había que hacer manualmente.

2.2. ¿Qué es exactamente XML?

XML es el acrónimo de eXtensible Markup Language, o lenguaje extensible de marcas; nombre que describe perfectamente su funcionamiento; un poco más adelante, en este mismo capítulo se comprenderá por qué.

Este lenguaje se desarrolló en 1996 bajo los auspicios del consorcio WWW (www.w3C.org). Partiendo de la base del trabajo de Charles Goldbarb (IBM), surgiría el SGML, y de éste se deriva el lenguaje XML.

XML es un lenguaje de marcas (o etiquetas) basado en texto plano (sin formato), esto es, el significado o interpretación de cada parte de documento depende directamente de la posición que ocupan sus etiquetas. Para una primera aproximación podemos decir que XML es un lenguaje parecido en las formas al archiconocido HTML, ya que los dos son lenguajes de marcas y los dos derivan del mismo lenguaje estándar.

Llamaremos marcas, etiquetas o “tags” a los elementos encerrados entre los signos menor que y mayor que (< y >).

Analicemos este pequeño fragmento de código HTML:

```
<h1> el chico corrió por la calle </h1><br>
```

```
<b>La calle estaba oscura</b>
```

En el código se pueden diferenciar tres marcas distintas (<h1>
 y); también se puede ver que algunas de las marcas van por parejas, siendo una la etiqueta de apertura <h1> y siendo la de cierre </h1>. La marca de cierre es idéntica a la marca de apertura, exceptuando la barra “/”. La modificación de las etiquetas se extiende a todo lo que se halla encerrado entre ellas.

Otras, en cambio, no tienen etiqueta de cierre como en este ejemplo:
. Estos casos no se darán nunca en el lenguaje XML.

Existen algunas trabas en la generación de documentos XML para aquellos que han trabajado durante tiempo en documentos HTML, puesto que XML tiene restricciones que no existen en HTML, y que por lo tanto son motivo de error y se hacen difíciles de depurar:

- En XML las marcas **siempre** irán en parejas. En HTML existen marcas que son simples, como <image> o <hr>.
- En XML lo que indican las marcas, es **qué** significan los datos encerrados entre ellas, dan información de contenido, mientras que en el HTML las marcas indican **cómo** se han de representar los datos que contienen, por ejemplo cómo debe ser una palabra, dónde debe ir una imagen o un enlace.
- El XML es sensible a las mayúsculas (*case sensitive*), si se abre una etiqueta como <HoLa>, el cierre tiene que ser **exactamente** igual: </HoLa>, mientras que en HTML no es obligatorio, es posible tener un enlace de la forma index
- Las marcas deben estar perfectamente anidadas, no se puede cerrar una marca sin haber cerrado previamente todas las marcas que, directa o indirectamente, descienden de ella.
- En XML es obligatorio el uso de un elemento raíz, mientras que en HTML si no se encierra el documento entre las etiquetas <HTML></HTML>, no pasa nada.
- En XML es el programador el que da sentido a las etiquetas, en HTML las etiquetas vienen impuestas por el World Wide Web Consortium.

Las marcas de apertura y cierre en XML deben ser **exactamente** iguales incluso en la capitalización de la letra (mayúscula o minúscula) y mantener una anidación perfecta.

En XML es posible definir etiquetas propias, y con ellas crear sus propias estructuras y dar a cada una el significado que más le convenga, por esta razón recibe el nombre lenguaje extensible de marcas, ya que se pueden crear tantas marcas distintas como quiera, siempre que mantengan algún significado. Al trabajar con distintos documentos, se verá que marcas llamadas de la misma manera, pueden tener significados totalmente distintos o incluso en distintas partes del mismo documento, simplemente dependiendo de la colocación de las etiquetas. Esto es lo que hace al XML un lenguaje tan flexible en cuanto a posibilidades.

Un ejemplo de código XML podría ser un mensaje de llamada:

```
<llamada>

  <de>Sr. Lopez</de>

  <para>Sra. Garcia</para>

  <asunto>Nada especial</asunto>

  <mensaje>

    Llamarle a la oficina a las 5 de la tarde.

  </mensaje>

  <telefonos>

    <oficina>987654321</oficina>

    <casa>987123456</casa>

    <movil></movil>

  </telefonos>

  <recibido>8-10-2002</recibido>

</llamada>
```

Como pequeño apunte de la flexibilidad de XML, se añadirá un nuevo registro. Para ello solamente se necesita indicar un nuevo elemento raíz (en este caso *<agenda>*) y generar una nueva entrada. La agenda podría quedar de la forma:

```
<agenda>

  <llamada>

    <de>Sr. Lopez</de>

    <para>Sra. Garcia</para>

    <asunto>Nada especial</asunto>

    <mensaje>
```


Haga ahora una primera prueba de trabajar con XML. Para ello abra su editor de texto favorito, e introduzca el código mostrado anteriormente. Una vez introducido lo guárdelo con el nombre ejemplo2_1.xml, y abra nuevamente este fichero con el navegador web que tenga instalado, si todo ha ido bien y el navegador tiene plantillas para la visualización de XML se deberá ver algo semejante a:

Es posible que al principio se haga un poco engorroso el trabajar con este tipo de archivos usando un editor de texto normal sobre todo si el documento es extenso y con muchos tipos distintos de etiquetas. Para documentos XML complicados y sobre todo si tienen documento de validación (más adelante se verá esta característica), existen programas que asisten al usuario en la creación de los contenidos, auto completando textos, etc... Uno de los programas más conocidos de este tipo es XMLSpy, aunque es de pago. Otro editor muy cómodo y potente es el editor gratuito JEdit (incluido en el CDROM) (www.jedit.org) por permitir plegados de texto y creación de macros de validación, además incluye la posibilidad de incorporar un plugin de creación de documentos XML.

```

- <agenda>
- <llamada>
  <de>Sr. Lopez</de>
  <para>Sra. Garcia</para>
  <asunto>Nada especial</asunto>
  <mensaje>Llamarle a la oficina a las 5 de la tarde.</mensaje>
- <telefonos>
  <oficina>987654321</oficina>
  <casa>987123456</casa>
  <movil />
</telefonos>
<recibido>8-10-2002</recibido>
</llamada>
- <llamada>
  <de>Gerencia</de>
  <para>Sra. Garcia</para>
  <asunto>Estado del informe</asunto>
  <mensaje>El informe debe estar terminado esta noche.</mensaje>
- <telefonos>
  <oficina />
  <casa />
</telefonos>
<recibido>8-10-2002</recibido>
</llamada>
</agenda>

```

Figura 3: Navegador mostrando código XML

Alguno de los problemas que se han podido presentar durante la realización de este sencillo ejercicio son los siguientes:

- En XML, tal y como se dijo anteriormente, las marcas siempre deben ir en parejas, por lo que si, sin querer, se ha escrito <llamada>...</llamadas>, se obtendrá un error, ya que las etiquetas no son exactamente iguales. Si no se ha producido este error, invito al lector a provocarlo intencionadamente, para ello simplemente debe variar alguna de las etiquetas del ejemplo anteriormente realizado y volverlo a leer desde el navegador.
- En XML, las etiquetas son *case sensitive*, es decir que no es lo mismo escribir algo en mayúsculas que en minúsculas (al contrario que en HTML); si se ha escrito <llamada>...</llamaDa>, dará error por no ser las etiquetas **exactamente** iguales.
- En XML la anidación ha de ser perfectamente jerárquica. **No** es posible cerrar una marca de orden superior sin haber cerrado todas las marcas contenidas en ésta. Un ejemplo de código erróneo sería:

```

<Libro>
  <Capitulo1>
    <Subcapitulo>a</Subcapitulo>
    <Subcapitulo>b</Subcapitulo >
  </Capitulo1>
  <Capitulo2>
    <Subcapitulo>a</SubSubelemento1>
    <Subcapitulo>b</Capitulo2>
  </Subcapitulo>
</Libro>

```

Es posible ver que no se ha cerrado correctamente el elemento `</Subcapitulo>`, ya que se intenta cerrar `</Capitulo2>` antes de haber cerrado todos sus elementos hijos o descendientes.

Como se ha demostrado a lo largo de este punto, pese a que HTML y XML tienen algunas similitudes por ser los dos lenguajes de marcas y tener los mismos orígenes, difieren bastante en reglas y usos, y esto puede llevar a distintas confusiones y errores.

2.3. Contenidos del documento XML

El contenido de los bloques en un documento XML no tiene ningún significado asociado, es el analizador y el programa que lo usa quien, a partir de las etiquetas encontradas, le da un significado propio a cada dato.

Suponga el código XML siguiente:

```

<Entrada>
  <ID>HUIJ78</ID>
  <A>7</A>
  <L>12</L>
</Entrada>

```

Es posible tener dos aplicaciones totalmente distintas para las cuales un mismo documento sea un documento correcto sintáctico y estructuralmente, pero sin embargo, que sus significados sean totalmente distintos en cada una de ellas.

Por ejemplo, el código del listado anterior, puede ser de una aplicación que trabaje con elementos que sean piezas de papel geométricas, de las cuales se guarde un identificador (etiqueta ID), su ancho (etiqueta A) y su largo (etiqueta L), pero también puede ser de otra aplicación que guarde datos de personas, almacenando igualmente un identificador (etiqueta ID), su edad (etiqueta A) y un identificador de localidad de nacimiento (etiqueta L).

Los nombres de las etiquetas pueden ser elegidas por el programador, pero se debe tener en cuenta que estos nombres no pueden contener espacios, ni algunos caracteres como por ejemplo los signos de mayor y menor. Además, aunque algunos analizadores aceptan el uso de caracteres no admitidos en la codificación UTF-8, como la letra “ñ”, es mejor no utilizarlos para mantener compatibilidades.

Una vez visto cómo es la aplicación la que realmente termina de dar sentido a los documentos XML, se verá un poco de su morfología.

Un documento XML puede contener:

- Comentarios
- Elementos sin atributos con datos
- Elementos con atributos y datos
- Elementos con atributos sin datos
- Elementos vacíos
- Entidades

2.3.1. Comentarios

Como en todo lenguaje, en XML se tiene la posibilidad de insertar comentarios. El comentario tiene la misma estructura que en HTML, comenzando por los caracteres “<!--” y terminando por “-->”.

Por ejemplo:

```
<ejemplo>

  <datos> En el ejemplo se incluye un comentario</datos>

  <!-- este es el comentario incluido-->

</ejemplo>
```

El comentario puede introducirse en cualquier parte del cuerpo, y puede extenderse varias líneas, siempre que esté incluido entre las marcas <!-- y -->.

2.3.2. Elementos sin atributos con datos

Los elementos de este tipo sólo contienen datos entre las marcas de apertura y cierre.

Todo lo que se encuentra entre la etiqueta de apertura y la de cierre correspondiente, es tratado como un “todo”, y representa el valor correspondiente a la variable indicada por las marcas.

Por ejemplo:

```
<marca>

  Valor

</marca>

<marcaMasLarga>

  Todo este texto pertenece a la etiqueta marcaMasLarga.

</marcaMasLarga>
```

Esto sería como asignar:

```
marca = Valor
```

```
marcaMasLarga = "Todo este texto pertenece a la etiqueta
marcaMasLarga."
```

Pero la ventaja que se tiene con estas representaciones es que se pueden obtener estructuras más complejas como arreglos (arrays):

```
<array>
```



```

    <marca>
        Valor 1
    </marca>

    <marca>
        Valor 2
    </marca>

    <marca>
        Valor 3
    </marca>

</array>

```

2.3.3. Elementos con atributos y datos

Los elementos pueden contener información no sólo entre las etiquetas de apertura y cierre, como vimos en el punto anterior, sino también dentro de los caracteres de apertura y cierre por ejemplo:

```

<marca id="miMarca">
    datos
</marca>

```

Estas informaciones adicionales de las marcas se denominan atributos, y al igual que en HTML, sirven para completar la información del elemento.

El hecho de usar atributos o añadir nuevos elementos hijos para completar la información, será una elección del programador, y deberá ser él, el que en cada caso, elija entre las dos opciones siendo, normalmente, totalmente válidas ambas y ser solamente cuestión de estilos el decantarse por una de las soluciones o por la otra.

Los datos encerrados entre dos marcas con el mismo nombre, se consideran el valor de dichas marcas, no importa que existan saltos de línea o no. Incluso si se añaden varios espacios en blanco seguidos, solamente se tomará en cuenta uno.

La construcción:

```
<Elemento>12345</Elemento>
```

es idéntica a

```
< Elemento >
```

```
    12345
```

```
</ Elemento >
```

Los atributos, al igual que en HTML, se forman mediante un identificador, un signo igual y el valor asignado, pero a diferencia del HTML, el valor del atributo debe encerrarse **obligatoriamente** entre comillas tanto si contiene espacios como si no los contiene.

Si se retoca el ejemplo anterior de las llamadas para usar atributos y elementos, podría quedar algo parecido a:

```

<llamada de="Sr. Lopez" para="Sra. Garcia" asunto="Nada especial">
    <mensaje>

```



```

    <entrada mes="2" dia="3" hora="13:20"
        motivo="Revisión del estado de cuentas"
        lugar="Sala principal del edificio B" >

</entrada>

</agenda>

```

Las entradas del documento no tienen por qué estar contenidas en una sola línea, sino que se pueden extender las líneas necesarias como en el ejemplo anterior. Se puede dividir en tantas líneas como sea necesario para conseguir una buena visualización.

2.3.5. Elementos vacíos

En algunas ocasiones puede que no interese indicar ningún valor para alguno de los elementos.

Suponga que en el ejemplo de la llamada, el interesado no dejó mensaje con su teléfono móvil. Pero se puede dar el caso que sea obligatorio que el elemento `<movil>` esté presente en el documento, aunque no esté informado, es decir que la aplicación que vaya a interpretar este XML necesite el elemento aunque no contenga dato alguno.

En este caso se podría representar la información como se ha hecho hasta ahora :

```
<movil></movil>
```

o bien de modo abreviado:

```
<movil/>
```

El modo abreviado permite indicar el comienzo y cierre de elemento con tan sólo una marca, pero representando a ambos, por lo que cumple con la normativa XML. Además, aunque el elemento no contenga datos, sí se le pueden añadir atributos como en el siguiente ejemplo, donde se indica que no tiene conocimiento del número del móvil azul ni del verde:

```
<movil color="azul"></movil>
```

```
<movil color="rojo">678123321</movil>
```

```
<movil color="verde"/>
```

Los elementos vacíos pueden llegar a aportar información con su mera presencia, sería el caso de la etiqueta `
` en HTML, que pese a no tener más información que el propio nombre del elemento, su presencia modifica la interpretación del documento.

2.3.6. Entidades

Las entidades son unidades de almacenamiento, pudiendo contener desde un carácter hasta el nombre de un fichero para incluir en el XML.

Las entidades vienen dadas en la forma:

```
&entidad;
```

Por ejemplo, si se tiene que escribir en un documento XML varias veces la fecha en la que se acabe de editar el propio documento, se puede informar una fecha cualquiera y remplazarla al finalizar de escribirlo o se puede utilizar una entidad, y así sólo se deberá cambiar el valor de la entidad al finalizar el documento. Son como la representación de constantes de otros lenguajes, pero en XML.

Otro uso muy normal de las entidades, es la sustitución de cadenas de texto excesivamente largas para no repetir las a lo largo del texto, o caracteres extraños. En este libro, `&soap;` podría ser sustituido por "Simple Object Access Protocol", y así cada vez que se necesitara escribir esta cadena tan larga, valdría con escribir `&soap;`.

En HTML se hallan definidas por defecto entidades como `Ñ` para representar la "Ñ".

2.4. Cabeceras en XML

En realidad no existe una cabecera definida como tal, sino que la cabecera de un documento XML es la zona donde se realizan las primeras definiciones que afectarán al documento. En esta zona existen unas marcas especiales en las que es posible indicar aspectos tan importantes como la manera en la que se ha codificado la información, detallando la página de códigos usada y que se deberá tener en cuenta durante el análisis del XML. Otro aspecto muy importante que se informa en esta zona son los archivos de validación del documento.

La declaración del documento XML se realiza en la primera línea; en ella se listan aspectos como la versión de XML que se usará. Además, a la hora de analizar, permitirá rechazar archivos que no cumplen la versión esperada.

En la declaración del documento XML, (primera línea) es obligatorio indicar la versión de XML que se usará para la creación del mismo.

Las declaraciones de cabecera se representan como marcas, pero estas marcas son un poco distintas ya que por ejemplo no contienen etiqueta de cierre. Un ejemplo de cabecera sería:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

En la cabecera además de la versión de XML que se utilizará (que es obligatorio incluir), se pueden añadir otros atributos:

- **encoding:** opcional, por defecto se usa UTF-8, pero en el caso de los españoles, deberíamos usar ISO-8859-1 (o la ISO-8859-15) que corresponde al Latín 1, y así se tendría acceso a las tildes, las “ñ” e incluso a los signos de apertura de exclamación e interrogación.
- **standalone:** servirá para indicar si el documento hará referencias a otros documentos externos o no. En caso de estar él solamente, se deberá escribir este atributo con el valor “yes”. También es opcional.

Otras declaraciones que se pueden realizar son las de los ficheros de estilo que se usará en la representación del documento:

```
<?xml-stylesheet ref="simple-ie5.xsl" type="text/xsl" ?>
```

Todas estas marcas que comienzan con “<?” y terminan con “?>” son instrucciones que se utilizan para indicar que deben hacerse procesados a los datos. Se deben escribir estas etiquetas para reflejar el uso aplicaciones específicas para el tratamiento de los siguientes datos. El esquema de uso es:

```
<?aplicación atributo1="valor" atributo2="valor" ?>
```

Hay que aclarar algunos puntos sobre este tipo de marcas:

- NO puede haber espacios entre el carácter “<” y el carácter “?”
- Los datos a procesar comienzan tras el primer espacio en blanco encontrado dentro de la marca por lo que se debe escribir:

```
<?aplicacion atributo1="valor" atributo2="valor" ?>
```

y nunca

```
<? aplicacion atributo1="valor" atributo2="valor" ?>
```

- Dentro de los datos sí se pueden incluir espacios, siempre y cuando se respeten las normas de XML, es decir, el dato debe ir encerrado entre comillas.
- Es buen estilo añadir dos puntos tras el nombre de la aplicación que procesará los datos (ya que esta forma es aceptada por la especificación W3C para indicar el receptor de una instrucción),

pero debido a que el navegador *Internet Explorer* da error en muchas de sus versiones al encontrar este tipo de nomenclatura, no lo se usará en el libro.

Con lo visto hasta ahora ya es posible escribir el primer ejemplo usando castellano, mediante la página de códigos ISO-8859-1:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<llamada>
  <de>Señor López</de>
  <para>Señora García</para>
  <asunto>Nada especial</asunto>
  <mensaje>
    Lllamarle a la oficina a las 10 de la mañana.
  </mensaje>
  <telefonos>
    <oficina>987654321</oficina>
    <casa>987123456</casa>
    <movil></movil>
  </telefonos>
  <recibido>8-10-2002</recibido>
</llamada>
```

2.5. Validadores XML

Se ha visto que con XML se puede representar infinidad de datos, pero ¿son todos ellos válidos?, de alguna manera hay que indicar al sistema qué datos se esperan y en qué formato deben entregarse. Lo que se quiere indicar ahora NO es el significado, sino la estructura, la composición del documento, los nombres de los elementos y que elementos hijos pueden contener. Así es como se guía al usuario para que no use otras etiquetas distintas a las que espera el programa que se encargará de procesar el documento, y evitar de esta manera las ambigüedades o interpretaciones distintas.

Existen lenguajes basados en XML para la definición del documento (como el Document Structure Description, DSD) que son muy completos, incluyendo sentencias condicionales, estructuras, auto validaciones, etc.. pero en este libro no se tratarán.

El documento validador, será la forma en la que se el programador moldeará la estructura del documento XML que espera encontrar, indicando a su vez qué etiquetas puede contener y las relaciones que pueden existir entre ellas.

Para ver la importancia de este tipo de documentos se verán unos ejemplos explicativos. Suponga que se requiere representar una familia mediante XML. Si se fija en el ejemplo siguiente, verá que la familia elegida está formada por los abuelos, tres hijos, de los cuales dos están casados y unos nietos de una de las parejas.

```

<abuelos>
  <nombre sexo="V">Juan</nombre>
  <nombre sexo="M">Lourdes</nombre>
  <hijos>
    <nombre sexo="V" pareja="Teresa">Juan</nombre>
    <nombre sexo="M">Marta</nombre>
    <nombre sexo="M" pareja="Alberto">Laura</nombre>
  <nietos>
    <nombre sexo="V">Alejandro</nombre>
    <nombre sexo="M">Julia</nombre>
  </nietos>
</hijos>
</abuelos>

```

Pero si se fija ahora en el siguiente listado, verá que también es un XML totalmente correcto en cuanto a su sintaxis se refiere, pero tiene las siguientes irregularidades:

- Sabemos que no es posible que nuestros abuelos puedan ser nuestros hijos.
- La aplicación espera encontrar el atributo "sexo" solamente con los valores M para el femenino o V para el masculino, mientras que los datos recibidos no contienen lo esperado por parte de la aplicaciones que lo va a analizar.
- El abuelo tiene un atributo que es la edad, pero el resto no; es posible que la aplicación no sepa el significado de este atributo.
- El elemento de nombre Julia tiene una marca distinta al resto.

```

<abuelos>
  <nombre sexo="V" edad="89">Juan</nombre>
  <nombre sexo="M">Lourdes</nombre>
  <hijos>
    <nombre sexo="V" pareja="Teresa">Juan</nombre>
    <nombre sexo="M">Marta</nombre>
    <nombre sexo="M" pareja="">Laura</nombre>
  <abuelos>
    <nombre sexo="MASCULINO">Alejandro</nombre>

```

```

    <comoSeLlama sexo="M">Julia</comoSeLlama>

    </abuelos>

    </hijos>

</abuelos>

```

Estos y otros muchos problemas se pueden presentar a la hora de tratar la información del XML, y no sólo a nivel informático sino también a la persona humana que lo está informando. Este ejemplo es bastante intuitivo, pero si se trata de la configuración de una aplicación (por ejemplo), ni si quiera la lógica del ser humano sirve para conocer las dependencias entre los elementos..

De alguna manera se han de dictar normas sobre la estructura de la información que ha de contener el XML, para ello se utilizan unas plantillas que indicarán la forma de rellenar el XML, rigiendo nombres de marcas, hijos que pueden / deben tener, número de ellos, etc...

Su labor principalmente es:

- Indicar el nombre de cada elemento (Este dato ha de tener este nombre).
- Indicar la forma del documento (Este elemento puede tener este hijo, de este elemento tiene que tener al menos uno, de este otro puede tener de cero a siete, etc...).
- Indicar tipos de los datos (Este dato tiene que tener este tipo).

Existen varios lenguajes para definir dichas plantillas, pero sobre todo se utilizan los XML schema y los DTD. Principalmente el DTD es el más extendido para configuraciones y paso simple de mensajes, aunque ofrece más posibilidades el uso de XML Schemas, que es empleado en mensajes más complicados por tener mayor control sobre el tipo de dato, y por ser un lenguaje basado directamente en XML.

Por ser estos dos métodos los más importantes pasaremos a describir su sintaxis y aplicación.

2.5.1. El fichero de validación DTD

El documento DTD (Document Type Definition, definición de tipo de documento) es un documento realizado en texto plano, sin formato. Esto permite realizar este tipo de documentos usando cualquier editor de textos (siempre que se pueda guardar en texto sin formato), como el *edit*, el *Emacs* o el *JEdit*.

Pese a que el uso de DTD está prohibido en SOAP, se explicará para que el lector comprenda mejor lo que es un documento validador, y ayude en el siguiente punto a una mejor comprensión de la validación mediante XML Schema.

Este tipo de documentos han recibido críticas, porque pese a ser un lenguaje de marcas, no es un lenguaje basado en XML.

El DTD contiene la definición formal de un tipo de documento particular, éste define los nombres que pueden utilizarse en los elementos, dónde pueden aparecer y cómo se interrelacionan entre ellos.

Un ejemplo de la necesidad de tener alguna herramienta para indicar cómo debe ser el documento puede ser el siguiente. Suponga que posee un sistema automático de inserción de datos personales sobre los empleados de la empresa y el programa espera en algún momento que se le introduzca la dirección. El hecho es que podríamos expresarla de muchas formas:

```

<direccion>

    Plz. Las Batallas 5 bis 4º B

</direccion>

o

<direccion>

```

```

    <calle>
        Plz. Las Batallas
    </calle>
    <portal>5 bis</portal>
    <piso>4º</piso>
    <puerta>B</puerta>
</direccion>
0
<direccion>
    <calle tipo="Plz.">
        Las Batallas
    </calle>
    <portal>
        <numero>5</numero>
        <modificador>bis</modificador>
    </portal>
    <restoDomicilio>
        <escalera/>
        <piso>4º</piso>
        <puerta>B</puerta>
    </restoDomicilio>
</direccion>

```

¿Cual es el formato de dirección más adecuado? En realidad esto depende de la persona que programe, y del programa que tenga que tratar el documento XML, lo que está claro es que el formato óptimo será aquel que sea comprensible por todas las partes que actúen en la transacción.

En el DTD principalmente encontraremos las definiciones de:

- Elementos
- Atributos
- Entidades

2.5.1.1. Los elementos

Ya se ha visto que XML no tiene elementos propios, sino que es el programador el que los tiene que definir. Así pues para generar el XML lo primero es indicarle qué elementos puede o no puede contener, y el contenido que puede mostrar. Por ejemplo:

```
<!ELEMENT familia (abuelos | hijos)*>
```

La sintaxis de la declaración de elementos es:

```
<!ELEMENT nombreElemento (contenidoDelElemento)>
```

Los elementos se pueden clasificar de la siguiente manera:

- Elementos sin contenido
- Elementos conteniendo:
 - datos
 - otros elementos
 - modificadores

2.5.1.1.1. Elementos sin contenido

En HTML existen elementos sin contenido, dando toda su información a través de los atributos como por ejemplo el elemento imagen ``, pero existen otros elementos en los que ni si quiera es necesario presentar sus atributos para que tengan valor en el proceso, como sería el caso de las etiquetas `<hr>` o `<p>` en HTML.

En DTD la definición sería:

```
<!ELEMENT elemento_vacio (EMPTY)>
```

Este tipo de elementos proporciona su valor al documento simplemente con su presencia.

Recuerde que estamos hablando de XML y no de HTML, y en XML las marcas van siempre emparejadas, así si se escribiera el elemento nueva línea de HTML con el estilo XML sería:
`
`

2.5.1.1.2. Elementos conteniendo datos

Los datos contenidos pueden ser analizados (*parsed*) o no, declarándose como `#PCDATA` y `#CDATA` respectivamente o bien puede declararse tipo `ANY` si puede contener cualquiera de ellos.

Un ejemplo en HTML sería los elementos de estilo de letra como

```
<H1>Texto con formato.</H1>
```

En DTD la definición sería:

```
<!ELEMENT nombreElemento (#PCDATA)>
```

Un archivo XML que estuviera validado por el documento anterior, podría contener esta etiqueta recién definida:

```
<nombreElemento>
```

Datos correspondientes a la etiqueta

```
</nombreElemento>
```

2.5.1.1.3. Elementos conteniendo otros elementos

Para obtener niveles de dependencia jerárquica, se debe hacer que unos elementos contengan a otros. Así en uno de los ejemplos anteriores, en el que se mostraba una familia, queda claro que si no existen los abuelos no puede haber nietos, pero esto que es lógico para los humanos, no lo sabe un programa, por lo que se habrá que indicárselo de alguna forma. Volviendo a los símiles sobre HTML suponga que se define una tabla del siguiente modo:

```
<table>

  <tr>

    <td>1</td>

    <td>3</td>

  </tr>

  <tr>

    <td>2</td>

    <td>4</td>

  </tr>

</table>
```

Se ve que el elemento `<td>` depende directamente del elemento `<tr>`, y éste a su vez, del elemento `<table>`. Se supone que si no se define la etiqueta `<table>`, no se podría definir ni `<tr>` ni `<td>` (si no hay tabla, no puede haber ni filas ni columnas).

Para definir un elemento llamado `<nombreElemento>`, que contendrá a otro llamado `<elementoHijo>` (y sólo contendrá uno), se realiza de la siguiente manera:

```
<!ELEMENT nombreElemento (elementoHijo)>
```

Si el elemento va tener hijos con distinto nombre (uno de cada clase), la declaración de dicho elemento sería semejante a:

```
<!ELEMENT nombreElemento (elementoHijo, ElementoHijo2,...)>
```

donde los puntos suspensivos indican que se debe completar la lista hasta cubrir los nombres de los distintos elementos que vaya a contener.

En el caso de la tabla sería a grandes rasgos algo como (se estudia más adelante el significado del carácter “*”):

```
<!ELEMENT table (td)*>
```

```
<!ELEMENT td (tr)*>
```

```
<!ELEMENT tr (#PCDATA)>
```

Los elementos hijos pueden ser a su vez padres de otros elementos, pero para ello hay que definirlo así en el DTD, este es el caso del elemento `<td>` del ejemplo anterior.

Si se piensa en la representación modular de una dirección de correos, es posible descomponerla en calle, número y piso, pero todos ellos forman conjuntamente la dirección, es decir, pertenecen a la dirección.

Ejemplo:

```
<!ELEMENT Direccion (calle,numero,piso)>
```

Cuando se realiza una declaración como esta, en la que tiene varios elementos hijos separados por comas, éstos se deben declarar en el mismo orden. El ejemplo anterior al completo:

```
<!ELEMENT Direccion (calle,numero,piso)>
```

```
<!ELEMENT calle (#PCDATA)>
```

```
<!ELEMENT numero (#PCDATA)>
```

```
<!ELEMENT piso (#PCDATA)>
```

2.5.1.1.4. Elementos conteniendo modificadores

Mediante el documento DTD se puede no sólo indicar el contenido de cada uno de los elementos, sino también cuantos elementos contendrán. Este control no implica un número exacto de ocurrencias (en DTD) como veremos, pero el control que se proporciona es más que suficiente en la mayoría de las ocasiones que se presentarán.

Para variar el número de apariciones de un elemento, se hace uso de un carácter modificador tras su nombre, empleando un carácter distinto dependiendo del número de ocurrencias del elemento que se desean conseguir.

Por defecto, si un elemento aparece en un DTD sin modificador alguno, el número de ocurrencias será igual a uno y sólo uno.

Los modificadores válidos son:

- El carácter “?”

Este carácter permite indicar que el elemento aparecerá una o ninguna vez dentro del ámbito en el que se define.

Por ejemplo en una sociedad monogámica, se puede tener esposa o no, pero en caso de tenerla, sólo se tiene una. Su DTD:

```
<!ELEMENT varon (esposa?)>
```

```
<!ELEMENT esposa (EMPTY)>
```

y en el xml podría aparecer como:

```
<varon>
```

```
  <esposa/>
```

```
</varon>
```

o como:

```
<varon>
```

```
</varon>
```

Si se quisieran representar los sacramentos (se supone que alguno de ellos sólo se puede recibir una vez como mucho, aunque algunos puedan recibir el del matrimonio varias veces), se crearía un DTD como:

```
<!ELEMENT Sacramentos (Bautizo?, Confirmación?, Eucaristía*, Penitencia*, Matrimonio?, Orden?, Extremaunción?)>
```

- El carácter “+”.

Permite indicar que el elemento al que modifica debe aparecer una o más veces, pero al menos debe aparecer una vez. Un ejemplo podría ser el destinatario de un mail, que debe ser al menos una persona, pero pueden ser más:

```
<!ELEMENT destinoMail (cuentaCorreo+)>
```

Al igual que con el carácter “?” es posible usar listas de varios elementos a los que modificar. Por ejemplo:

```
<!ELEMENT libro (autor, capitulo, paginas)+>
```

- El carácter “*”.

Permite indicar cualquier número de ocurrencias del elemento. Un ejemplo serían los hijos que puede tener una persona. Las personas pueden tener decenas de hijos o ninguno.

```
<!ELEMENT papa (bebe*)>
```

Otros modificadores (uno de ellos ya utilizado) son:

- El carácter “,”

Se usa para separar elementos hijos dentro de las listas de definición. Ya se ha visto su aplicación en los ejemplos anteriores, y obliga a la aparición de los dos hijos situados a los lados de la “,” (dependiendo de los otros modificadores).

- El carácter “|”

Sirven para realizar una operación O (OR) lógico, con lo que se indica que en el documento se podrá añadir bien el elemento que está a la izquierda del “|” o bien el que está en la derecha, pero no los dos. Por ejemplo en el sexo de una persona:

```
<!ELEMENT sexo (masculino | femenino | indeterminado)>
```

Si una persona tiene sexo masculino, está claro que no lo tiene femenino. Son excluyentes entre sí, a no ser que se indique lo contrario. Por ejemplo:

```
<!ELEMENT descendientes (hijos | hijas)*>
```

Se pueden hacer combinaciones entre estos modificadores para conseguir los efectos que se necesite para sus propósitos, consiguiendo DTD tan complicados como lo requieran los documentos a los que deban validar.

Suponga que queremos realizar una aplicación que maneje correos electrónicos mediante XML. El DTD de un correo sería semejante a:

```
<!ELEMENT email (from, to+,subject,cc*,message,attach*)>
```

```
<!ELEMENT from (#PCDATA)>
```

```
<!ELEMENT to (#PCDATA)>
```

```
<!ELEMENT subject (#PCDATA)>
```

```
<!ELEMENT cc (#PCDATA)>
```

```
<!ELEMENT message (#PCDATA)>
```

```
<!ELEMENT attach (#PCDATA)>
```

Hay que tener cuidado con las construcciones que se realizan en los documentos, porque cuando se complican, pueden llevar a confusiones. Imagine que es el dueño de un restaurante en el que de primero hay solamente sopa (cosas de la economía), pero de segundo se da a escoger entre carne o pescado, en este caso se deben evitar construcciones como:

```
<!ELEMENT menu ((Sopita, Carne) | (Sopita | Pescado))>
```

Los problemas que se derivan de este tipo de construcciones se sufren unos en unos casos más que en otros, por ejemplo si se usa un analizador serie (más adelante en este capítulo, se verá como funciona) y analiza una parte de XML del estilo a:

```
<menu>

    <Sopita>

        De marisco

    </Sopita>

    <Carne>

        Hamburguesa

    </Carne>

</menu>
```

El analizador de XML necesita procesar hasta “Carne” para reconocer el caso en el que está. Es mucho más clara una declaración:

```
<!ELEMENT menu (Sopita, (Carne | Pescado))>
```

Para finalizar con este apartado, se verán unas pequeñas variaciones del DTD y los cambios que significan en el menú del restaurante. En el siguiente caso se podrá repetir una vez del segundo siempre que fuera pescado.

```
<!ELEMENT menu (Sopita, (Carne | Pescado+))>
```

También se podría optar por no tomar segundo, pero en caso de tomar, únicamente se podría pedir un plato:

```
<!ELEMENT menu (Sopita, (Carne | Pescado)?)>
```

Buffet libre:

```
<!ELEMENT menu (Sopita ,Carne , Pescado)*>
```

¿Podría el lector hacer la definición del DTD para que se pueda repetir varias veces de primer plato si y sólo si se elige pescado de segundo?

Como se ha podido observar, el uso de combinaciones de modificadores y paréntesis da mucho juego a la hora de crear reglas de validación, pero hay tener en cuenta que una estructura se puede representar de muchas formas, por lo que si alguna vez se ve obligado a usar combinaciones de modificaciones extrañas, quizá sea el momento de pensar en otra estructura distinta, ya que podría equivocarse con facilidad.

2.5.1.2. Atributos:

Los atributos son modificadores de los elementos, gracias a ellos es posible enriquecer o terminar de completar la información dada por éstos.

La declaración de los atributos se realiza mediante la marca ATTLIST, cuya sintaxis es:

```
<!ATTLIST nombreElemento nombreAtributo tipoAtributo modificadorValor>
```

Donde *nombreElemento* es el nombre del elemento al cual se le añadirá el atributo, *nombreAtributo* será el nombre del que se le dará al atributo dentro del elemento, *tipoAtributo* indicará el carácter del atributo (ver ejemplos) y *modificadorValor* permitirá fijar el comportamiento del valor del atributo.

Si se necesita más de un atributo para un mismo elemento, se definen uno tras otro, separándose unos de otros mediante espacios, tabuladores o nuevas líneas. Más adelante se realizarán unos ejemplos.

El *tipoAtributo* puede tomar los valores:

- CDATA : El valor es alfanumérico, y no se realizará análisis sobre él. Sería la manera de indicar cadenas texto.
- (dato|dato|..): el valor debe ser algún elemento de la enumeración, y sólo uno de ellos si no se especifica lo contrario con algún modificador de los vistos.
- ID: su contenido es único en todo el documento y será el valor que identifique al elemento de manera unívoca.
- IDREF: el valor va referido al identificador de otro elemento.
- IDREFS: el valor es una lista de IDREF, separados mediante espacios en blanco, tabuladores o nuevas líneas.
- NMTOKEN: el valor es un nombre XML válido.
- NMTOKENS: el valor es una lista de NMTOKEN, separados mediante espacios en blanco, tabuladores o nuevas líneas.
- ENTITY: el contenido es una entidad definida previamente en el DTD.
- ENTITIES: el valor es una lista de ENTITY, separados los valores mediante espacios en blanco, tabuladores o nuevas líneas.
- NOTATION: el contenido es un nombre de una notación, que describe datos que no tienen formato XML.
- XML: el valor está predefinido

Como modificador del comportamiento del valor, se puede utilizar uno cualquiera de los valores siguientes:

- #DEFAULT valorPorDefecto: El atributo tiene de valor por defecto, el indicado en la declaración.
- #REQUIRED: El valor es obligatorio, debe aparecer en el documento.
- #IMPLIED: el valor está implícito, si no se especifica nada en el documento, se usará un valor definido en la aplicación.
- #FIXED valor: el valor es fijo, y no se puede cambiar en el documento.

Algunos ejemplos del uso de estos atributos:

```
<!ELEMENT vehiculo (conductor+,motor) >
<!ATTLIST motor
    id      IDREF #IMPLIED
    manufac CDATA  #FIXED
    combustible (gasolina95 |
                gasolina97 |
                diesel) #IMPLIED
>
<!ELEMENT conductor (persona)*>
<!ELEMENT persona (dirección?,telefono+) >
```

```
<!ATTLIST persona
    id      CDATA #IMPLIED
    nombre  CDATA #REQUIRED
    cargo   (jefe | trabajador | becario) #IMPLIED
>

<!ELEMENT dirección (#PCDATA)>
```

2.5.1.3. Entidades

Las entidades servirán, entre otras cosas, para abreviar textos a la hora de escribir el XML.

Las entidades se son referenciadas mediante *&nombreEntidad;*. Este tipo de sintaxis puede existir también en HTML por ejemplo para representar la letra ñ cuando no está definida la codificación latina, su código es: *ñ*.

Así, es posible crear una entidad llamada *Entidad* que tenga referencia a toda una frase, de manera que simplemente escribiendo *&Entidad;*, ésta “se transformará” en toda la frase.

Para crear nuevas entidades propias, se usa la marca ENTITY con la siguiente sintaxis:

```
<!ENTITY nombreEntidad valorAsociado>
```

En el caso que se quiera hacer un documento sobre Don Quijote de la Mancha, podrían definirse las entidades siguientes:

```
<!ENTITY titulo "Don Quijote de la Mancha">
```

```
<!ENTITY autor "M. De Cervantes">
```

Ahora, en el documento XML se puede hacer referencia a ellas mediante:

```
<sinopsis>
```

```
    El libro a estudiar fue &titulo;, que fue escrito por &autor;.
```

```
    &autor; nació en 1547 y el &titulo; lo escribió en 1605.
```

```
</sinopsis >
```

Las declaraciones de entidades pueden realizarse externamente al DTD, haciendo referencia a los archivos que los contienen mediante una URL:

```
<!ENTITY nombre SYSTEM URL>
```

como por ejemplo:

```
<!ENTITY bestSeller SYSTEM "http://www.acme.com/best.dtd">
```

2.5.1.4. Relación entre DTD y documento XML

Una vez que se conoce cómo se determinan las reglas que servirán como plantilla en la generación de los documentos XML, habrá que relacionar estas reglas con el documento XML.

La declaración de los elementos, entidades y atributos se puede realizar en el mismo fichero en el que se guarda la definición del documento XML o en un fichero DTD aparte, que como se dijo anteriormente, puede estar incluso alojado en otro servidor. La extensión de los archivos DTD es, como no podría ser de otra forma “dtd”.

Cuando la declaración de las reglas se realiza en el mismo documento se deben colocar al principio de éste, indicando a su vez el elemento raíz.

La declaración comenzará con la marca DOCTYPE como por ejemplo:

```
<!DOCTYPE elementoRaiz [
<!--El resto de las declaraciones irán aquí -->
]>
```

Si el se opta por guardarlo en un fichero distinto, se deberá hacer una referencia desde el fichero que contenga el código XML, incluyendo al principio de dicho archivo una línea del tipo:

```
<!DOCTYPE elementoRaiz SYSTEM "nombreFichero.dtd">
```

Como ejemplo de DTD y XML en un mismo fichero (DTD interno), se muestra el siguiente código, que el lector puede probar como hizo anteriormente, generando un fichero de texto plano con este contenido, y leyéndolo desde el navegador web:

```
<?xml version="1.0"?>
<!DOCTYPE email [
  <!ELEMENT email (from, to+,subject,cc*,message,attach*)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT subject (#PCDATA)>
  <!ELEMENT cc (#PCDATA)>
  <!ELEMENT message (#PCDATA)>
  <!ELEMENT attach (#PCDATA)>
]>
<email>
  <from>Marcos</from>
  <to>Manuel</to>
  <subject>Es un mensaje de prueba</subject>
  <cc>Antonio</cc>
  <cc>Carlos</cc>
  <message>El cuerpo del mensaje</message>
</email>
```


Si se utilizara un documento validador DTD externo (llamado en este caso email.dtd), el código del fichero XML sería:

```
<?xml version="1.0"?>

<!DOCTYPE email SYSTEM "email.dtd">

<email>

  <from>Marcos</from>

  <to>Manuel</to>

  <subject>Es un mensaje de prueba</subject>

  <cc>Antonio</cc>

  <cc>Carlos</cc>

  <message>El cuerpo del mensaje</message>

</email>
```

2.5.1.5. DTD AVANZADO

Debido a que no existe forma alguna de definir partes condicionales en el XML, se pueden definir secciones condicionales dentro del DTD, que serán controladas mediante las palabras clave "INCLUDE" e "IGNORE".

```
<![ INCLUDE [

    <!--definiciones que se tomarán en cuenta -->

]]>

<![ IGNORE [

    <!-- definiciones que no se tomarán en cuenta -->

]]>

<!-- definiciones comunes-->
```

Se pueden tener "comentadas" partes de un DTD, partes que simplemente cambiando la palabra clave que precede al bloque, se tendrán en cuenta sus definiciones o no. Aunque este método puede ser flexible, está claro que es pesado recorrerse todo el DTD cambiando las palabras dependiendo del XML que se vaya a tratar; por esto mismo se van a usar parámetros en el DTD, de forma que se tendrá un control del contenido del XML.

En el documento DTD pueden definirse parámetros mediante la sintaxis *%nombreParametro*; permitiendo darle el valor correspondiente en otro lugar del documento, bien del DTD o del XML

Si se incluyen unas referencias a parámetros, el DTD quedaría:

```
<![ %PRIMERO; [

    <!--definiciones que se tomarán en cuenta-->

]]>
```

```
<![ %SEGUNDO; [
    <!-- definiciones que se tomarán en cuenta -->
]]>
```

```
<!-- definiciones comunes-->
```

En el XML es donde se darán los valores a estos parámetros definidos. Para incluir el primer bloque y excluir el segundo se escribiría:

```
<!DOCTYPE miXML SYSTEM "dtdConParametros.dtd" [
    <!ENTITY % PRIMERO "INCLUDE">
    <!ENTITY % SEGUNDO "IGNORE">
]>
```

```
<miXML>
```

```
<!--definiciones del XML-->
```

```
</miXML>
```

Se habrá dado cuenta el lector, que los valores de la parametrización, se hacen definiendo entidades dentro del XML. Esto permite hacer plantillas muy potentes, por ejemplo se puede definir en el documento DTD:

```
<!ENTITY titulo "El libro tiene por título %TITULO">
<!--otras definiciones -->
```

Y en el XML definir el nombre del título para poderlo usar más adelante en forma de entidad:

```
<!DOCTYPE libro SYSTEM "libro.dtd" [
    <!ENTITY % TITULO "El Quijote de la Mancha">
]>
```

```
<libro>
```

```
<sinopsis>
```

```
    Título: &TITULO;
```

```
<sinopsis>
```

```
</libro>
```

2.5.2. El XML Schema

El XML Schema es muy potente, pero también es muy extenso en cuanto a tipos, normas y distintos usos de las mismas (por ejemplo los *patterns*), y su total descripción se escapa al alcance de este libro ya que para una explicación detallada sería necesario un libro entero sobre este tema, pero dada su importancia, se explicará de forma que al lector le pueda servir para poder interpretar documentos de este tipo que encontrará a lo largo del libro.

Para más información: <http://www.w3.org/TR/xmlschema-0/>

En muchas ocasiones el documento DTD no cubre las expectativas para la realización de la validación del documento XML, bien por ser éste extremadamente complejo o por violar la restricción del uso de documentos con lenguaje XML (El DTD no es un documento XML).

En este caso se podría usar para la validación, los esquemas XML (XML Schemas), estos ficheros se presentan con la extensión xsd. El XML Schema presenta varias ventajas respecto al DTD:

- Sintaxis en modo XML
- Más de treinta y siete tipos básicos de datos soportados
- Reutilización de nombres
- Herencias
- Definición de datos no necesariamente ordenada
- Restricciones y extensiones de los tipos de datos
- Mayor control del número de ocurrencias de elementos que deben aparecer
- Posibilidad de uso de datos complejos definidos por el usuario

Como primer acercamiento a las diferencias existentes entre el XML Schema y el DTD puede comparar estos ejemplos de un catálogo de libros. El primero se trata de un documento DTD y el segundo ejemplo es un XML Schema:

```
<!--Catálogo de libros DTD-->

<!ELEMENT CatalogoLibros (Libro)*>

<!-- Libro = Título+Autor+Fecha+ISBN+Editor -->

<!ELEMENT Libro (Titulo, Autor, Fecha, ISBN)>

<!ELEMENT Titulo (#PCDATA)>

<!ELEMENT Autor (#PCDATA)>

<!ELEMENT Fecha (#PCDATA)>

<!ELEMENT ISBN (#PCDATA)>

<!ELEMENT Editor (#PCDATA)>
```

Listado del documento DTD para la ficha de un libro

```
<?Xml version "1.0"?>

<!--Catálogo de libros XML Schema-->

<Schema xmlns="http://www.w3.org/2000/10/ XMLSchema"/>

<Element name="CatalogoLibros">

  <Type>
```

```

<Element name="Libro" minOccurs="0"
    maxOccurs="*">
    <Type>
        <Element name="titulo" type="string"/>
        <Element name="autor" type="string"/>
        <Element name="fecha" type="string"/>
        <Element name="ISBN" type="string"/>
        <Element name="editor" type="string"/>
    </Type>
</Element>
</Type>
</Element>
</Schema>

```

Listado del documento XML Schema para la ficha de un libro

Hasta aquí, por lo visto en los ejemplos anteriores, no parece que el XML Schema presente ninguna ventaja respecto al DTD. Pero se realizarán algunas modificaciones para ver que realmente se tiene mayor control sobre los datos y sus formas.

Es posible definir que la fecha en lugar de ser una cadena, sea realmente un tipo fecha, con lo que lleva consigo (no existen meses de ochenta días):

```

<element name="Date" type="string"/>
por
<element name="Date" type="date"/>

```

El control sobre el número de ocurrencias de un elemento ahora es total y no como en el DTD que mantenía un control muy pobre en este área, por ejemplo para que se sitúe entre dos valores que se quieran:

```

<Element name="Libro" minOccurs="0" maxOccurs="*">
<Element name="diasDeLaSemanaTrabajados" minOccurs="0" maxOccurs="7">

```

O controlar mejor la forma que tiene el código ISBN. El código ISBN es un código que identifica de forma única a una publicación. Suponiendo el ISBN de la forma XX-XXX-XXXX-L siendo las X cualquier número y la L cualquier letra mayúscula, se va a permitir en el siguiente ejemplo, que el XML pueda tener los ISBN en la forma "XX-XXX-XXXX-L" o como "ISBN XX-XXX-XXXX-L". Un XML Schema válido en este caso, sería:

```

<simpleType name="ISBNType">
    <restriction base="string">

```

```

    <pattern value="\d{2}-\d{3}-\d{4}-\p{Lu}" />
    <pattern value="ISBN\s\d{2}-\d{3}-\d{4}-\p{Lu}" />
  </restriction>

</simpleType>

<element name="ISBN" type=" ISBNType " />

```

Aunque sea con este pequeño ejemplo ya es posible ver que realmente se tiene un mayor control sobre el tipo de los datos mediante XML Schema que mediante DTD.

Además, con el uso de este tipo de validación, se realiza una comprobación de dos etapas, esto es, al ser el XML Schema un XML, es posible validarlo también contra su documento validador, cosa que no se podía hacer cuando se utilizaba DTD.

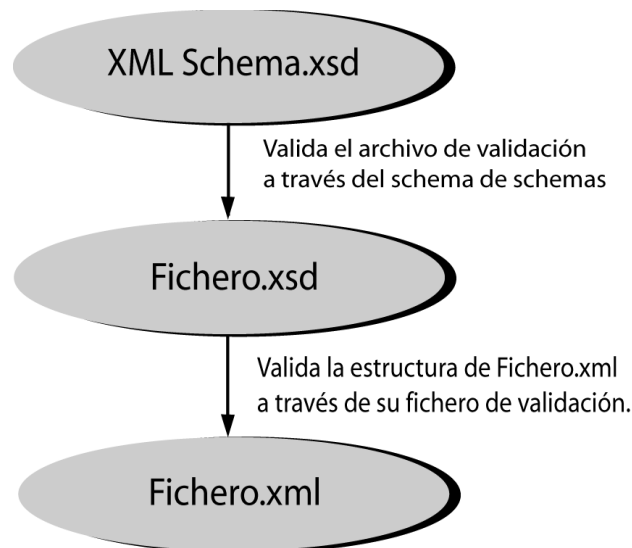


Figura 4: Esquema de validación de un XMLSchema.

En las siguientes secciones se verá un poco más en profundidad todo lo que puede ofrecer este otro método de validación de documentos.

2.5.2.1. Tipos de datos

Principalmente es posible realizar la división de los tipos de datos en simples y complejos, atendiendo a la forma que presentan. Los simples no llevan atributos ni subelementos que modifiquen su estructura, mientras que en los complejos aparecerán toda clase de variaciones, permitiéndose definir casi cualquier tipo de dato.

2.5.2.1.1. Tipos de datos simples

Los tipos de datos simples se diferencian de los complejos por no contener ningún atributo o subelementos que varíe su comportamiento.

Dentro de los datos simples se puede realizar una nueva división en varias categorías como fechas, numéricas, etc..

Como datos con tipo referente al tiempo:

- time (22:30:15.925)

- timeDuration (P3Y4M5DT20H)
- timeInstant (2002-04-05T15:20:45)
- date (2002-12-06)
- month (2002-12)
- year (2002)
- century (21)
- recurringDate (--30-04)
- recurringDay (---15)

Como se puede observar, la cantidad de tipos de datos referentes al tiempo, permiten un gran control sobre éste, pudiendo especificar en cada caso el tipo que más se ajuste a las necesidades del programa. Respecto a los números también ofrece distintas opciones para realizar su plantilla, siendo en algunos casos, pequeñas diferencias entre tipos:

- decimal
- integer
- float
- double
- integer
- non-negative-integer
- positive-integer

Otros tipos simples son :

- string (mi mama me mima)
- boolean (true)
- uri-reference (<http://www.gui.uva.es/>)
- ...

Para más información sobre los tipos disponibles en XML Schema, se puede consultar la dirección: <http://www.w3.org/TR/xmlschema-2/>

La potencia de los tipos simples no queda solamente reducida a lo que ofrece el XML Schema por defecto, sino que el usuario puede hacer nuevas restricciones y extensiones para que se adapte mejor a cada necesidad. En el ejemplo acerca del ISBN se vio ya algo referente a las restricciones en el formato, pero es posible añadir más restricciones y más características.

Con lo que respecta a los números, permite controlar aspectos como valores entre los que deben estar, precisión que debe tener o escalados a aplicar.

Para un XML Schema de un equipo de fútbol:

```
<simpleType name="dorsal">
  <restriction base="integer">
    <minInclusive value="1">
    <maxExclusive value="12">
  </restriction>
</simpleType>
```

Para los días del mes:

```
<simpleType name="diasDelMes">
  <restriction base="integer">
    <minInclusive value="28">
    <maxInclusive value="31">
  </restriction>
</simpleType>
```

Otra herramienta muy potente para la realización de reglas en los tipos de datos, es el uso de patrones (*patterns*). Antes de nada, un pequeño ejemplo al respecto.

Una población se compone (según el INE) de cinco partes diferenciadas, que son: provincia (dos caracteres), municipio (tres caracteres), código de entidad colectiva (dos caracteres), código de entidad singular (dos caracteres), y núcleo (dos caracteres) siendo un total de once caracteres. Además, para este ejemplo, se separarán mediante guiones por lo que habrá un total de quince caracteres por población. El XML Schema en este caso sería:

```
<simpleType name="poblacion">
  <restriction base="string">
    <pattern value="\d{2}-\d{3}-\d{2}-\d{2}-\d{2}" />
    <length value=15/>
  </restriction>
</simpleType>
```

...

```
<Valladolid type="xsd:poblacion">
  47-186-00-01-01
</Valladolid>
```

Los *patterns* se usan con cadenas de texto y permiten dejar claramente identificada la morfología de ella. En el ejemplo de generación del ISBN se ha usado ya un *pattern*, pero era muy sencillo y quedaban algunos aspectos sueltos, por ejemplo no había ninguna regla que prohibiera al usuario meter cualquier número en cada uno de los apartados, y hay algunas restricciones en el uso de ciertos números para la generación de ISBN (depende del país en el que se realice el ISBN).

Probemos algún *pattern* más complicado para ver el potencial de estos:

```
<xsd:pattern value="0-[2-6][0-9]{2}\d{5}-[0-9x]">
```

En el *pattern* anterior se especifican cadenas cuyo primer carácter es un cero, tras él un guión, luego un número que va desde el doscientos hasta el seiscientos noventa y nueve, seguido de cinco números, un guión y un número de 0 a nueve o bien una "x".

Como quizá no esté claro la parte en la que obligamos que el número sea entre doscientos hasta el seiscientos noventa y nueve, lo se explicará un poco más. La parte [2-6] indica que es un número entre dos y seis, y [0-9] indica que el número debe ser entre cero o nueve. Este último apartado está además modificado por {2}, indicando que son dos caracteres.

El siguiente *pattern* sería el de la generación de ISBN en España. En este *pattern* se especifica que los primeros caracteres son "84" seguido de un espacio (el espacio se especifica mediante \s), un dígito, y

luego una secuencia de seis caracteres que pueden ser un dígito decimal o un espacio. Tras esta secuencia, se espera un dígito decimal, un espacio y terminamos con un dígito o una “x”.

```
<xsd:pattern value="84\s\d([0-9]|\s){6}\d\s[0-9x]">
```

Un valor ISBN válido para el pattern anterior es:

```
84 1232 328 x
```

Los caracteres que modifican las repeticiones son los mismos y con el mismo significado que los vistos durante la explicación del DTD:

- * : cero o más
- + : uno o más
- ? : cero o uno

Para indicar intervalos numéricos de modificación de elementos, se usan las llaves “{}”, dentro de las cuales se especifica el número de repeticiones:

- {X}
Exactamente se repetirá X veces. Ej. \d{4} → 1234
- {X,}
El elemento se repetirá un mínimo de X veces, no importando el máximo. Ej.: \d{4,} → 12345
- {X,Y}
El elemento se deberá repetir de X a Y veces. Ej.: \d{4,7} → 123456

Si no se especifica ningún número de repetición entonces su número de ocurrencias es uno y solamente uno.

Los patterns se pueden complicar hasta niveles insospechados. ¿Puede el lector distinguir las partes y el funcionamiento del siguiente pattern?

```
<pattern value="x{0,8}*[a-h]+(\d\s)?">
```

Listas:

Son colecciones de datos, semejantes a los arrays. Se puede hacer listas por ejemplo para representar una alineación en un equipo de baloncesto:

```
<simpleType name="alineacion">
  <list base="string"/>
  <length value="5"/>
</simpleType>
```

Puede ser interesante que el valor introducido en el documento XML sea el perteneciente a una lista determinada, para ello es posible definirla como un nuevo tipo simple:

```
<simpleType name="reyesMagos" type="string">
  <enumeration value="Melchor"/>
  <enumeration value="Gaspar"/>
  <enumeration value="Baltasar"/>
</SimpleType>
```

Ahora podría el lector dirigir la carta a su rey mago favorito:


```
<!-- Carta a los reyes -->

<de>...</de>

<para type="reyesMagos"> Melchor </para>

<!-- resto de la carta -->
```

Además, a todos estos tipos se les puede dar un valor por defecto. Por ejemplo:

```
<element name="provincia" type="string" default = "Valladolid">
incluso podemos obligar a que tengan un valor fijo:
<element name="diasSemana" type="integer" fixed = "7"/>
<element name="descubridor" type="string" fixed = "Cristobal"/>
```

2.5.2.1.2. Tipos de datos complejos

Aunque los tipos simples pueden cubrir gran parte de las necesidades que no estaban cubiertas por los tipos predefinidos en el XML Schema, se pueden usar los tipos complejos para todas aquellas necesidades para las que las otras opciones son insuficientes. Por ejemplo, generar nuevos tipos a partir de las definiciones de tipos simples, como fechas.

Elementos ordenados

Cuando interesa que unos elementos aparezcan en un orden determinado, lo más común es utilizar un nuevo tipo creado por el usuario, que se ajuste perfectamente a las necesidades. Por ejemplo, cada país tiene una forma de indicar la fecha, se podrá entonces arreglar las incoherencias mediante el uso de un nuevo tipo creado por el programador, en el que se definen los elementos y el orden en el que tienen que aparecer.

```
<complexType name="mifecha">
  <sequence>
    <element name="dia" type="string">
    <element name="mes" type="string">
    <element name="anyo" type="string">
  </sequence>
</complexType >
```

Se pueden usar los tipos de fecha definidos por el XML Schema para definir uno más complejo, usando tipos como *year* o *month* para generar la fecha completa, incluso se permite obligar el uso del calendario gregoriano:

```
<complexType name="mifecha">
  <sequence>
    <element name="dia" type="gDay">
    <element name="mes" type="gMonth">
    <element name="anyo" type="gYear">
```

```

    </sequence>
</complexType >

```

Otra opción sería usar tipos propios para la generación de estos tipos compuestos:

```

<simpleType name="dia">
    <restriction base="integer">
        <minInclusive value="1">
            <maxInclusive value="31">
                </restriction>
    </simpleType>
<simpleType name="mes">
    <restriction base="integer">
        <minInclusive value="1">
            <maxInclusive value="12">
                </restriction>
    </simpleType>
<simpleType name="anyo">
    <restriction base="integer">
        <minInclusive value="2000">
            <maxInclusive value="2005">
                </restriction>
    </simpleType>

```

Atributos

Al igual que en el DTD, se pueden añadir nuevos atributos a los elementos, pudiendo así extender y completar el significado y la información.

```

<complexType name="persona">
    <sequence>
        <element name="nombre" type="string" />
        <element name="papellido" type="string" />
        <element name="sapellido" type="string" />
    </sequence>

```

```

</sequence>

<attribute name="edad" type="integer" use="required"/>

</complexType >

```

En este caso se ha definido un atributo que indicará la edad de esta persona, que además como se puede observar en su atributo "use", es obligatoria su presencia. Los distintos valores que puede tomar este atributo son:

- required
- optional
- prohibited

No se puede definir nunca un atributo tipo y un subelemento de tipo complejo en un mismo elemento.

El siguiente código NO ES CORRECTO:

```

<element name="elementoErroneo" type="otroTipo">

  <complexType>

    <!-- declaración del tipo complejo-->

  </complexType>

</element>

```

Si se tiene un conjunto de atributos que son usados en varios elementos, XML Schema permite unirlos en un grupo, y hacer que dichos elementos los tengan una referencia hacia ellos.

Suponga que se tiene un XML en el que se guardarán distintas piezas, que son representadas por distintos elementos, pero de todos ellos se necesita mantener la fecha de entrada, el lote y proveedor, por lo que se puede agrupar estas tres características como un grupo atributos, y así no tener que expresarlo en todos y cada uno de los elementos:

```

<attributeGroup name="mercanciaAtt">

  <attribute name="f_entrada" type="date">

  <attribute name="lote" type="integer">

  <attribute name="proveedor" type="string">

</attributeGroup>

<complexType>

  <sequence>

    <element name="cantidad" type="integer" />

    <element name="calidad" type="string" />

  </sequence>

  <attribute ref="mercanciaAtt">

```

```
</complexType>
```

Se puede obligar al usuario a indicar ciertos elementos, que deben aparecer aunque no sea en orden. Por ejemplo si se hace un elemento que guarde las fichas componen un tablero de ajedrez:

```
<complexType name="piezasAjedrez">
  <all>
    <element name="peon" type="string" />
    <element name="rey" type="string" />
    <element name="reina" type="string" />
    <element name="caballo" type="string" />
    <element name="torre" type="string" />
    <element name="alfil" type="string" />
  </all>
</complexType>
```

En este caso, como no se han tenido en cuenta las cantidades de cada una de ellas, se tiene indicar una vez cada una. Se deja como sencillo ejercicio al lector, el modificar este XML Schema para que se puedan incorporar al elementos las piezas reglamentarias.

Complementos de los elementos

En ocasiones es posible que se necesiten nuevos tipos de datos que son muy parecidos entre ellos, con características básicas comunes. En programación orientada a objetos se usaría lo que se denomina herencia, es decir, generar un objeto con las características básicas y luego extenderlo para obtener el objeto deseado. Si piensa en los vehículos, todos tienen ruedas, todos tienen un motor con ciertos caballos de potencia, pero por ejemplo en un coche interesa conocer la capacidad del maletero, o el número de personas que pueden viajar en él, mientras que en un camión interesa conocer el peso máximo de la carga admitida, la autonomía, etc... En XML Schema se puede controlar estas herencias mediante el elemento *<extension>*. En el siguiente ejemplo se verá como se extiende el tipo base de nombre vehículo, para formar los elementos camión y coche.

```
<complexType name="vehiculo">
  <sequence>
    <element name="nruedas" type="integer">
    <element name="caballos" type="integer">
    <element name="marca" type="string">
    <element name="combustible" type="combusList">
  </sequence>
  <attribute name="identificador" type="string">
</complexType>
```

```

<simpleType name=" combusList" type="string">
    <enumeration value="Gasoleo"/>
    <enumeration value="Super 95"/>
    <enumeration value="Super 97"/>
</simpleType>

    <!--! Ahora creamos los elementos coche y camión haciendo una
extensión del elemento vehículo -->

<complexType name="coche">
    <complexContent>
        <extension base="vehiculo">
            <element name="plazas" type="integer">
            <element name="velocidadM" type="decimal">
        </extension>
    </complexContent>
</complexType>

<complexType name="camion">
    <complexContent>
        <extension base=" vehiculo">
            <element name="pmc" type="integer">
            <element name="autonomia" type="float">
        </extension>
    </complexContent>
</complexType>

```

Y para usar estos nuevos tipos, sólo se tiene que elegir en cada caso si se quiere declarar un coche o un camión:

```

<elemento type="camion">
    <nruedas>10</nruedas>
    <caballos>800</caballos>
    <marca>ACME</marca>

```

```

    <combustible>Gasoleo</combustible>

    <pmc>125800</pmc>

    <autonomia>3345.23</autonomia>

</elemento>

```

2.6. *Los NAMESPACES*

Ya ha quedado más que demostrado que XML es realmente flexible, ya que el propio programador es el que define las normas, él es quién dice como se llamarán los elementos, que forma tendrán y que significado tomarán dependiendo de su posición en el documento, pudiendo representar todo tipo de datos. Si además de todo esto se añade que es legible por cualquier plataforma (incluso para los humanos), los “clientes” de un documento XML pueden ser muchos y muy variados, dando la posibilidad de que alguno de ellos cruce el fichero con nuevos datos.

Aquí aparecerá un inconveniente, ya que el ser tan flexible puede traer problemas de colisiones de nombres, pudiendo llegar a llamarse igual, dos elementos de un mismo XML.

Suponga que realiza una base de datos con las películas que posee.

```

<?xml version="1.0" ?>

<videoteca>

    <pelicula titulo="El enemigo">

        <reparto>

            <director>Klaren Marcus</director>

            <actores>

                <actor>Malen Jhonson</actor>

                <actor>L.S. MacSmith</actor>

                <actriz>Lanie Stef</actriz>

            </actores>

        </reparto>

        <fecha>12/10/98</fecha>

        <duracion>1.80</duracion>

    </pelicula>

    <pelicula titulo="Sombras">

        <reparto>

            <director>Marlon Meth</director>

```

```

        <actores>
            <actor>Collin M.Dono</actor>
            <actriz>Yenie Power </actriz>
        </actores>
    </reparto>
    <fecha>29/01/99</fecha>
    <duracion>2.05</duracion>
</pelicula>
</videoteca>

```

Este XML puede satisfacer sus expectativas de cara a cubrir la información de sus películas, pero la información que contiene es ciertamente escasa, por lo que puede que haya alguien que opine igual, y decida completarla con nueva información, por ejemplo, añadiendo una clasificación por edades, y genera un documento como el siguiente:

```

<videoteca>
<pelicula titulo="Sombras">
    <reparto>
        <director>Marlon Meth</director>
        <actores>
            <actor>Collin M.Dono</actor>
            <actriz>Yenie Power </actriz>
        </actores>
    </reparto>
    <fecha>29/01/99</fecha>
    <duracion>2.05</duracion>
    <clasificacion>+13</clasificacion>
</pelicula>
</videoteca>

```

Puede que otra persona piense que lo que es realmente útil, es una clasificación por crítica cultural, y no quiera para nada mantener la clasificación anterior generando el siguiente documento:

```

<videoteca>
<pelicula titulo="Sombras">

```

```

    <reparto>
      <director>Marlon Meth</director>
      <actores>
        <actor>Collin M.Dono</actor>
        <actriz>Yenie Power </actriz>
      </actores>
    </reparto>
    <fecha>29/01/99</fecha>
    <duracion>2.05</duracion>
    <clasificacion>Obra maestra</clasificacion>
  </pelicula>
</videoteca>

```

Pero puede que aparezca ahora otro usuario, al cual le interese mantener las dos clasificaciones, pero bajo estas circunstancias, se van a dar incompatibilidades entre las dos últimas extensiones de su documento original, por aparecer dos veces el elemento clasificación:

```

<videoteca>
  <pelicula titulo="Sombras">
    <reparto>
      <director>Marlon Meth</director>
      <actores>
        <actor>Collin M.Dono</actor>
        <actriz>Yenie Power </actriz>
      </actores>
    </reparto>
    <fecha>29/01/99</fecha>
    <duracion>2.05</duracion>
    <clasificacion>+13</clasificacion>
    <clasificacion>Obra maestra</clasificacion>
  </pelicula>

```



```
</videoteca>
```

¿Qué pasará cuando el software analice el listado con dos elementos *<clasificación>* diferentes? ¿Cómo sabe si una clasificación es sobre cultura o sobre edades?. El analizador no sabrá que hacer con el elemento *<clasificación>*, y posiblemente devuelva error, pero lo curioso es que el último documento XML obtenido, puede llegar a cumplir las normas del DTD de alguno de los programas que trabajara con los documentos anteriores, por lo que podría estar leyendo datos corruptos, y pasar inadvertido. O puede que el primero de los usuarios que hizo modificaciones sobre el XML inicial, encuentre un documento XML realizado por el segundo usuario modificador. El XML cumple perfectamente las normas del primero, pero no serían datos válidos.

Existe un problema con los nombres de los elementos. Este inconveniente se puede solucionar con tan sólo cambiar el nombre de la etiqueta, pero se había dicho que XML era flexible, y el limitar el nombre de los elementos dependiendo de los que haya ya definidos en el escrito, puede ser un buen ejercicio de imaginación, y además puede llegar a ser muy molesto, recortando mucho las posibilidades de cambios rápidos (si se cambia el nombre del elemento puede que haya que cambiar código de los programas que lo usen). Aquí es donde entran en juego los *namespaces*.

El listado completo usando *namespaces* podría quedar de la siguiente manera:

```
<?xml version="1.0" ?>

<videoteca xmlns:ce="http://www.acme.com/clasificacion_edades"
           xmlns:cc="http://www.megacines.com/clasificacion_cultural"
           xmlns="http://www.miservidor.com/">

  <pelicula titulo="El enemigo">

    <reparto>

      <director>Klaren Marcus</director>

      <actores>

        <actor>Malen Jhonson</actor>

        <actor>L.S. MacSmith</actor>

        <actriz>Lanie Stef</actriz>

      </actores>

    </reparto>

    <fecha>12/10/98</fecha>

    <duracion>1.80</duracion>

    <cc:clasificacion>Entretenida</cc:clasificacion>

    <ce:clasificacion>Todos los públicos</ce:clasificacion>

  </pelicula>

  <pelicula titulo="Sombras">
```

```

    <reparto>
      <director>Marlon Meth</director>
      <actores>
        <actor>Collin M.Dono</actor>
        <actriz>Yenie Power </actriz>
      </actores>
    </reparto>
    <fecha>29/01/99</fecha>
    <duracion>2.05</duracion>
    <cc:clasificacion>Obra maestra</cc:clasificacion>
    <ce:clasificacion>+13</ce:clasificacion>
  </pelicula>
</videoteca>

```

Ahora ya está diferenciado cada uno de los elementos mediante un nombre único.

Hay que destacar que el *namespace* no es en sí la referencia (“cc” y “ce” en el ejemplo anterior), sino el URL al que se refieren, y esto es lo que lo hace único, ya que los URL están controlados por organismos como la interNIC, que se encargan de mantener una base de datos con entradas únicas para cada URL, garantizando que no existan dos iguales.

Como URL se pueden escribir direcciones simples o escribir dirección y ruta. Por ejemplo un URL con dirección:

```
http://www.acme.com
```

si usamos dirección y ruta:

```
http://www.acme.com/clasificacion_edades
```

Esta dirección puede contener un documento con las especificaciones de la clasificación, pero no es imprescindible, es decir, que se permite reseñar rutas que en caso de usarlas en un navegador reportaría un error del tipo 404 (documento no hallado).

Volviendo al hecho que la referencia es en sí el URL, implica que durante el análisis se trabajará con el URL, no en el prefijo, por lo que la siguiente declaración daría problemas:

```

<videoteca xmlns:ce="http://www.acme.com/clasificacion_edades"
  xmlns:cc="http://www.acme.com/clasificacion_edades"
  xmlns:dd="http://www.miservidor.com/">

```

El hecho de que dé problemas es por que, pese a que las clasificaciones de edades y culturales tienen distinto prefijo (ce y cc respectivamente), las dos se refieren al mismo URL, con lo que colisionarán las definiciones de nombres de elementos.

Se ha de tener de tener en cuenta donde se realiza la referencia al *namespace*, ya que el *namespace* no será válido en cualquier parte del documento, sino solamente en el elemento en el que se ha definido y en cualquiera de sus hijos. Por lo que si se necesita que se pueda usar a lo largo del documento, se deberá definir en el elemento raíz.

En siguiente caso, la última película ha sido mal definida puesto que el alcance de la definición del *namespace* muere con el elemento dentro del cual se ha definido, en este caso muere con `</duracion>`, y al ser un elemento hermano de la clasificación, no alcanzan las definiciones.

Si se fija en la primera película, el *namespace* en este caso ha sido definido en un elemento padre de la clasificación por lo que sí que puede ser usado sin problemas en ésta.

```
<?xml version="1.0" ?>

<videoteca xmlns:ce="http://www.acme.com/clasificacion_edades"

    xmlns="http://www.miservidor.com/">

  <pelicula titulo="El enemigo">

    <reparto>

      <director>Klaren Marcus</director>

      <actores>

        <actor>Malen Jhonson</actor>

        <actor>L.S. MacSmith</actor>

        <actriz>Lanie Stef</actriz>

      </actores>

    </reparto>

    <fecha>12/10/98</fecha>

    <duracion>1.80</duracion>

    <cc:clasificacion
xmlns:cc="http://www.megacines.com/clasificacion">

  Entretenida

</cc:clasificacion>

  <ce:clasificacion>Todos los públicos</ce:clasificacion>

</pelicula>

<pelicula titulo="Sombras"
xmlns:cc="http://www.megacines.com/clasificacion">

  <reparto>

    <director>Marlon Meth</director>
```

```

        <actores>
            <actor>Collin M.Dono</actor>
            <actriz>Yenie Power </actriz>
        </actores>
    </reparto>
    <fecha>29/01/99</fecha>
    <duracion>2.05</duracion>
    <cc:clasificacion>Obra maestra</cc:clasificacion>
    <ce:clasificacion>+13</ce:clasificacion>
</pelicula>
<pelicula titulo="La ultima puerta">
    <reparto>
        <director>T.R. Scot</director>
        <actores>
            <actriz>Z. Pomodoro</actriz>
            <actor>Ivan J. Rosas</actor>
        </actores>
    </reparto>
    <fecha>29/01/99</fecha>
    <duracion xmlns:cc="http://www.megacines.com/clasificacion">
2.05</duracion>
    <cc:clasificacion>Obra maestra</cc:clasificacion>
    <ce:clasificacion>+13</ce:clasificacion>
</pelicula>
</videoteca>

```

Supongo que la duda ya habrá atacado al lector.. si existen problemas de duplicidad de definición en los elementos.¿existen también en los atributos?. Efectivamente, aunque es bastante raro, se pueden dar casos, y la manera de solucionarlo vuelve a ser mediante *namespaces*, aunque muchas veces la mejor solución pasa por un diseño mejor del esquema.

Suponga que como nombre de película se quiere mantener el título original y el comercializado en varios países (que curiosamente, rara vez coinciden las traducciones), y se mantendrán con el atributo

“titulo”. Además, para hacer un poco más representativo el ejemplo, imagine también que existe una empresa llamada “acme”, que es propietaria de varias páginas web a escala mundial, con las traducciones, en distintos idiomas, de las informaciones de las películas. El ejemplo que se detalla a continuación, sirve para ilustrar un problema que puede llegar a darse, pero no es muy frecuente.

Para cada uno de los títulos se crea un *namespace* para en el primer elemento (raíz) y así aprovecharlo en todos los elementos. Obtendremos un XML:

```
<?xml version="1.0" ?>

<videoteca xmlns:ce="http://www.acme.com/clasificacion_edades"

    xmlns:ms="http://www.miservidor.com/"

    xmlns:us="http://www.acme.us/tittle"

    xmlns:fr="http://www.acme.fr/tittle"

    xmlns:en="http://www.acme.en/tittle"

    xmlns:ge="http://www.acme.ge/tittle"

>

<pelicula ms:titulo="El enemigo" us:titulo="The enemy" fr:titulo=""
en:titulo="" ge:titulo="">

    <reparto>

        <director>Klaren Marcus</director>

        <actores>

            <actor>Malen Jhonson</actor>

            <actor>L.S. MacSmith</actor>

            <actriz>Lanie Stef</actriz>

        </actores>

    </reparto>

    <fecha>12/10/98</fecha>

    <duracion>1.80</duracion>

    <cc:clasificacion xmlns:cc="
http://www.megacines.com/clasificacion">

    Entretenida

    </cc:clasificacion>

    <ce:clasificacion>Todos los públicos</ce:clasificacion>

</pelicula>
```

```

<pelicula xmlns:cc=" http://www.megacines.com/clasificacion"
ms:titulo="Sombras" us:titulo="Sombree" fr:titulo="" en:titulo=""
ge:titulo="">

  <reparto>

    <director>Marlon Meth</director>

    <actores>

      <actor>Collin M.Dono</actor>

      <actriz>Yenie Power </actriz>

    </actores>

  </reparto>

  <fecha>29/01/99</fecha>

  <duracion>2.05</duracion>

  <cc:clasificacion>Obra maestra</cc:clasificacion>

  <ce:clasificacion>+13</ce:clasificacion>

</pelicula>

```

La mayor parte de las colisiones de atributos se dan por la definición de atributos globales, por lo que muchas veces es mejor realizar otros diseños.

Las clasificaciones se han definido de formas distintas, en cada caso, la clasificación por edades se ha definido al principio del documento, en el elemento raíz, mientras que la clasificación cultural, le ha definido en cada elemento donde se necesitaba. Lógicamente, se podrían haber definido de muchas maneras distintas.

2.7. Analizadores

El procesado de los documentos XML puede ser, más tedioso que difícil, por lo que existen unas utilidades que ofrecen al programador su ayuda durante el análisis de la estructura del documento. Estos analizadores son conocidos como *parsers* (del inglés to parse: analizar).

Pese a que nada impide que el programador se ponga a trabajar directamente con el documento, explorándolo y extrayendo las partes que le interesen en cada momento, no es lo más normal (¿para qué hay que volver a descubrir la rueda?), siempre se usa un parser, que será el encargado de acceder al medio de almacenamiento donde se tenga el documento XML (buffers, ficheros,...) y lo recorrerá, mientras que el programador no tendrá más que encargarse de transmitirle al *parser* sus intenciones, es decir darle órdenes dependiendo de lo que esté leyendo en cada momento, bien leer atributos de un elemento, leer la lista de hijos, validar, etc...

Los dos *parsers* más comunes son el SAX (Simple Api for XML, interfaz de acceso simple al XML) y el DOM (Document Object Model) usando nivel uno o dos, que difieren, sobre todo, en su modo de funcionamiento a la hora de acceder a los datos.

Estos analizadores no son exclusivos de un lenguaje, ni de una compañía. Existen infinidad de implementaciones para ambos, e incluso existen compañías que ofrecen un mismo *parser* en varias versiones, unas con mas opciones que otras, para que el programador elija dependiendo de sus necesidades.

2.7.1. SAX

SAX proporciona un mecanismo de acceso al XML de forma secuencial, por lo que será ideal para aplicaciones cuyos documentos XML a analizar, vengan dados por comunicaciones entre procesos (misma o distinta máquina), tanto en transmisión como en recepción.

Este *parser* necesita relativamente poca memoria para trabajar, ya que una vez procesada la sección de documento pedida, la abandona. Un analogía de la manera de trabajar de SAX, sería una cinta de casete, que es recorrida secuencialmente, y en cada momento lee tan solo una parte de ella.

En cada momento se tiene una parte del documento en la memoria, y si se necesita otra parte distinta, se abandona la actual, hasta encontrar lo que se estaba buscando, se almacena esta sección en memoria y se procesa.

SAX comenzará a recibir el documento a analizar, y avisará de los elementos que se va encontrando, el programador será el que se encargue de mirar el nombre del elemento leído y actuar en consecuencia.

En SAX el programador es el que se debe encargarse de generar el código que atienda a los eventos que se suceden al recorrer el documento, distinguiendo comportamientos para atributos, marcas de proceso de programa, elementos...

Esto es lo que se llama un *parser* orientado a evento (event-driven).

La memoria utilizada por este *parser*, no depende de ninguna manera del tamaño del documento a analizar, sino que depende de la implementación del analizador SAX que se esté usando.

2.7.2. DOM

DOM es un *parser* más pesado que SAX, ocupa más memoria, pero por otro lado se tiene la ventaja de que mucha de la programación que se tenía que hacer para usar SAX, ahora no es necesaria en DOM. Además se tiene mayor control sobre la estructura física del documento y de las descendencias jerárquicas entre elementos.

DOM es un *parser* ideal para volcar en pantalla información contenida en documentos XML, o para la lectura de un fichero de configuración de alguna aplicación (se deberá tener acceso total al fichero, no debe ser transmitido por red).

La manera en la que trabaja DOM es totalmente distinta a SAX. DOM lo primero que hace es leer el documento en su totalidad, generando internamente un árbol, correspondiendo a la estructura del documento leído.

Debido a que DOM carga todo el documento en memoria, para después ser procesado, se requiere mucha memoria, y se requerirá más cuanto más mayor sea el tamaño del documento analizado y puede traer problemas en dispositivos móviles en los que la memoria es un bien escaso.

Existen implementaciones de DOM que ocupan más memoria que otras, ya que generan y guardan índices sobre la colocación de los datos, y tablas de jerarquía, lo que les hace más rápidos.

El acceso al documento para la recuperación, grabación y/o modificación de datos, se realiza por nodos. Estos nodos pueden ser principalmente de dos tipos, nodos de texto o nodos elemento. La diferencia entre ellos es clara; el nodo elemento puede contener otros elementos, atributos, o texto, mientras que los nodos de texto son sólo texto.

Normalmente las implementaciones de DOM, permiten acciones como peticiones de devolución de la información de cierto nodo en concreto, o sus valores, o sus atributos, o una lista con todos sus elementos descendientes, etc..

Aunque SAX sea mejor que DOM en documentos transmitidos, nada impide usar SAX para leer un fichero desde disco duro, ni utilizar DOM para trabajar con un documento transmitido, pero las eficiencias no serían las mismas

2.8 Conclusión

Como se ha podido ver, el XML permite infinidad de posibilidades de representación de datos, así pues no es extraño que se esté tomando como estándar de cientos de utilidades, y como día a día surgen nuevos campos donde se introduce el uso de XML su importancia crece aún mas.

En estas páginas se ha tratado de dar una visión general de lo que puede ofrecer XML y como se debe usar, aunque recomiendo al lector que realice más pruebas en su ordenador, escribiendo ejemplos de XML y usando archivos de validación, así conseguirá una mayor familiarización con este lenguaje.

Espero que le haya quedado claro al lector lo que es XML y como se usa, puesto que en los capítulos posteriores se hará uso de él, por ser la base de representación de datos en los servicios web.

El lenguaje SOAP

3.1. Introducción

SOAP son las siglas de Standar Object Access Protocol, o protocolo estándar de acceso a objetos.

Mediante SOAP se define el estándar que regula las reglas de serialización o empaquetado de datos y de generación de los mensajes transmitidos durante una transacción en la que se incluye un web service. Aunque SOAP no es la única manera de realizar esta acción, sí es la más extendida actualmente.

Pese a que en ocasiones se trate a SOAP como un nuevo lenguaje de programación, no es más que un conjunto de normas que hacen uso de XML, una definición de la composición del documento, de sus partes y de sus contenidos.

Realmente de lo que se encarga SOAP es de dictar cómo se deben empaquetar los datos para que puedan ser transmitidos interpretados por otro usuario; marcará las directrices acerca de cómo se deben posicionar los datos y cómo definir el significado cada uno de ellos.

Cada conjunto de datos SOAP empaquetados correctamente se denomina mensaje, es decir con el uso de los servicios web, se está definiendo un área de trabajo controlada por mensajes, en la que cada uno de ellos transmite información que implica la realización de una o varias acciones.

Como protocolo de transporte entre emisor y receptor del mensaje SOAP, puede usarse cualquiera de los protocolos soportados por Internet de manera estándar (HTTP, SMTP, FTP) o incluso sin ser estándares (Jabber).

3.2. El mensaje SOAP

El mensaje SOAP será un documento que contendrá los datos necesarios para realizar una llamada a un servicio. Entre la información contenida en estos datos, figurarán aspectos tan importantes como el nombre de la función a la que se quiere llamar y los parámetros que sean necesarios para realizar dicha llamada.

Al igual que en la llamada al servicio, en la respuesta de éste, también se usará un documento SOAP, que contendrá entre otros datos, la información correspondiente al resultado de la ejecución de la función demandada en el documento de llamada.

Veamos un ejemplo de cómo sería un mensaje SOAP a través de una petición HTTP:

```
POST /soap/servlet/rpcrouter HTTP/1.1
Host: www.acme.com
Content-Type: text/xml; charset="utf-8"
Content-Length: X

SOAPAction: "my-URI"

<?xml version="1.0" encoding="UTF-8"?>

<SE:Envelope
  xmlns:SE="http://schemas.xmlsoap.org/soap/envelope/"
  SE:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SE:Body>
    <dat:getDeudas xmlns:dat="http://www.acme.com">
      <id>00293</id>
    </dat:getDeudas>
  </SE:Body>
</SE:Envelope>
```

Y su respuesta:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: X
Date: Fri, 03 Jan 2002 00:30:50 GMT

Server: Apache Tomcat/4.0.6 (HTTP/1.1 Connector)

Set-Cookie: JSESSIONID=0FAAD3E15FFC3D053FEE03E45F724C4E;Path=/soap

<?xml version="1.0" encoding="UTF-8"?>
<SE:Envelope
  xmlns:SE="http://schemas.xmlsoap.org/soap/envelope/"
  SE:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  <SE:Body>
    <dat:getDeudasResponse xmlns:dat="Some-URI">
      <Valor>132.63</Valor>
    </dat:getDeudasResponse>
  </SE:Body>
</SE:Envelope>

```

Si se fija, podrá ver que el mensaje SOAP no es nada nuevo, es realmente un uso específico del XML, y el hecho es que se apoya directamente sobre él (por lo que también se podrá observar el uso de *namespaces* aunque no de DTD ya que su uso está explícitamente prohibido en SOAP), pero que además, gracias a las propiedades heredadas del uso del XML (ya que este lenguaje está estandarizado) se pueden realizar cosas tan increíbles como poder usar objetos que estén en Nueva York desde cualquier parte del mundo, sin importar el lenguaje de programación que se haya utilizado en cada caso, ni necesidad de conocer la plataforma que aceptará la petición.

En el ejemplo anterior se llama a la función “*getDeudas*”, a la cual se le pasa como parámetro de la llamada el identificador de la persona o socio del cual se quiere hallar la deuda contraída. En un principio no se sabe nada (ni interesa) de cómo funciona el objeto que atenderá la petición, ni si va a buscar el dato a un fichero, a una base de datos o llama a otro web service, lo único que interesa en este caso, es que se tiene que recibir un mensaje que contenga en su cuerpo (elemento *<Body>*) un elemento hijo llamado *<getDeudasResponse>*, que éste será de un tipo específico (en coma flotante por ejemplo), y que representa la respuesta a la petición realizada, o en otras palabras y para este ejemplo, representa la cantidad de deuda contraída por el socio indicado en la petición.

3.2.1. Tipos de mensajes

No todos los mensajes tienen la misma forma, ni tienen la misma finalidad. Como se vio en el ejemplo anterior, tras una petición se obtuvo una contestación, pero esto no es siempre así, en ocasiones no existe respuesta por parte del servicio, y esto es simplemente por que un documento de petición no implica un documento de respuesta.

Tomando como referencia las respuestas recibidas se podrían clasificar los tipos de las transacciones de mensajes como:

- **Petición**
En este tipo de transacciones sólo interviene un documento, en ningún momento se espera una contestación por parte del servicio que atiende esta petición. Suelen ser documentos comúnmente llamados EDI (Electronic Document Interchange, Intercambio Electrónico de documentos)
- **Petición - recepción**
En este tipo de transacciones se emite un mensaje, que es procesado por el servicio y se espera una contestación por parte de éste, bien con los resultados de la ejecución o con cualquier otro tipo de datos que se puedan necesitar. Suelen ser los documentos normalmente llamados RPC. Aunque la realidad es que los sistemas basados en SOAP, son sistemas de una sola dirección, por lo que los mensajes de ida y de vuelta, serán independientes.

Si se tiene en cuenta el contenido de los mensajes, la clasificación podría quedar:

- Documentos EDI (Electronic Document Interchange)

Estos documentos suelen ser informaciones a nivel administrativo, como peticiones de compras o presentaciones de información. Por ejemplo, si se tuviera que fichar en el trabajo antes de irse a casa, se podría enviar un documento EDI con las horas y los conceptos en lo que se ha trabajado.

Un símil de un documento EDI sería el correo electrónico; se envía una petición y no se espera respuesta de confirmación de llegada de mensaje, simplemente se confía en que haya llegado a su destinatario.

- Documentos RPC (Remote Procedure Call)

Estos documentos, como su propio nombre indica, servirán para realizar peticiones de ejecución de procedimientos. En ellos se especificarán parámetros de la función a la que se desea llamar, y tras su emisión se recibe una respuesta (normalmente, aunque no es obligatorio) con el resultado de la petición realizada.

Un ejemplo podría ser una petición a un servicio de consulta de códigos postales, al que se le llama dándole como parámetros una dirección y una población, y él devuelve el número de código postal que corresponde a los datos suministrados. Como es lógico, no tiene sentido que no haya una respuesta.

Las peticiones de este tipo pueden ser síncronas o asíncronas, si bien lo normal es que sean síncronas, para permitir una interacción (soft real time o tiempo real blando) entre cliente y servidor. Existen protocolos que disponen de mecanismos para enviar un mensaje y olvidarse de él, y cuando sea procesado, bien enviar de forma desatendida la información resultante de la acción del servicio web (vía correo electrónico por ejemplo), o bien no dar ningún mensaje de confirmación. Un ejemplo de este tipo de protocolos es Jabber.

Una muestra del uso de protocolos que permitan este tipo de mensajes es, si existiera un servicio en un ordenador que permitiera realizar un chequeo de la información de un archivo que se envía en el propio mensaje. No tiene sentido remitir la petición de este servicio y estar esperando mientras se procesa este fichero, sino que lo lógico, es que se haga de forma desatendida, y cuando finalice muestre los resultados de alguna manera, como por ejemplo mediante un correo electrónico.

3.3. Constitución del mensaje SOAP.

Como ya se ha dicho en anteriores capítulos, los mensajes SOAP sirven para el intercambio de información. Pero ¿pueden estos mensajes tener cualquier forma?, La respuesta es sí y no. Suponga una urbanización de casas unifamiliares en las que todas las casas son iguales... por fuera, por que por dentro, cada familia la decora y distribuye los muebles según sus necesidades. Algo semejante se da en el mensaje SOAP. Los primeros niveles (como si se viera por fuera) que son los correspondientes a los elementos *<Envelope>*, *<Header>* y *<Body>* son siempre iguales, cambiando los contenidos internos para el acomodo del uso que le vaya a dar el programa.

La estructura de un mensaje SOAP se parece en las formas a un documento HTML; se compone de una cabecera y un cuerpo. La cabecera posee un carácter opcional, y en ella se pueden incluir distintos bloques especificando aspectos como autentificaciones. El cuerpo es obligatorio y al igual que la cabecera, puede estar compuesto de varios bloques con información.

Esquemáticamente podría representarse de la siguiente forma:

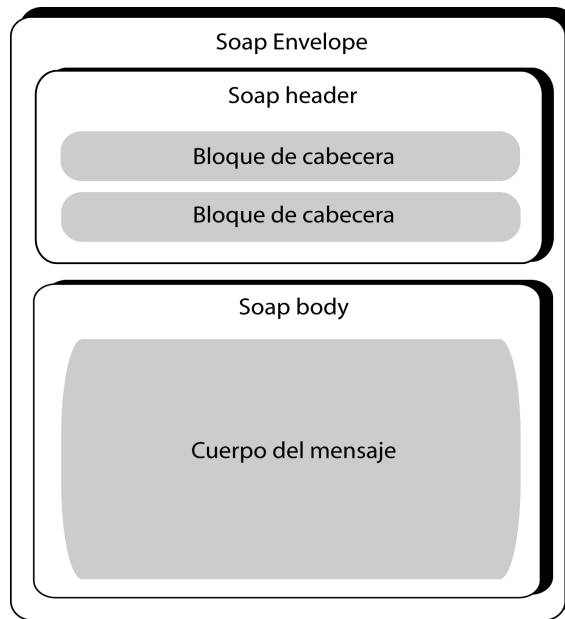


Figura 5: Representación de un mensaje SOAP.

El esquema llevado código SOAP sería:

```
<s:Envelope xmlns:s="http://www.w3.org/2001/6/soap-envelope">
  <s:Header>
    <!-- bloques de la cabecera-->
  </s:Header>
  <s:Body>
    <!-- bloques del cuerpo-->
  </s:Body>
</s:Envelope>
```

Todo mensaje SOAP se compone de tres partes diferenciadas:

- Envelope
- Header
- Body

3.3.1. Envelope

Es el elemento raíz de todo el documento, su nombre tiene que ser obligatoriamente *<Envelope>*, y dentro de él se encontrarán un elemento (solamente uno o ninguno) *<Header>* y un y sólo un bloque *<Body>*.

El elemento `<Body>` tiene que estar presente en todos los mensajes SOAP para ser un mensaje válido, ya que como toda la información enviada tiene que ir encerrada en su interior, si no existiera, sería un mensaje vacío y por lo tanto no serviría para nada.

Por otra parte el elemento `<Envelope>` tiene que estar identificado usando *namespaces* (full name qualified, nombres totalmente calificados), al igual que todos los atributos que contenga.

Usando la definición DTD que se ha visto en el capítulo dos, podríamos representar el mensaje SOAP de la siguiente forma:

```
<!--representación de un mensaje SOAP -->
<!ELEMENT Envelope (Header?,Body)>
<!ELEMENT Header(...)>
<!ELEMENT Body(...)>
```

3.3.1.1. Versión

Dentro de la sección `<Envelope>` se deberá especificar la versión de SOAP utilizada en el documento, para que el receptor sea capaz de interpretarla correctamente. La versión más difundida es la versión 1.1, aunque la última versión disponible es la 1.2. La causa de que la versión 1.1 esté tan difundida es que hay algunos lenguajes que aún no tienen disponibles implementaciones para la versión 1.2.

Al escribirse en el documento la versión utilizada, permitirá al servidor que reciba el mensaje, rechazarlo si no cumple su misma versión. Si un servidor espera una versión 1.2 y le llega una petición con versión 1.1 se puede procesar o rechazar, pero en caso de que la versión esperada sea una 1.1 y llegue una versión 1.2, la acción inmediata es el rechazo del mensaje.

Esta reacción de devolución del mensaje, puede usarse para incluir en la respuesta avisos de actualización indicando la versión que se esperaba. Hay que tener en cuenta que si el cliente ha usado SOAP 1.1 y el servicio está utilizando 1.2, habrá que devolver el error usando 1.1, ya que sino no lo podría comprender.

Ejemplo de cabecera conteniendo la versión 1.1:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope">
  <!--Otros bloques -->
</s:Envelope>
```

Ejemplo de cabecera conteniendo la versión 1.2:

```
<s:Envelope xmlns:s="http://www.w3c.org/2001/06/soap-envelope">
  <!--Otros bloques -->
</s:Envelope>
```

Si se produce un error por un uso de versión incorrecto, se debe explicar al cliente que no se ha podido procesar el mensaje e indicarle la causa para que le ponga remedio. A continuación se presenta un ejemplo de respuesta con errores de versión, este error se ha dado por que el servidor espera una versión 1.2 y el cliente está usando una 1.1, hay que fijarse que el mensaje será devuelto usando la versión 1.1, para que el cliente sea capaz de procesarlo:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope">
  <s:Header>
```

```

    <e:Upgrade xmlns:e="http://www.w3.org/2001/6/soap-upgrade">
        <envelope qname="m:Envelope"
xmlns:m="http://www.w3.org/2001/6/soap-envelope"/>
            </e:Upgrade>
    </s:Header>
    <s:Body>
        <s:Fault>
            <faultcode>s:VersionMismatch</faultcode>
            <faultstring>Version Mismatch</faultstring>
        </s:Fault>
    </s:Body>
</Envelope>

```

Más adelante se verán otros tipos de errores.

3.3.2. Cabecera

El elemento *<Header>* (cabecera) no es obligatorio en los mensajes, si bien es aconsejable su uso para complementar la información referente a la transmisión; y por estructura, es buena costumbre añadir siempre la cabecera aunque esté vacía.

Si está presente, entonces tiene que ser el primer elemento hijo del elemento *<Envelope>*.

```

<s:Envelope xmlns:s="http://www.w3.org/2001/6/soap-envelope">
    <s:Header>
        <!--otros bloques -->
    </s:Header>
    <!--otros bloques -->
</s:Envelope>

```

Otros puntos a tener en cuenta en la generación de una cabecera son los siguientes:

- El elemento se tiene que llamar obligatoriamente *<Header>*.
- El elemento puede contener ningún o varios bloques hijos de información y todos ellos tienen que estar identificados mediante un *namespace*.
- Sólo se tendrán en cuenta aquellos atributos que estén definidos en alguno de los hijos directos del elemento *<Header>*.
- La cabecera permite introducir variantes en el mensaje SOAP, como rutas, autenticación, etc.. Todo lo que se incluye en la cabecera, es procesado antes de analizar cualquier parte del elemento *<Body>*, por lo que si se insertan elementos de seguridad en el mensaje, será en esta sección donde se deba hacer.

- Se debe usar un estilo de codificación, mediante el atributo *encodingStyle*, en todos los bloques o bien definir un estilo en el bloque raíz.

Para los elementos de *<Header>* hay disponibles tres atributos que tienen una relevancia especial por ser bastante usados:

- *mustUnderstand*
- actor o role
- *encodingStyle*

3.3.2.1. *mustUnderstand*

Es un atributo de tipo booleano e indica si un elemento de la cabecera puede ser pasado por alto o no en el receptor.

Si al receptor le llega una cabecera marcada con este atributo y con un valor igual a uno, entonces el receptor debe procesar esta entrada de forma que si se produjera un error en la interpretación, el resto del mensaje se desecharía y se retornaría como erróneo, devolviendo un mensaje de error denominado "*MustUnderstand fault*" (fallo debido al atributo *MustUnderstand*).

Si se produce el error, el mensaje de respuesta debería contener información acerca de los bloques, de la cabecera de petición que han sido la causa que ha originado el error. El problema es que en el único lugar donde se podría dar ésta información sería en un bloque del elemento *<Body>*, pero estos bloques están reservados (como veremos más adelante) para devolver información de la causa, no del lugar.

En la versión 1.2 de SOAP se introduce un bloque de cabecera específico para poder informar del lugar del error. El nombre del elemento es *<Misunderstood>* y en él se especifica el bloque que propició el rechazo del mensaje por no poder comprenderse.

El valor del atributo *mustUnderstand* puede ser 0 (o "false", falso) o 1 (o "true", verdadero), siendo por defecto el valor *mustUnderstand="0"* y el alcance de influencia se extiende desde el elemento en el que es definido hasta todos sus descendientes, así puede indicarse de manera global en la etiqueta de la cabecera, para referenciar que todos y cada uno de los bloques deben de ser "comprendidos".

Ejemplo de una cabecera con un bloque marcado como *mustUnderstand="1"*:

```
<SE:Envelope xmlns:SE="http://www.w3.org/2001/6/soap-envelope">
```

```
<SE:Header>
```

```
  <p:elem1 xmlns:p="http://miurl.com/ext"
```

```
    SE:mustUnderstand="1" />
```

```
  <d:elem2 xmlns:d="http://miurl.com/ext2"
```

```
    SE:mustUnderstand="1" />
```

```
  <d:elem3 xmlns:d="http://miurl.com/ext3"
```

```
    SE:mustUnderstand="1" />
```

```
</SE:Header>
```

```
  <!-- Resto de bloques-->
```

```
</SE:Envelope>
```

Si de la cabecera anterior no se pudiera comprender ni el elemento *<elem1>* ni el elemento *<elem2>*, el mensaje devuelto tendría la siguiente forma. Hay que fijarse que la cabecera está definiendo un mensaje SOAP versión 1.2, y así se tiene la oportunidad de incluir el elemento *<Misunderstood>*, donde es posible dar explicaciones del lugar exacto donde se ha producido el error:

```

<?xml version="1.0" ?>

<env:Envelope xmlns:env='http://www.w3.org/2002/06/soap-envelope'
              xmlns:flt='http://www.w3.org/2002/06/soap-faults' >

  <env:Header>

    <flt:Misunderstood qname='p:elem1'
                      xmlns:p='http://miurl.com/ext2' />

    <flt:Misunderstood qname='d:elem2'
                      xmlns:d='http://miurl.com/ext2' />

  </env:Header>

  <env:Body>

    <env:Fault>

      <env:Code><env:Value>env:MustUnderstand</env:Value></env:Code>

      <env:Reason>One or more mandatory
                  headers not understood</env:Reason>

    </env:Fault>

  </env:Body>

</env:Envelope>

```

3.3.2.2. actor o role

Un mensaje SOAP no tiene por que ser entregado directamente al destinatario, sino que es posible que desde que sale del emisor hasta que llegue al destinatario, pase por una serie de intermediarios, capaces de recibir y transmitir mensajes. Incluso puede ser que no todas las partes del mensaje SOAP sean para el último destinatario, sino que el mensaje pueda contener datos para varios de los intermediarios, como por ejemplo firmas de seguridad.

Estos destinos intermedios se marcan mediante el atributo *actor*. La diferencia sustancial entre el atributo actor y role, es la versión de especificación en la que se base el mensaje. En la versión SOAP 1.1 aparece como *actor*, y en la versión 1.2 aparece como *role*.

El atributo *actor* puede ponerse como global en la cabecera o como local en alguno de los elementos de la misma, indicando el receptor mediante una URL. Existe una URL especial para indicar que el siguiente intermediario del mensaje (sea quien sea), es el destinatario de ese elemento, esta dirección es <http://schemas.xmlsoap.org/soap/actor/next>.

En el caso de no indicar ningún actor, el destinatario será el último eslabón de la cadena por la que pase el mensaje SOAP.

Si uno de los intermediarios detecta un elemento en el que esté él marcado como actor, éste no deberá volver a transmitir dicho elemento al siguiente intermediario, ni si quiera si el siguiente intermediario es el destinatario final, sino que debe quedarse con él y atenderlo, y en su caso (si procede) seguir con la transmisión (una vez eliminado su elemento).

3.3.2.3. *encodingStyle*

Este atributo es utilizado para indicar las reglas de empaquetado que se usarán en el elemento en el que está definido. Este atributo debe aparecer en cada elemento, y su alcance es el propio elemento y todos sus hijos, a no ser que éstos tengan definido a su vez otro atributo *encodingStyle*. No existe ningún valor por defecto.

3.3.3. Cuerpo

El cuerpo de mensaje SOAP es donde esta contenida la información a tratar. Normalmente (al igual que en HTML), es la parte que más ocupa en el documento.

Obligatoriamente su nombre debe ser “*Body*” y su presencia dentro del elemento `<Envelope>` es de carácter obligatorio y único, esto es, tiene que haber un y solamente un elemento `<Body>` dentro del elemento `<Envelope>`, además tiene que ser descendiente directo de éste.

Si existiera el elemento `<Header>`, entonces `<Body>` tiene que estar al finalizar esta cabecera, y ser lo inmediatamente siguiente, si no existiera, entonces `<Body>` sería el primer y único descendiente del elemento `<Envelope>`.

El elemento puede contener varios bloques. Todos los bloques tienen que estar cualificados con un *namespace*.

El cuerpo del mensaje SOAP es el lugar donde se empaquetará la información que se desee pasar entre el emisor y el receptor. La serialización de la información se realizará siguiendo unas pautas marcadas por los esquemas.

Como se ha comentado anteriormente, dentro del elemento `<Envelope>`, sólo se puede tener un cuerpo, pero dentro de éste se pueden marcar tantos bloques como se necesite para obtener un documento bien formado. Su esquema sería

```
<s:Envelope xmlns:s="http://www.w3.org/2001/6/soap-envelope">

    <!--El elemento Header es opcional -->

    <s:Header>

        <!--otros bloques -->

    </s:Header>

    <!--El elemento Body es obligatorio -->

    <s:Body>

        <!--otros bloques -->

    </s:Body>

</s:Envelope>
```

3.4. *Empaquetado de datos*

Como se ha mencionado anteriormente, uno de los objetivos de SOAP, es hacer llegar datos de una máquina a otra sin saber que tipo de sistemas están involucrados en el intercambio; por lo que no sirve sólo que sepa leer el documento y que entienda sus partes, sino que tiene que entender también los datos que en ellas se contienen.

Para ello se debe especificar de alguna manera el método de empaquetado (serialización) de los datos introducidos, así tanto cliente como servidor sabrán llevar los datos contenidos en el texto a datos nativos específicos del lenguaje de programa encargado de analizar el XML.

SOAP nos define una posible manera de realizar esta serialización de datos, para que podamos pasar rápidamente de código nativo a XML y viceversa. La sección 5 del estándar SOAP define estas reglas.

El tipo de codificación (*encoding style*) viene dado por el atributo con el nombre “*encodingStyle*”, y afectará al elemento en el que se defina y a todos sus elementos descendientes a no ser que se especifique otro tipo de codificación en alguno de ellos.

El alcance de la declaración del estilo de empaquetado, afecta al elemento dentro del cual se declare, y a todos sus descendientes que no tengan definido dicho atributo.

El uso de esta codificación en concreto, no es obligatorio, ya que se permite usar otros tipos definidos por algún fabricante, por ejemplo de IBM o nuevos estilos definidos por el programador, pero ambas partes deben usar el mismo para poder interpretar correctamente el contenido.

Un ejemplo del uso de *encodingStyle*:

```
<m:Envelope xmlns:m="http://www.w3.org/2001/6/soap-envelope">
<m:Body>
  <n:codigoPostal xmlns:n="urn:ProveedorCP"
m:encodingStyle="http://www.w3.org/2001/6/soap-encoding">
    <calle xsi:type="xsd:string">Relatores</calle>
    <portal xsi:type="xsd:int">5</portal>
    <ciudad xsi:type="xsd:string">Valladolid</ciudad>
  </n:codigoPostal >
</m:Body>
</m:Envelope >
```

3.4.1. ¿Qué tipos se pueden serializar?

Básicamente se pueden serializar los tipos definidos en la especificación de datos XML (XML Schema Data Types).

SOAP incluso admite la utilización de datos binarios para su transporte (datos que no son cadenas de texto, por ejemplo un objeto complejo o un JavaBean). Para ello se han de convertir en cadena de texto (no se tiene que olvidar que se trabaja con texto plano), y se debe usar el mismo método de serialización en el emisor que en el receptor. SOAP recomienda en este caso el uso de la codificación base64. Un ejemplo de serialización de una cadena de bits:

```
<myBean xsi:type="SOAP-ENC:base64">
  AD559AAE9BFA346895ABF08878236DD47ACB33
</myBean>
```

3.4.1.1. Tipos simples

Los tipos básicos que se pueden codificar, son los definidos por especificación de datos XML (XML Schema Data Types), y comprenden, entre otros, enteros, datos en punto flotante y cadenas. Además soporta también cualquier dato derivado de éstos, como estructuras.

Por ejemplo:

```
<element name="pisos" type="int" />
<element name="precio" type="float" />
...
<pisos>11</pisos>
<precio>3.90</precio>
```

Las declaraciones se pueden realizar “en línea”, indicando el tipo del dato a la vez que su valor:

```
<SOAP-ENC:int id="pisos">11</SOAP-ENC:int>
```

Incluso es posible la codificación de enumeraciones, previa declaración.

```
<element name="raza">
  <simpleType base="xsd:string">
    <enumeration value="Blanco" />
    <enumeration value="Negro" />

    <enumeration value="Mulato" />
  </simpleType>
</element>
```

...

```
<raza>Mulato</raza>
```

3.4.1.1.1. Cadenas de texto

Con este tipo se ha de tener cuidado puesto que no coincide exactamente con lo que en algunos lenguajes se llama cadena de texto, ya que por ejemplo en C, la cadena de texto termina con el carácter nulo. Dependiendo de las codificaciones y el lenguaje utilizado, la cadena de texto se empaquetará con un tipo distinto al *string*, como por ejemplo *hexBinary*.

Ejemplo:

```
<empleado xsi:type="xsd:string">Juan Manuel Mero</empleado>
```

```
<direccion xsi:type="xsd:string">Plz. Los Laureles</direccion>
```

3.4.1.1.2. Enumeraciones

SOAP hereda este mecanismo directamente de XML. Las enumeraciones permiten definir un conjunto de datos del mismo tipo y con distinto valor.

Las enumeraciones pueden ser de todos los tipos simples excepto del booleano (implícitamente es una enumeración de valores “true” y “false”), y éste se debe indicar a la hora de definir la enumeración. Si no se especifica ningún tipo en la definición, se usará el tipo cadena (*string*).

Como ejemplo a continuación se define una enumeración con los pisos correspondientes a un edificio (tipo entero):

```
<element name="Calle" type="xsd:string" />
```

```
<element name="Piso">
  <simpleType base="int">
    <enumeration value="1" />
    <enumeration value="2" />
    <enumeration value="3" />

    <enumeration value="4" />
  </simpleType>
```

```

</element>

<element name="Puerta">
  <simpleType base="xsd:string">
    <enumeration value="A"/>
    <enumeration value="B"/>
    <enumeration value="C"/>

  </simpleType>
</element>

...

<direccion>
  <Calle>El olmo roto</Calle>
  <Piso>3</Piso>
  <Puerta>B</Puerta>
</direccion>

```

3.4.1.1.3. Arrays de bytes

Como arrays de bytes se consideran datos de tipo binario como serializaciones de objetos, beans, etc.. o porciones de archivos como imágenes, etc.

Para el empaquetado de este tipo de datos se aconseja el uso de la codificación base64 también usada en empaquetados MIME (Multipurpose Internet Mail Extensions)

3.4.1.1.4. Estructuras

Una estructura es una colección de valores que son accedidos mediante distintos nombres. Además cada uno de los elementos puede tener distinto tipo (no es obligatorio).

3.4.1.1.5 Referencias

SOAP soporta la representación de referencias, que son punteros a otros datos. Este tipo normalmente se utiliza para la generación de datos complejos.

Las referencias apuntan al identificador del elemento referenciado.

```

<m:RestoDomicilio id="add-2">
  <Portal>67</Portal>

  <Piso>5</Piso>

  <Puerta>F</Puerta>
</m:RestoDomicilio>

<m:Direccion id="add-1">
  <Tipo>Avenida</Tipo>
  <NombreCalle>San Sibilio</ NombreCalle>

  <RestoDomicilio href="#add-2"/>
</m:Direccion>

...

```

```
<m:Socio>
  <Nombre>My Life and Work</Nombre>
  <Direccion href="#add-1"/>
</m:Socio>
```

Las referencias también se usan para evitar la escritura redundante de datos. Suponga que tiene un documento con los participantes de una olimpiada universitaria, como es lógico varios alumnos pertenecen a la misma universidad, por lo que se podría guardar la información de la universidad como un dato “externo” a los alumnos y hacer uso de las referencias para enlazarlos.

Además este tipo de referencias se usa en el caso que un mismo dato se necesite llamar desde varias zonas del documento. Si se tiene un objeto llamado persona, conteniendo los datos personales de un socio, y se debe de comprobar varias cosas como su dirección, su teléfono, etc.. se puede usar la siguiente notación para no tener que escribir el elemento *<persona>* en cada una de las llamadas (*<checkCodigoPostal>*, *<checkTelf>* y *<insertarBaseDatos>*):

```
<m:persona xmlns:m="urn:miURL" id="personal">
  <m:nombre xsi:type='xsd:string'>Joaquin Ternet</m:nombre>
  <m:direccion xsi:type='xsd:string'>
    Av. Antigua</m:direccion>
  <m:ciudad xsi:type='xsd:string'>Murcia</m:ciudad>
  <m:telefono xsi:type='xsd:string'>676-334532</m:telefono>
</m:persona>
...
<m:checkCodigoPostal xmlns:n="urn:miURL">
  <m:persona href="#personal"/>
</m:checkCodigoPostal>
<m:checkTelf xmlns:n="urn:miURL">
  <m:persona href="#personal"/>
</m:checkTelf >
<m:insertarBaseDatos xmlns:n="urn:miURL">
  <m:persona href="#personal"/>
</m:insertarBaseDatos>
```

El uso del atributo *href*, permite utilizar elementos definidos en otras partes del documento, en otros documentos e incluso en documentos que no estén alojados en la misma máquina, usando protocolos como el HTTP, valdrá con indicar el URL del documento que contenga la información.

3.4.1.2. Arrays

Los arrays o arreglos, son colecciones de datos de semejante tipo y bajo un mismo nombre y, que normalmente, son accedidos a través de su número de colocación en la estructura. Además el array viene especificado por el tipo: "SOAP-ENC:Array".

Los arrays son constituidos por secuencias de elementos enumerados, pero además pueden contener un nombre, de tal forma, que dichos elementos pueden ser accedidos bien por su número de colocación en el array o bien por su nombre.

Un ejemplo de array serían los hijos de una familia:

```
<hijos xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[4]">
<hijo xsi:type="xsd:string">Manuel</hijo>
<hijo xsi:type="xsd:string">Sandra</hijo>
<hijo xsi:type="xsd:string">Laura</hijo>
<hijo xsi:type="xsd:string">Nuria</hijo>
</hijos>
```

En el ejemplo se detalla un array de cuatro valores, con cuatro elementos de tipo cadena (string). Se indica en la definición que los elementos forman parte de un array: `xsi:type="SOAP-ENC:Array"` y además se define también que los elementos del array serán del tipo string:

```
SOAP-ENC:arrayType="xsd:string[4]" .
```

Para transmitir arrays se puede usar dos tipos de codificaciones:

- anónima
- con nombre

La diferencia es clara, en el primer caso los elementos irán sin nombre alguno en su transmisión y simplemente se accederán en el orden en que aparecen en el documento SOAP, mientras que en el segundo, se especifica un nombre en el elemento.

Suponga que se trabaja con datos sobre poblaciones, que están compuestos por un array de dos elementos, en el primero se almacenará la provincia y en el segundo se almacenará el municipio. Si lo hace nombrando los elementos podría usarse:

```
<poblacion>
  <provincia xsi:type="xsd:int">47</provincia>
  <municipio xsi:type="xsd:int">134</municipio>
</poblacion>
```

En este caso está claro que elemento es la provincia y cual es el municipio, si se utilizara el método anónimo, el resultado podría ser algo como:

```
<poblacion>
  <xsi:type="xsd:int">47</xsi:type="xsd:int">
  <xsi:type="xsd:int">134</xsi:type="xsd:int">
</poblacion>
```

En este caso se debe acceder a los datos de la población por orden; de esa manera, si se cambian de orden los datos, la aplicación no tendría manera de controlar que dato es el que está leyendo. Esta forma de codificar los datos se usa sobre todo en llamadas a funciones.

Se puede también definir el arreglo sin tener que indicar el tipo en cada uno de los elementos que lo componen:

```
<hijos xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[2]">
    <hijo>Manuel</hijo>
    <hijo>Sandra</hijo>
</hijos>
```

3.4.1.2.1 Array de arrays

Los arrays no tienen por que ser unidimensionales, sino que pueden ser multidimensionales, estos son arrays que sus elementos contienen a su vez otros arrays:

```
<matriz xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:int[2, 2]">
    <elemento>13</elemento>
    <elemento>17</elemento>
    <elemento>2</elemento>
    <elemento>96</elemento>
</matriz>
```

El listado anterior pertenece a la representación de una matriz 2 x 2.

Como se puede apreciar, la diferencia entre una matriz unidimensional y multidimensional, está en la declaración `SOAP-ENC:arrayType="xsd:int[2, 2]"`, esto significa que se va a hacer un array de dos elementos, que cada uno de ellos contendrá otros dos elementos, si se quisieran indicar tres elementos que contengan a otros dos, se indicaría `[3,2]`.

Hay que tener en cuenta la forma en la que se llenan los elementos del array; el primer elemento será el `[1,1]`, el segundo será el `[1,2]`, el tercero será el `[2,1]` y el último pertenecerá al `[2,2]`

3.4.1.2.2. Arrays sin límites

En muchas ocasiones es muy difícil e incluso imposible conocer el tamaño de un array, en este caso se usan arrays sin límites. Para indicar un array de este tipo se dejará sin rellenar el número contenido entre los corchetes.

```
<matriz xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:array[2][ ]">
<elementoCompuesto
xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:String[2]">
    <elementoSimple>dato1</elementoSimple>
    <elementoSimple>dato2</elementoSimple>
</elementoCompuesto>
```

```

<elementoCompuesto
    xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:String[2]">
    <elementoSimple>dato3</elementoSimple>
    <elementoSimple>dato 4</elementoSimple>
</elementoCompuesto>
</matriz>

```

Esta representación significa que se tiene un array de dos elementos, cada uno de los cuales contiene un arreglo sin límite de elementos

Muchas veces no se usa este tipo de representación, sino que se usan referencias a los elementos compuestos, haciendo uso del atributo *href*, que hemos visto anteriormente.

```

<matriz xsi:type="SOAP-ENC:Array"
    SOAP-ENC:arrayType="xsd:string[2][ ]">
    <elementoCompuesto href="#ec1_2" />
    <elementoCompuesto href="#ec3_4" />
</matriz>
...
<elementoCompuesto id="ec3_4"
    xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:String[2]">
    <elementoSimple>dato1</elementoSimple>
    <elementoSimple>dato2</elementoSimple>
</elementoCompuesto>

<elementoCompuesto id="ec3_4"
    xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:String[2]">
    <elementoSimple>dato3</elementoSimple>
    <elementoSimple>dato 4</elementoSimple>
</elementoCompuesto>

```

En este ejemplo se puede ver como hacer referencia a elementos mediante el atributo *href*. Así en el ejemplo anterior, los bloques *<elementoCompuesto>* están definidos fuera de la estructura matriz, pero para la aplicación será como si estuvieran dentro, ya que se hallan referenciados por el atributo *href* de la matriz.

3.4.1.2.3. Arrays parciales

Para transmisiones incompletas de arreglos, SOAP permite la transmisión parcial de arrays, ya que en ocasiones es posible que sólo interese transmitir una parte del array sin perder la posición que ocupan los datos dentro del mismo.

Imagine que se tiene aplicación que controle una carrera de coches y que se mantienen en un array el nombre de los conductores, y al final de la carrera este array se ordena según la clasificación obtenida en línea de meta; si un periódico dice que quiere publicar el podium de ganadores, lo que no tiene sentido es transmitir todos los participantes, sino sólo los primeros (no quedaría bien si piden los tres primeros y como respuesta, se le devuelven todos los participantes).

En este caso se usaría un array parcial.

```
<corredores xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[100]">
    <nombre m:dorsal='15'>Samuel</nombre>
    <nombre m:dorsal='92'>Carlos</nombre>
    <nombre m:dorsal='02'>Benito</nombre>
</corredores>
```

Como ya sabe, es posible acceder a los elementos según el orden en el que se han definido en la estructura, por lo que queda claro que son los tres primeros, pero ¿qué pasaría si pidieran los tres últimos? En este caso no se pueden escribir los tres elementos sin más, ya que, o se especifica de alguna forma que son los tres últimos, o no hay manera de ver que realmente los corredores enviados son los que ocupan las últimas posiciones en el array.

Para poder transmitir estos últimos, se añade el atributo “*SOAP-ENC:offset*”. Este atributo muestra el número de elementos que hay que contar hasta el primer elemento transmitido. Por ejemplo:

```
<corredores xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[100]"
SOAP-ENC:offset="[97]">
    <nombre m:dorsal='15'>Samuel</nombre>
    <nombre m:dorsal='92'>Carlos</nombre>
    <nombre m:dorsal='02'>Benito</nombre>
</corredores>
```

3.4.1.2.4. Arrays referenciados elemento a elemento

Imagine ahora que por alguna causa el periódico pidiera los tres primeros corredores, de aquellos cuyo nombre comenzara por la letra J. En este caso se tiene que transmitir solamente partes del array, para ello se usa el atributo “*SOAP-EN:position*” en los elementos transmitidos indicando la posición que ocupan en el array :

```
<corredores xsi:type="SOAP-ENC:Array"
SOAP-ENC:arrayType="xsd:string[100]">
    <nombre m:dorsal='15' SOAP-EN:position="[15]">
```

```

        Juan
    </nombre>

    <nombre m:dorsal='15' SOAP-EN:position = "[35]">

        Javier
    </nombre>

    <nombre m:dorsal='15' SOAP-EN:position = "[56]">

        Jorge
    </nombre>

</corredores>

```

También es posible indicar la posición en arrays multidimensionales, como por ejemplo para indicar el elemento de fila seis columna siete:

```
<item SOAP-ENC:position="[5,6]">Elemento (6,7)</item>
```

3.4.1.3. Elementos nulos

Muchas veces las aplicaciones no tratan de la misma manera un elemento nulo que un elemento vacío, ya que realmente no es lo mismo. Por ejemplo, puede que una aplicación reaccione de manera diferente frente a un nulo que frente a una cadena vacía, o llevándolo a objetos serializables, está claro que se necesita alguna manera de hacerle llegar que el valor es un nulo.

Para poder indicar que el elemento es nulo se utiliza el atributo *nil*. En el siguiente ejemplo, se muestra una cadena nula:

```
<elementoNulo xsi:type="xsd:string" xsi:nil="true" />
```

Otra manera de representar un elemento nulo, es no introducirlo, “no escribirlo”, pero esto dependería del esquema de validación que se esté utilizando en el documento, ya que a veces no es posible realizarlo de esta última forma.

3.4.1.4 Elementos defecto

Al analizar un documento puede encontrarse con que no existe un elemento específico. En tal caso se puede tomar su valor por defecto.

El valor por defecto depende de la situación y del tipo del dato que se intentaba recuperar, por ejemplo, si el dato era del tipo booleano, se suele tomar como valor falso, si era de tipo entero, se suele tomar como valor cero o desconocido, y en caso de ser cadena o tipo complejo, se toma como valor nulo, aunque todos estos valores se pueden definir en el documento de validación como se explicó en el tema referente al lenguaje XML.

3.5. Respuestas

Ya conoce que cuando se usa el estilo SOAP-RPC, el receptor de la petición actuará como emisor del mensaje respuesta. El mensaje respuesta será de la misma forma que la petición, es decir usará la misma estructura, y normalmente la misma versión de codificación.

Si bien no existe ninguna obligación en cuanto los nombres usados en el documento de respuesta, si existe una convención, la de añadir la palabra “Response” al nombre de la función que ha provocado la respuesta. Por ejemplo si se tiene una petición:

```
<m:Envelope xmlns:m="http://www.w3.org/2001/6/soap-envelope">
```

```

xmlns:e="http://www.acme.org/stoks">

<m:Body>

    <e:obtenerStockLibro
m:encodingStyle="http://www.w3.org/2001/6/soap-encoding">

        <e:codigoLibro xsi:type="xsd:string">HHJDL48SJY</e:codigoLibro>

        <e:codigoEditorial xsi:type="xsd:int">17</e:codigoEditorial >

    </e:obtenerStockLibro>

</m:Body>

</m:Envelope >

```

La respuesta podría ser algo parecido a :

```

<m:Envelope xmlns:m="http://www.w3.org/2001/6/soap-envelope"
xmlns:e="http://www.acme.org/stoks">
<m:Body>
    <e:obtenerStockLibroResponse
m:encodingStyle="http://www.w3.org/2001/6/soap-encoding">
        <e: response xsi:type="xsd:int">17</e:response>
    </e:obtenerStockLibroResponse>
</m:Body>
</ m:Envelope >

```

Como se aprecia, en la respuesta se incluye el nombre de la función a la que se ha llamado (en este caso *obtenerStockLibro*), pero acabada con la palabra *Response*. Aunque se trata de una convención y no es de uso obligatorio, si que es conveniente su utilización. En todo caso, queda en manos del cliente interpretar de forma correcta el documento.

Si se “escucha” el puerto de servicio HTTP durante una petición en la que se use HTTP como protocolo de transmisión, se pueden ver mensajes semejantes a los que se han mostrado en este capítulo, por ejemplo:

```

POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

```

```

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

```

    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Y la respuesta:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

```

```

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <Price>34.5</Price>
    </m:GetLastTradePriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

En el capítulo seis se verán unas aplicaciones para permitir escuchar los puertos y poder ver la forma real de los mensajes utilizados en los ejemplos.

3.6. Faults

Como en todo proceso, durante las transacciones SOAP, se pueden dar errores tanto en el proceso de la petición (en la ejecución del objeto que atiende la llamada) como en el análisis del documento (fallo al analizar, por ejemplo por versión incorrecta de SOAP).

Los errores en SOAP son denominados *faults*. Cuando aparece un elemento `<Fault>`, debe aparecer dentro del `<Body>` y debe aparecer como máximo una vez.

El elemento `<Fault>` define a su vez cuatro subelementos:

- faultcode

Es utilizado por el software como mecanismo para identificar el origen del fallo. Este subelemento debe estar obligatoriamente dentro de cada elemento `<fault>`, y tiene que estar definido mediante *namespace*. El `<faultcode>` puede ser uno de los definidos por SOAP o bien algún otro definido por el programador. Más adelante se verán los distintos `<faultcodes>` que proporciona SOAP.

- faultstring

Es la explicación del código del error, está pensado como información para el humano, no para el proceso. Tiene que estar presente en cada elemento `<fault>`.

- faultactor

Es usado para dar información acerca de quien es el causante del error. Semejante al atributo *actor*, pero en lugar de indicar el destinatario del elemento, indica el generador del error, el valor es un URI.

- details

Se utiliza para dar información específica de la aplicación que causó el error durante el proceso del `<Body>`, por lo que tiene que estar presente si y sólo si el error se produce dentro del `<Body>`. La información referente a fallos en la cabecera, debe ir en la propia cabecera, no se debe incluir en el elemento `<Body>`.

Su ausencia en el elemento `<fault>` indica que el error no se produjo en el bloque `<Body>`.

Dentro del elemento `<details>` se pueden definir nuevos bloques para ofrecer una mejor información.

Los valores del elemento `<faultcode>` pueden ser definidos por SOAP o por el usuario. Las definiciones vienen dadas en forma de dirección de Internet, es decir separados por un punto (“.”), pero al contrario que las direcciones, el valor es más genérico cuanto más a la izquierda nos desplazemos. En el ejemplo siguiente se puede ver un error de autenticación. En el elemento `<faultcode>` se puede ver la manera en la que se definen estos, es decir, la parte más genérica queda a la izquierda, ya que “*Client*” es más genérico que “*Authentication*” (el fallo de autenticación pertenece al cliente), mientras que en las direcciones utilizadas en Internet, la parte más genérica se sitúa a la derecha, por ejemplo en *www.acme.com*, lo más genérico es “*com*” que señala que es una compañía, luego “*acme*” que indica el nombre de ésta y por último “*www*” para indicar que es su página web.

El ejemplo de mensaje retornado por de fallo de autenticación es el siguiente:

```
<s:Envelope xmlns="my-URI">

  <s:Header>

    <!--otros bloques-->

  </s:Header >

  <s:Body>

    <s:Fault>

      <faultcode>Client.Authentication</faultcode>

      <faultstring>

        Error en el password

      </faultstring>

      <faultactor>http://www.acme.net</faultactor>

      <details/>

    </s:Fault>

  </s:Body>

</s:Envelope>
```

3.6.1. Elementos faultcode estándar de SOAP

El identificador *namespace* de los elementos `<faultcode>` definidos por SOAP es “<http://schemas.xmlsoap.org/soap/envelope/>”, o “<http://www.w3.org/2001/06/soap-envelope/>”, dependiendo de la versión que se quiera utilizar, y sus valores

- VersionMismatch

Se encontró una versión del elemento “Envelope” no compatible con la esperada.

- MustUnderstand

El elemento de la cabecera que contenía el atributo *mustUnderstand* bien no se entendió o bien no cumplía las especificaciones del proceso

- Client

Indica que el mensaje no se había construido de manera correcta, o no contiene la información en el orden esperado. Se suele dar si un elemento depende directamente de otro que no ha sido encontrado.

- Server

Se usa para indicar que el mensaje no pudo ser procesado por causas ajenas al mensaje en si, pero sí relacionadas con él. Por ejemplo, si durante el proceso del mensaje se realizaran transacciones entre varios servidores y fallara alguno de éstos que además debiera atender alguna parte del tratamiento del mensaje. Sería semejante al error 500 del protocolo HTTP.

Por último se muestra un ejemplo de un error en el desempaquetado de un mensaje en el que el servidor no ha sido capaz de encontrar la forma de procesar un objeto MIME (una fichero) que se añadió al mensaje. Dentro del mensaje se puede observar en el elemento `<faultactor>`, el causante del error:

```
<?xml version='1.0' encoding='UTF-8'?>

<SOAP-ENV:Envelope

xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<SOAP-ENV:Header>

    <!-- bloques de la cabecera-->

</SOAP-ENV:Header>

<SOAP-ENV:Body>

    <SOAP-ENV:Fault>

        <faultcode>SOAP-ENV:Client</faultcode>

        <faultstring>

            No Deserializer found to deserialize a
            'urn:mimeprocessor:file' using encoding style 'null'.

        </faultstring>

        <faultactor>/soap/servlet/rpcrouter</faultactor>

    </SOAP-ENV:Fault>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

3.6.2. Elementos faultcode del usuario

Aparte de los códigos de error que ya ofrece SOAP, se permite además la definición de códigos propios de cada programador. Como requisito indispensable está el hecho de que se deben usar *namespaces* en su definición. En el siguiente ejemplo, se muestra un mensaje en el que el código de error, es uno definido por el usuario, se puede ver cómo está definido el *namespace* mediante *cf*.

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/" ">
```

```

<s:Header>

  <!--otros bloques-->

</s:Header >

<s:Body>

  <s:Fault xmlns:cf="mi_URI">

    <faultcode>cf:MiNuevoFault</faultcode>

    <faultstring>

      No se puede acceder desde esta IP

    </faultstring>

    <faultactor>http://www.acme.net</faultactor>

    <details/>

  </s:Fault>

</s:Body>

</s:Envelope>

```

Aunque en ocasiones los faults estándar de SOAP se quedan cortos en la descripción, o no se adapta bien al fallo, es conveniente perder algo de información y usar los *faults* ya definidos por SOAP, para evitar las incompatibilidades.

3.7. SOAP con Attachments

Anteriormente se ha visto que SOAP puede serializar datos binarios usando codificación *base64*, al igual que se hace con los mensajes MIME, pero en ocasiones, el dato binario se encuentra en forma de archivo, por lo que para poder transmitirlo como un dato más en el cuerpo del mensaje, habría que realizar un buffer, al que se le iría añadiendo la información leída del fichero, y una vez leído aplicar el empaquetado y transmitir. Esta forma de hacerlo, aunque es perfectamente válida, pero suele consumir muchos recursos, por lo que no es la manera más idónea de realizar esta transmisión.

Existe otra manera, que pese a no estar muy extendida al principio, va tomando protagonismo por las posibilidades que ofrece, es el llamado SOAP con *attachments*. Un *attachment* es un “pegado”, un anexo al mensaje, esto es, se tiene un mensaje SOAP normal, como los que se han visto hasta ahora, y en él una referencia a un añadido que irá codificado tras el mensaje.

La manera en la que se realiza este empaquetado, es la misma que en otras áreas de Internet como el envío de ficheros binarios mediante formularios HTML, y es la *Multipart/Related MIME media type* (documento multiparte con tipo MIME asociado) que está recogida por la RFC 2387.

Para realizar este tipo de transmisiones se debe utilizar SOAP 1.1 (o superior).

Los elementos que vayan pegados al mensaje deberán tener un identificador MIME (Content-ID), o bien un localizador. En el siguiente listado se puede ver una cabecera de un MIME que hace referencia a un fichero, y se ha usado un localizador para su codificación.

```
Content-Type: image/jpeg
```

```
Content-Transfer-Encoding: binary
```

Content-ID: <http://midireccion.com/koko.jpg >

Content-Location: http://midireccion.com/koko.jpg

Las cuatro formas de poder apuntar a un mime son:

- 1.- Mediante un Content-ID, o identificador de parte MIME.
- 2.- Mediante un Content- Location relativo (localizador de contenido relativo).
- 3.- Mediante un Content- Location relativo sin base URI
- 4.- Mediante un Content- Location absoluto.

La primera de ellas es un identificador que se ha formado partiendo de la dirección de Internet donde se encuentra el fichero y de su nombre.

La segunda, se realiza a partir del nombre del fichero, y la dirección se obtiene de la dirección de donde proviene el documento raíz.

La tercera mantiene en el *Content-Location* el nombre del fichero, pero el identificador apunta directamente a la estructura interna del mensaje multiparte.

Por último, la cuarta como su propio nombre indica, se forma a través de su URL.

Aunque no es obligatorio para el SOAP primario (que tiene que ser el elemento raíz del documento multiparte), es aconsejable el uso de identificadores en él. Donde sí es obligatorio el uso de identificadores es en el resto de las partes, sean estas datos binarios u otros documentos XML.

Tanto los elementos de la sección *<Header>* como los elementos de la sección *<Body>* del mensaje SOAP, necesitan hacer referencias hacia las otras partes del mensaje MIME multipart. Para ello se puede usar mecanismo ya visto en ejemplos anteriores, y es mediante una referencia a un URI a través del atributo *href*. La forma en la que se resuelven estos URI es equivalente a la que se usa en otros documentos como HTML, esto es, las URI se transforman en direcciones absolutas, y éstas son las que se procesan.

Los mecanismos de transformación de direcciones relativas a absolutas se escapan del alcance de este libro, pero el lector los puede encontrar en la RFC 2396 .

Para acabar con este apartado, se incluye un ejemplo donde se puede ver como son referenciados los distintos archivos que se transmiten de forma adjunta al mensaje.

MIME-Version: 1.0

Content-Type: Multipart/Related; boundary=MIME_boundary;
type=text/xml;

start="<http://midireccion.com/xml/test.xml>"

Content-Description: SOAP con attachments.

--MIME_boundary

Content-Type: text/xml; charset=UTF-8

Content-Transfer-Encoding: 8bit

Content-ID: <http://midireccion.com/xml/test.xml>

Content-Location: http://midireccion.com/xml/test.xml


```
<?xml version='1.0' ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <!--añadimos los enlaces a los gráficos adjuntados-->
    <!--mediante Content-ID-->
    <primerGraf href="cid:camiseta.jpg@midireccion.com"/>
    <segundoGraf href="http://midireccion.com/koko.jpg"/>
    ..
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
--MIME_boundary--
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <camiseta.jpg@midireccion.com>
```

```
...binary JPEG image...
```

```
--MIME_boundary--
```

```
--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <http://midireccion.com/koko.jpg>
Content-Location: http://midireccion.com/koko.jpg
```

```
...binary JPEG image...
```

```
--MIME_boundary--
```

En el capítulo seis se verá un ejemplo realizado en lenguaje Java, en el que se utiliza esta forma de generación de mensajes SOAP para transmitir un fichero desde cliente al servicio web.

El documento WSDL

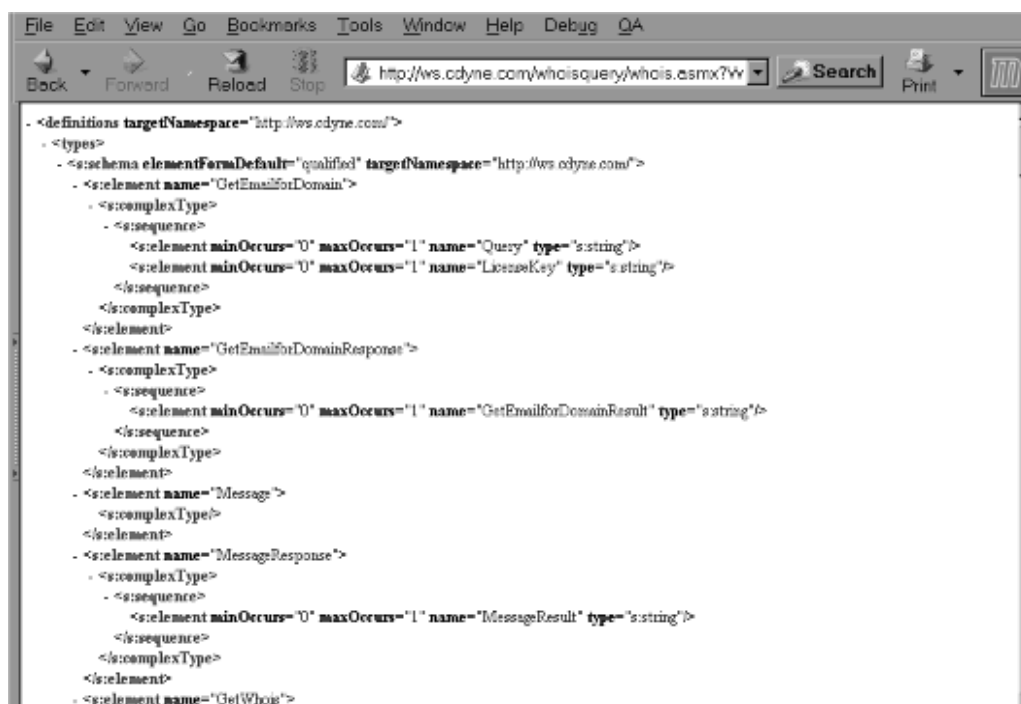
4.1 El documento WDSL

El programador necesita saber cómo está constituido el web service para poder codificar un cliente capaz de invocarlo, debe conocer su API, que servicios ofrece, con que parámetros, etc... además es necesario conocer que tipo de protocolo de codificación se usará. Tampoco se debe pasar por alto que este servicio puede estar alojado en cualquier parte del planeta, por lo se debe indicar también la dirección (con puerto y protocolo) donde debe ser invocado. Le surge pues al programador la necesidad de poseer un documento informativo con las características técnicas del servicio para poder crear el código del cliente que lo llamará.

Todas estas necesidades y algunas más se cubren mediante un documento llamado WSDL, que es el acrónimo de Web Service Description Language (o lenguaje de descripción del web service).

Este documento es la herramienta para poder describir la funcionalidad y modo de acceso al servicio. El lenguaje WSDL está también basado en XML y mediante su análisis es posible conocer infinidad de datos acerca del servicio, como pueden ser el nombre de las funciones disponibles, nombres y tipos de dato de los parámetros que han de recibir estas funciones, tipo de respuestas producen, cómo será devuelta esta respuesta, etc.

Los documentos WSDL están disponibles al público en general (siempre que lo esté el servicio; los servicios privados no tienen sus WSDL a la vista de cualquiera, como es lógico), y pueden ser accedidos por protocolos estándar de Internet, como el HTTP. Lo más normal es encontrar estos WSDL en webs especializadas, que funcionan como un buscador de Internet normal (como por ejemplo en <http://www.xmethods.net/>), también podemos encontrarlos en registros UDDI (se explicará en el siguiente capítulo).



```

- <definitions targetNamespace="http://ws.cdyne.com/">
  <types>
    <!-- schema elementFormDefault="qualified" targetNamespace="http://ws.cdyne.com/" -->
    <!-- element name="GetEmailforDomain" -->
    <!-- complexType -->
    <!-- sequence -->
    <!-- element minOccurs="0" maxOccurs="1" name="Query" type="s:string" -->
    <!-- element minOccurs="0" maxOccurs="1" name="LicenseKey" type="s:string" -->
    </sequence>
    </complexType>
  </element>
  <!-- element name="GetEmailforDomainResponse" -->
  <!-- complexType -->
  <!-- sequence -->
  <!-- element minOccurs="0" maxOccurs="1" name="GetEmailforDomainResult" type="s:string" -->
  </sequence>
  </complexType>
  </element>
  <!-- element name="Message" -->
  <!-- complexType -->
  </element>
  <!-- element name="MessageResponse" -->
  <!-- complexType -->
  <!-- sequence -->
  <!-- element minOccurs="0" maxOccurs="1" name="MessageResult" type="s:string" -->
  </sequence>
  </complexType>
  </element>
  <!-- element name="GetWhois" -->

```

Figura 6: Documento WSDL accedido mediante navegador.

El registro UDDI (UDDI Business Registry, registro de negocios para descripción universal, descubrimiento e integración) es una base de datos a nivel mundial gestionada por una alianza de empresas (IBM, Microsoft,...), que permite búsquedas por varios conceptos distintos y que se puede usar

libremente para obtener información no solamente de web services, sino de las empresas que los proporcionan, de sus negocios...

Debido a la importancia del tema de localización de servicios y ficheros WSDL, se tratará con mayor detenimiento en un capítulo posterior, donde se verá el uso de registros UDDI y de ficheros WSIL.

Como ejemplo de una consulta mediante protocolo HTTP de un documento WSDL, se puede acceder a la dirección <http://ws.cdyne.com/whoisquery/whois.asmx?WSDL> donde es posible ver el documento WSDL representado en la figura 6.

4.2. Importancia del WSDL

Aunque es aconsejable el uso de WSDL, no es obligatorio. Si se tiene una serie de web services en una red privada, se puede guardar perfectamente un registro con las características de los servicios en una base de datos, o simplemente no guardar nada, si es una red privada, es posible que se sepa en qué dirección y puerto están alojados y qué servicios ofrecen; pero es aconsejable su uso, porque como se verá en el capítulo seis en un ejemplo de servicio web, facilita muchas tareas, entre ellas, la codificación del cliente.

Uno de los aspectos más interesantes que presentan los web services, es que pueden llegar a ser autodescritivos, lo que permitirá automatizar tareas como la creación de clientes. La manera de conocer la forma de acceder a un objeto, y por tanto a sus funcionalidades, es mediante su API. El programador debe conocerla para saber qué servicios puede ofrecer dicho objeto y cómo se deben de llamar, no vale simplemente con saber sólo el nombre del servicio, sino también será necesario saber la forma de su llamada, para ello existen las interfaces.

La interfaz explica como se debe llamar a cada función, con que parámetros y en que orden. Una vez conocida esta API, ya se puede codificar el cliente, pero aunque esta tarea parezca sencilla, hay veces que se complica, y se deben realizar documentos explicativos para que el programador encargado de realizar la tarea, pueda comprender la interfaz, y así realizar correctamente la codificación del cliente.

El uso de documentos WSDL en web services permite que éstos puedan ser interpretados por las aplicaciones, y a éstas generar automáticamente el interfaz, agilizando la tarea de codificación de clientes, y evitando errores de transcripción.

Aunque WSDL es el lenguaje de descripción de web services más usado y más extendido, no es el único, y además presenta grandes carencias como la falta de control de versiones. En WSDL no hay manera de ver si el proveedor del servicio, ha realizado cambios en la interfaz de entrada (y por consiguiente en el WSDL), por lo que se deben enviar mensajes a los clientes avisando de ello (esta desventaja está previsto cubrirla en revisiones posteriores del WSDL), aunque el uso de un cliente sobre una interfaz que no tiene la misma morfología, reportaría un error.

Existen otros lenguajes de descripción de web services más completos que WSDL, pero no están tan difundidos como éste, bien por ser más nuevos o por ser demasiado complejos. Alguno de estos lenguajes son DAML (DARPA Agent Markup Language) y RDF (Resource Description Framework)

Las ventajas del uso del WSDL no quedan “sólo” en tener perfectamente definido la interfaz hacia nuestro servicio, sino que van más allá. Por medio de la utilización de WSDL obtendremos las siguientes ventajas:

- Se reduce el coste de los cambios en los clientes, permitiendo actualizaciones casi automáticas, y reduciendo drásticamente los tiempos entre codificación y producción.
- Ayuda a la estructuración de objetos, y facilita la construcción de éstos.
- Se hace “atractivo” el uso de web services, por su facilidad de implementación y herramientas de automatización.
- La generación del WSDL también puede automatizarse, existiendo herramientas para distintos lenguajes, encargadas de dicha tarea.

Todo aquel programador que haya trabajado con objetos remotos de terceras personas, comprenderá que los cambios que se realizan en éstos, pueden llegar a ser bastante molestos para el programador, por

tener que revisar la información del nuevo objeto mirando los cambios que podrían llegar a afectarle a su código. Además, también es muy engorroso tener que hacer la documentación de los cambios efectuados y de la nueva interfaz del objeto. Con el documento WSDL tendremos herramientas (en casi todos los lenguajes) tanto para generar automáticamente el código fuente de los clientes, como para generar el propio documento a partir del código fuente del objeto, con el ahorro de tiempo (y por consiguiente de dinero) que esto supone. Un ejemplo de la posibilidad de realizar un cliente adaptable a cada WSDL es la aplicación de la página web <http://www.soapclient.com/soaptest.html>.

Aunque existen herramientas para la creación de los WSDL, no está de más conocer su estructura y funcionamiento para realizar ajustes manualmente si se diera el caso (por ejemplo cambiar las direcciones de acceso).

La interfaz de un web service, es como el resto de los interfaces que se pueden encontrar en otros lenguajes como Java o C++, salvando la diferencia de que, como estamos en un entorno de trabajo guiado por mensaje, también deberá definir éstos y su orden.

4.3. Descripciones del WSDL

El documento WSDL es eminentemente descriptivo, y su finalidad es la de dar toda la información posible al programador (o al programa analizador) para poder realizar la codificación del cliente que accederá al servicio o servicios descritos en él.

Una interfaz, tiene la misión de dar a conocer la forma que tendrá un objeto visto desde fuera, conociendo sus “enganches” con el resto del sistema, pero no tiene que dar información de qué o cómo se realizan las operaciones por dentro. No importa que lenguaje sea, la misión será siempre la misma. Lo que tienen de especial estas interfaces es que deben ser interpretables por innumerables lenguajes y tecnologías distintas, por lo que se tiene que dejar totalmente definidos todos los parámetros que puedan surgir, tales como número de mensajes que toman parte en la transacción, tipos que se usan, protocolo de transmisión...

Realmente en el conjunto de los web services, el documento WSDL trata de describir seis grandes elementos, que son los siguientes:

- **types (tipos):**
Se utiliza para describir los tipos que tomarán parte a lo largo del documento, más específicamente en los mensajes de intercambio, los tipos de los datos que se transmitirán en estos mensajes. Estos tipos se definen usando herramientas como el XML Schema (ver capítulo dos).
- **message (mensaje)**
Son colecciones de datos, representaciones abstractas de los datos transmitidos. Cada mensaje se compone de varias partes lógicas, cada una de las cuales tiene un tipo que tiene que estar previamente definido.
- **portType (tipo del puerto)**
Es una representación abstracta del servicio, indicando el orden de intercambio de mensajes y es físicamente, la interfaz real del servicio.
- **binding**
Es la definición concreta de los `<portType>`, indicando qué protocolo y qué tipo de transporte se va a utilizar en la transmisión, como por ejemplo HTTP.
- **ports (puertos)**
Direcciones implementando el servicio. Se presenta de dos formas, los llamados `<portType>`, y los `<binding>` (enlaces).
- **service (servicio)**
Es una colección de puertos.

4.4. Morfología

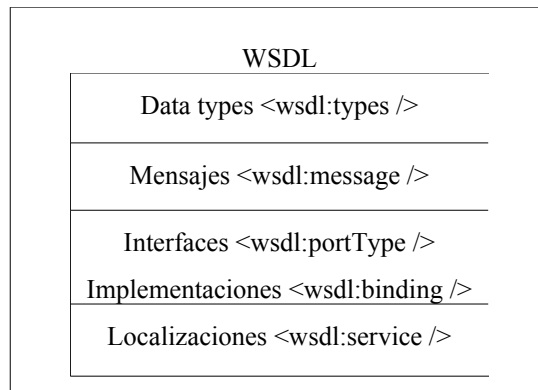
El fichero que compone el documento WSDL, es un documento escrito en texto plano, más concretamente usando XML, por lo tanto en su composición, tendrá un elemento raíz, que siempre deberá llamarse `<definitions>`. Dentro del elemento `<definitions>` se encuentran el resto de elementos portadores de datos referentes al web service (o a varios de ellos).

Como todo documento XML, también deberá cumplir unas reglas de composición de datos. Estas reglas vienen fijadas por el *namespace* `http://schemas.xmlsoap.org/wsdl/`.

El documento WSDL tiene cuatro partes bien diferenciadas:

- Preámbulo
- Descripción de la traducción de método a mensaje
- Información de mensajes
- Localización del servicio

Formando una estructura semejante a:



Estructura de los documentos WSDL

4.4.1. Preámbulo

El preámbulo servirá para introducir el servicio, indicando los tipos de datos usados en las distintas transacciones, así como una presentación de los mensajes que se darán durante las mismas, sin olvidar la declaración de los *namespace* que se van a utilizar.

Como lenguaje para la descripción y especificación de datos usados a lo largo del WSDL, se usa principalmente el XML Schema (ver capítulo 2). Para ligar la definición de los datos al WSDL se puede optar por dos opciones, o bien definir los datos incrustándolos en el preámbulo del documento, mediante el elemento `<wsdl:types ... />` (como se verá seguidamente en ejemplos posteriores) o bien se puede generar la definición en un archivo independiente y luego incorporarlo al documento WSDL mediante el elemento `<wsdl:import />`. La recomendación dada por los proveedores de herramientas de generación de WSDL es que no se use esta segunda forma, puesto que existen muchas implementaciones en la que no se da soporte o que bajo ciertas condiciones no se realiza correctamente la importación, por lo que sólo debe ser utilizada si no es posible el uso del primer método.

El esquema del preámbulo es:

```

<wsdl:definitions name="nmtoken"? targetNamespace="uri">

  <import namespace="uri" location="uri"/> *

  <wsdl:documentation .... /> ?
  
```

```

<wsdl:import ..... /> *

<wsdl:types> ?

    <wsdl:documentation .... /> ?

    <xsd:schema ..... /> *

</wsdl:types>

<wsdl:message name="ncname"> *

    <wsdl:documentation .... /> ?

    <part name="ncname" element="qname"? type="qname"?/> *

</wsdl:message>

```

4.4.2. Descripción

El elemento que se encuentra en esta sección es el `<portType>`. Deberá haber uno por cada servicio que se quiera concretar. Dentro de él se definirán las operaciones que se puedan realizar en su llamada, o dicho de otra forma las funciones públicas que soporta, las funciones a las que puede acceder un cliente.

Las funciones son llamadas mediante mensajes, y contestarán también a través de ellos, por lo cual dentro de cada elemento `<operation>`, se hace necesario definir los mensajes que se darán durante su llamada, y el orden en el que se pueden dar.

Los mensajes pueden ser de tres tipos:

- 1.- de input: mensajes que se emiten por parte del cliente como entrada al servicio.
- 2.- de output: mensajes que son emitidos por el servicio, normalmente como respuestas a los mensajes de input, y que será donde se contengan los datos de respuesta en caso de ser una consulta.
- 3.- de fault: mensajes que se emiten bajo condiciones de error.

En los elementos `<portType>` se deben definir las interfaces que se harán disponibles mediante este documento WSDL, en ellos se especifica el tipo de los mensajes que se van a utilizar, además se les dará un nombre. Este nombre será utilizado más adelante, en las siguientes secciones del WSDL, para generar referencias a estos elementos.

Su esquema es:

```

<wsdl:portType name="ncname"> *

    <wsdl:documentation .... /> ?

    <wsdl:operation name="ncname"> *

        <wsdl:documentation .... /> ?

        <wsdl:input message="qname"> ?

            <wsdl:documentation .... /> ?

        </wsdl:input>

        <wsdl:output message="qname"> ?

            <wsdl:documentation .... /> ?

```

```

</wsdl:output>

<wsdl:fault name="ncname" message="qname"> *

    <wsdl:documentation .... /> ?

</wsdl:fault>

</wsdl:operation>

</wsdl:portType>

<wsdl:serviceType name="ncname"> *

    <wsdl:portType name="qname"/> +

</wsdl:serviceType>

```

4.4.3. Información de mensajes.

Aún quedan dos aspectos muy importantes por describir para el funcionamiento del cliente del web service: el enlace (elemento *<binding>*) que define como se transmitirán los mensajes y que protocolos usarán éstos y el servicio (elemento *<service>*), que se encarga de definir la dirección donde la interfaz está implementada, donde se encuentra físicamente.

En esta sección se explicará la primera, presentando la localización exacta del servicio en la siguiente sección.

En puntos anteriores se han definido mensajes que se usarán a lo largo de las llamadas, e incluso se ha reflejado la forma tendrán dichas llamadas, pero es necesario tener más información de ellas, como por ejemplo cómo se realizará el transporte, o cómo se realizará la codificación.

En realidad esta sección es bastante semejante a la anterior, ya que el enlace, lo único que hace es completar la información del elemento *<portType>*, por lo que se podrían haber definido ambas secciones de forma conjunta, fusionándolas en una sola, pero esto no es así, y se hace para obtener una mayor claridad. Si el lector inspecciona documentos WSDL de servicios con muchas llamadas, o con llamadas sobrecargadas, tendrá la oportunidad de comprobar que el elemento *<binding>* (enlace) se puede complicar bastante, y es posible que en ocasiones sólo interese saber el nombre de las llamadas disponibles, sin necesidad de saber su funcionamiento o protocolos que se deben usar, por lo que bastaría con mirar la sección *<portType>*, que no suele complicarse aunque las llamadas sean complejas.

La diferencia entre las definiciones del elemento *<portType>* y del *<binding>* está, en que dentro de la segunda (en el enlace), aparecen unos elementos (y sub elementos) que no aparecen en la primera, los más importantes son:

- *<documentation />*
Se puede utilizar para incorporar información acerca de cada uno de los elementos que pueden contenerlo (ver el XML Schema siguiente).
- *<soap:binding />*
Define el protocolo de transporte (mediante el atributo *transport*) y el tipo de mensaje SOAP que se va a emitir (atributo *style*). Como valor del atributo *style*, se usa normalmente el método *RPC* (Remote Process Control, control de proceso remoto), frente al otro método disponible llamado *document*, que sirve para transacciones EDI (Electronic Document Interchange), en las que se emitirán datos que no deben ser analizados.
- *<wsdl:operation />*
Dentro de este elemento se detalla la información de las funciones que están disponibles en la interfaz. El nombre de estas funciones debe coincidir con el nombre que se indicó en la sección

`<portType>`. Mediante el atributo `name` de este elemento, se indica el nombre de la función que se definirá.

- `<soap:body />`

Indica cómo tienen que aparecer los elementos del WSDL en el elemento `<Body>` del mensaje SOAP indicando si son tipos no codificados o bien, si por el contrario sí lo son y en tal caso dando las reglas de codificación utilizadas.

- `<soap:fault />`

Esta parte aparecerá si se quiere indicar que se realice una acción distinta en caso de cualquier tipo de error y especifica el contenido del mensaje que se recibirá en tal caso.

- `<soap:header />`

Es semejante al elemento `<soap:body>`, pero en este caso hace referencia a los elementos que podremos encontrar en la cabecera del mensaje SOAP.

- `<soap:operation />`

Distinto del elemento `<wsdl:operation>` definido anteriormente, se comprenderá la diferencia en un ejemplo posterior. Define el valor del elemento "SOAPAction" cuando se usa el protocolo HTTP. Sirve mediante el atributo `style` para conocer el tipo de mensaje que se dará.

Todos estos elementos ayudan a tener un mayor conocimiento del mensaje que se transmitirá, e incluso, las partes que lo compondrán.

Su esquema puede representarse de la siguiente forma:

```
<wsdl:binding name="ncname" type="qname"> *
  <wsdl:documentation .... /> ?
  <!-- binding details --> *
  <wsdl:operation name="ncname"> *
    <wsdl:documentation .... /> ?
    <!-- binding details --> *
    <wsdl:input> ?
      <wsdl:documentation .... /> ?
      <!-- binding details -->
    </wsdl:input>
    <wsdl:output> ?
      <wsdl:documentation .... /> ?
      <!-- binding details --> *
    </wsdl:output>
    <wsdl:fault name="ncname"> *
      <wsdl:documentation .... /> ?
      <!-- binding details --> *
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
```

4.4.4. Localización del servicio

Ya está perfectamente definido el acceso al web service, pero falta quizá lo más importante, saber dónde está, dónde hay que enviar las peticiones de servicio. Así será posible llegar a codificar clientes que descubran en tiempo de ejecución la localización del servicio al que van a llamar.

En esta última sección se explicará como indicar en qué lugar de la red se puede encontrar el servicio que hemos definido en los apartados anteriores. No hay que olvidar que en ocasiones no vale sólo con indicar una dirección, ya que por ejemplo si se usa como protocolo de transporte el HTTP, la llamada se producirá al puerto 80 de la dirección, y muchas veces no es ahí donde se encuentran los servicios, sino que se dedica otro puerto a este menester, por ejemplo el 8082. Por esta razón las direcciones usadas en este apartado suelen contener un número tras la dirección URL, separado de ésta mediante dos puntos

(":"), dicho número representa el número del puerto al que se conectará el cliente para realizar las peticiones Ej. <http://www.acme.com:8082>.

Aclarar que un servicio no tiene por que estar ligado a una sola dirección. Es posible definir diferentes direcciones para un mismo servicio con el simple hecho de definir varios puertos. Si se fija en la definición del esquema del elemento `<service>`, verá que se pueden definir varios elementos `<port>`. La línea correspondiente es:

```
<wsdl:port name="ncname" binding="qname"> *
```

El esquema completo de esta sección, sería:

```
<wsdl:service name="ncname" serviceType="qname"> *
  <wsdl:documentation .... /> ?
  <wsdl:port name="ncname" binding="qname"> *
    <wsdl:documentation .... /> ?
    <!-- address details -->
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

4.4. Ejemplo práctico

Quizá el mejor método para aclarar todos los puntos respecto al documento WSDL sea poner en práctica un ejemplo.

El problema que se plantea a la hora de poner ejemplos de programación de web services es qué lenguaje elegir. Como se ha dicho en capítulos anteriores, existen muchas implementaciones para muchos lenguajes, así pues hay que elegir entre alguno de los lenguajes disponibles. El lenguaje elegido para este ejemplo es Java.

El ejemplo será un sencillo programa, que, como no puede ser de otra manera, tiene que ser el “hola mundo”. En el capítulo seis, se explicará con más detenimiento este programa, y se realizarán clientes para él, pero en esta sección lo que interesa realmente es ver como quedan reflejados en el documento WSDL los distintos aspectos del servicio, tales como las funciones o los parámetros de éstas.

Se acordará al lector que se vio que no es lo mismo un mensaje de petición que uno de respuesta, por lo que se hará una pequeña modificación sobre el típico programa de hola mundo, pasándole un parámetro con el nombre de una persona, para obtener como respuesta: “Hola nombrePersona, soy un web Service”.

El código Java correspondiente es:

```
package primer;

public class DimeHola {

    public DimeHola() {

    }

    //función correspondiente al servicio

    public static String getHola(String quien){

        return "Hola " + quien + ", soy un web service"; // devolvemos el
saludo

    }

}
```

El listado es el correspondiente a una clase llamada *DimeHola*, que tiene definida una función que acepta como parámetro una cadena, y devuelve otra correspondiente a un saludo más la cadena pasada como parámetro.

Es de esperar que en el documento WSDL aparezca un servicio que tenga disponible una función que tenga como nombre *getHola* y que posea dos mensajes definidos, uno de petición, que se encargará de transmitir los parámetros de la función, y otro de respuesta con la contestación del saludo. Además se puede ver también que el elemento *<portType>* se denomina de la misma forma que la clase Java (*DimeHola*).

El WSDL del siguiente listado ha sido generado automáticamente por una herramienta, a partir de la clase Java presentada anteriormente, por lo que aparecerán más definiciones de las vistas anteriormente, sobre todo en la sección preámbulo. Esto lo hacen muchas empresas para mantener compatibilidades con otras de sus herramientas, como servidores de aplicación, etc...

```
<?xml version="1.0" encoding="UTF-8" ?>

<wsdl:definitions

    targetNamespace="http://holaholita"

    xmlns="http://schemas.xmlsoap.org/wsdl/"

    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"

    xmlns:impl="http://primer-impl"

    xmlns:intf="http://primer"

    xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"

    xmlns:wSDLsoap="http://schemas.xmlsoap.org/wsdl/soap/"

    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <types>

        <schema targetNamespace="http://primer"
            xmlns="http://www.w3.org/2001/XMLSchema">

            <complexType name="ArrayOf_SOAP-ENC_string">

                <complexContent>

                    <restriction base="SOAP-ENC:Array">

                        <attribute ref="SOAP-ENC:arrayType"
                            wsdl:arrayType="xsd:string[]" />

                    </restriction>

                </complexContent>

            </complexType>

            <element name="ArrayOf_SOAP-ENC_string" nillable="true"
                type="intf:ArrayOf_SOAP-ENC_string" />

        </schema>

    </types>


```

```
</schema>

</types>

<wsdl:message name="getHolaRequest">
  <wsdl:part name="quien" type="SOAP-ENC:string"/>
</wsdl:message>

<wsdl:message name="getHolaResponse">
  <wsdl:part name="return" type="SOAP-ENC:string"/>
</wsdl:message>

<wsdl:portType name="DimeHola">
  <wsdl:operation name="getHola" parameterOrder="quien">
    <wsdl:input message="intf:getHolaRequest"/>
    <wsdl:output message="intf:getHolaResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="DimeHolaSoapBinding" type="intf:DimeHola">
  <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getHola">

    <wsdlsoap:operation soapAction="" style="rpc"/>

    <wsdl:input>
      <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://primer" use="encoded"/>
    </wsdl:input>
  </wsdl:operation>
</wsdl:binding>
</wsdl:service>
</wsdl:definitions>
```

```

        </wsdl:input>

        <wsdl:output>

                <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://primer" use="encoded"/>

        </wsdl:output>

</wsdl:operation>

</wsdl:binding>

<wsdl:service name="DimeHolaService">

        <wsdl:port binding="intf:DimeHolaSoapBinding" name="DimeHola">

                <wsdlsoap:address
location="http://localhost:8080/axis/services/DimeHola"/>

        </wsdl:port>

</wsdl:service>

</wsdl:definitions>

```

Merece la pena comentar las partes anteriormente presentadas.

Las definiciones de los mensajes deben atender a la definición de la clase. Primero se deben exponer los mensajes como elementos independientes, para más tarde ligarlos a las funciones con las que estén relacionados.

Recalcar que ya es posible ver el nombre de las variables que se utilizarán, que en este caso coinciden con las definidas en el código fuente de la clase; aunque no es necesario que sea así, ayuda mucho cuando hay que retocar documentos WSDL a mano.

```

<wsdl:message name="getHolaRequest">

        <wsdl:part name="quien" type="SOAP-ENC:string"/>

</wsdl:message>

<wsdl:message name="getHolaResponse">

        <wsdl:part name="return" type="SOAP-ENC:string"/>

</wsdl:message>

```

La definición del elemento *<binding>*, muestra como se define el nombre del mismo (que será utilizado posteriormente). Es posible ver también que el estilo del documento es SOAP-RPC, y el transporte se realizará mediante el protocolo HTTP.

Dentro de él también se encuentra la operación definida en el código, que se llama *getHola*, al igual que en la clase anterior.

Los mensajes quedan encerrados entre las marcas de *<operation>*, y están definidos en el orden en el que se dan lugar. Además se define como *<input>* el mensaje de paso de parámetros y como *<output>* el mensaje correspondiente a la respuesta.

```
<wsdl:binding name="DimeHolaSoapBinding" type="intf:DimeHola">
    <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getHola">
        <wsdlsoap:operation soapAction="" style="rpc"/>
        <wsdl:input>
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://primer" use="encoded"/>
        </wsdl:input>
        <wsdl:output>
            <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://primer" use="encoded"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

Por último queda definido el lugar donde se almacenará el objeto que atenderá las llamadas, en este caso en la propia máquina.

```
<wsdl:service name="DimeHolaService">
    <wsdl:port binding="intf:DimeHolaSoapBinding" name="DimeHola">
        <wsdlsoap:address
location="http://localhost:8082/localservices/DimeHola"/>
    </wsdl:port>
</wsdl:service>
```

Está claro que este documento WSDL sólo servirá para clientes que se ejecuten en la misma máquina que la que aloja el servicio. Si se llamara desde otra máquina distinta, daría error, ya que la dirección haría referencia a ella misma. Para cambiar esto, se debería asignar una IP o una dirección fija en la dirección del *<port>*. Por ejemplo cambiar:

```
<wsdlsoap:address
location="http://localhost:8082/localservices/DimeHola"/>
por
```

```

    <wsdlsoap:address
location="http://mydireccion.com:8082/localservices/DimeHola"/>

```

Dentro de la sección `<service>` se puede definir más de un elemento `<port>`, esto puede servir para, por ejemplo, permitir varias implementaciones del mismo servicio, bien por tenerlas en distintos lenguajes, o por tenerlas alojadas en distintas máquinas, y así poder equilibrar la carga de peticiones, distribuyéndolas entre todas ellas.

A continuación se presenta una sección `<service>` que define cuatro puertos distintos, dos en cada máquina, con dos implementaciones en Java, una en Perl y otra en .NET, pero todas utilizando el mismo esquema de mensajes .

```

<wsdl:service name="HolaMundo">

    <wsdl:port name="HolaMundo_Java"

        binding="intf:DimeHolaSoapBinding">

            <soap:address location="http://localhost:8080/java"/>

        </wsdl:port>

    <wsdl:port name="HolaMundo_Perl"

        binding="intf:DimeHolaSoapBinding">

            <soap:address location="http://localhost:8082"/>

        </wsdl:port>

    <!-- Otras máquinas -->

    <wsdl:port name="HolaMundo_NET"

        binding="intf:DimeHolaSoapBinding">

            <soap:address

                location="http://miotramaquina.com/hw.asmx"/>

            </wsdl:port>

    <wsdl:port name="HS_Java"

        binding="intf:DimeHolaSoapBinding">

            <soap:address location="http://miotramaquina/java"/>

        </wsdl:port>

</wsdl:service>

```

Los lenguajes UDDI y WSIL

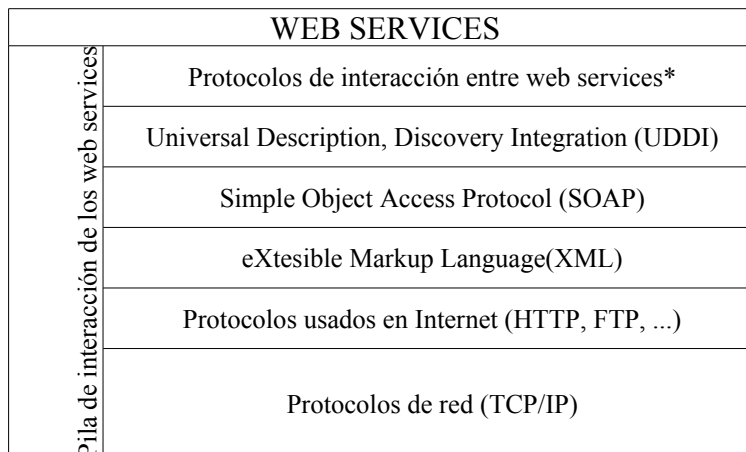
5.1 Introducción

Internet es algo vivo, está en continuo crecimiento, y esto queda claro cuando se miran las cifras referentes al mundo de Internet, por ejemplo, cada día (según la empresa Network Solutions) se incorporan a la red quince mil nuevos nombres de dominio.

Es más que posible que el lector haya tenido alguna vez dificultades para encontrar un documento en su propio ordenador, o en la Intranet de la empresa... está claro lo difícil que puede llegar a ser el encontrar un servicio concreto en Internet. Para facilitar la tarea, se puede utilizar un buscador de servicios. Pero si se tiene en cuenta el grado de automatización que ha alcanzado en los capítulos anteriores, donde los servicios eran incluso capaces de describirse a sí mismos, habrá que buscar la manera en la que los propios servicios sean capaces de realizar las tareas de publicación y búsqueda de forma automática o al menos semiautomática. Además el usuario debe tener la certeza que el servicio que está usando es realmente de la empresa que él quiere, y que no le está usurpando la información ninguna otra.

Si además se puede tener más información de la compañía a la que se está accediendo, mejor por que suponga que se hace uso de un servicio, y los resultados han sido muy satisfactorios, entonces, es posible el uso de métodos para descubrir más servicios (no necesariamente informáticos) de esta empresa, u otro tipo de informaciones como su dirección, teléfonos etc, y así favorecer la creación de nuevos clientes.

Con UDDI y WSIL se alcanza una cota más en la carrera hacia la consecución de un sistema basado en web services totalmente universal y automático.



Pila de protocolos y lenguajes en la estructura de llamadas de un web service.

- Ya se vio que es posible la interacción directa entre web services, puesto que cada uno de ellos puede actuar como cliente de otro web services, actualmente se están desarrollando nuevos protocolos para la comunicación entre ellos, haciendo así un entorno más potente, se verán algunos de estos protocolos en el capítulo nueve.

5.2 UDDI

Una de las maneras más fáciles y popularizadas de encontrar información en Internet, es mediante el uso de buscadores en forma de página web, como Google o Yahoo, pero en estos buscadores no siempre se encuentra lo que se está buscando, y mucho menos en los primeros intentos, por lo que se deben hacer varias consultas, y manualmente ir investigando si es la información que se está buscando o no. Si se

analiza la manera de funcionar de estos buscadores, uno se da cuenta que no se adaptará bien a las necesidades que se intentan cubrir en esta sección. Para la realización de la base de datos de un buscador, se suele proceder de dos maneras, una de ellas es dar la posibilidad al usuario de introducir su página web, con una pequeña descripción y las palabras clave para poder buscarla, y otra forma es mediante el uso de *crawlers* (gateadores), que son unos “investigadores” de páginas. Funcionan conectándose a las direcciones web, analizando el contenido de la página y haciendo una extracción de las palabras clave del código de ella (código HTML META). Además hay que decir que los *crawlers* están preparados para buscar páginas de contenido html, es decir, no indexan ni páginas sgml, ni xml ni, pl... Está claro que este método no se ajusta a las exigencias, aunque sí podría valer el primero de los métodos, en el que es el usuario quien se encarga de enviar la información.

Así existen páginas de búsqueda de servicios web con este funcionamiento (www.xmethods.net), en las que el usuario se conecta y envía la descripción de su web service (WSDL), y se publica en modo página web. Hay que recordar que dentro del WSDL se pueden introducir comentarios en cada una de las secciones (ver definiciones en el capítulo anterior), por lo que se puede explicar en ellas el modo de uso del servicio. Pero analizando este sistema uno se da cuenta que tiene varias carencias en cuanto al mundo del web service se refiere. Piense en los aspectos que se detallan a continuación:

- Existen una gran cantidad de acciones automatizadas, por lo que las nuevas deben ser lo más automáticas posibles.
- La búsqueda debe usar protocolos de transporte ya establecidos, no merece la pena realizar un protocolo específico, pudiendo usar los estándares. Al usar estándares, permite llegar a más clientes y facilita la implantación.
- El protocolo de búsqueda debe ser de fácil implementación y automatización.
- Se debe integrar de forma sencilla.
- Tiene que poder usarse a escala mundial y ser útil, es decir, debe proporcionar los datos que se están buscando (aunque no siempre es posible), y no todos los países tienen la misma información, por ejemplo en Estados Unidos existe un código de empresas que no se maneja en otros países.
- La base de “conocimiento” de servicios / empresas también tiene que ser a escala mundial, para poder buscar información de empresas de cualquier lugar.
- Tiene que poder ser usado por distintas máquinas, con distintas tecnologías, distintos software, distintos sistemas operativos...

Muchos de los problemas mencionados en la lista anterior ya surgieron cuando se trataba de plantear una computación basada en objetos distribuidos a través de Internet. La solución está en convertir el entorno en un medio gestionado por mensajes, y como no podía ser de otra manera, estos mensajes estarán en XML. Pero si se usan mensajes XML, y si se emplean protocolos de red estándares en Internet, ¿Por qué no aprovechar directamente SOAP?

Esta es la solución adoptada. En septiembre de 2000 sale a la luz la primera versión de UDDI (Universal Description Discovery and Integration), combinación de XML, HTTP y SOAP.

La última versión disponible del UDDI es la 3.0, aparecida en Julio de 2002, aunque la más utilizada es la 2.0, que presenta varias ventajas de clasificaciones y morfología frente a la versión 1.0. La versión 3.0 se realizó para poder tener mayor control sobre los registros públicos y privados, permitiendo a las empresas tener su propia “intranet” de servicios web apoyada por un registro UDDI.

UDDI permite publicar información sobre los web services, ayudándose de los documentos WSDL de estos, así se puede hacer llegar al usuario la información de la empresa y de sus servicios de manera sencilla y rápida. Además UDDI permite ser consultado para la creación de clientes de forma automática, por mantener referencias de los documentos WSDL.

UDDI es un compendio que abarca la descripción, publicación y búsqueda de un web service. La idea es clara, se tiene un web service, un WSDL (que puede tener información adicional referente a la empresa, a los servicios ofrecidos por ésta, sus sucursales...) y se tiene que hacer llegar a manos de los

clientes potenciales la descripción de los servicios ofrecidos y la manera de conseguirlos. Lo que se hace, es publicar la información pertinente en un “directorio”, que pueda ser consultado por cualquiera. UDDI será quien permitirá publicar, y consultar servicios basados en web, que estén incluidos en un servidor accesible mediante Internet.

Este caso es como las páginas amarillas, se va al índice, se busca un servicio, y se encuentra varias empresas que se podrían encargar de proporcionárselo.

Así por ejemplo, se pueden encontrar rápidamente varios proveedores de un mismo producto, consultar sus precios y condiciones de envío y hacer el pedido, sin tener que estar varias horas buscando en diferentes web y haciendo cálculos.

En el CDROM que acompaña la guía se ha incluido un paquete de desarrollo de servicios web en Java llamado Java Web Services Developer Pack. En él se puede encontrar un explorador de registros UDDI. El ejecutable correspondiente es `jaxr-browser.sh` en Unix y `jaxr-browser.bat` en Windows.

La especificación UDDI, puede obtenerse mediante de distintos juegos de definiciones XML (concretamente mediante XML Schema), que están disponibles al público. Estos juegos de XML Schema, se pueden clasificar en cuatro grupos distintos:

- Replicación UDDI

Se da información acerca de los procesos de réplica de datos, de las interfaces que ofrecen los distintos nodos para los procesos de duplicación de datos entre los diferentes nodos UDDI. Como programador no interesa esta especificación, ya que solamente es para procesos automáticos de copia.

- Operadores UDDI

Ofrece la especificación que debe ser utilizada por los operadores del registro. Define los procedimientos que se deben seguir para el manejo de datos. Por ejemplo, esta especificación se utiliza para generación de copias de seguridad de los datos, comprobación su integridad (verificación de direcciones, evitar elementos huérfanos durante los procesos de borrado, etc.). Esta especificación es utilizada solamente en el registro UDDI “universal”; aunque se puede usar en registros privados, no es lo normal.

- API para programadores UDDI

Definen las funciones de publicación y búsqueda de datos acerca de negocios, entidades o servicios dentro del registro. Concreta una serie de mensajes SOAP, que son analizados por el registro UDDI y éste es capaz de responder ante ellos, bien rechazándolos, bien registrando la información o bien devolviendo la respuesta a una búsqueda. Las estructuras de datos que se pueden utilizar en este apartado vienen definidas mediante otro XML Schema. Este API está soportado por todos los registros, públicos o privados.

- Estructuras de datos UDDI

Esta especificación define cinco tipos de estructuras de datos y las relaciones que existen entre ellas. Estas estructuras las encontramos en los mensajes SOAP definidos dentro del API para programadores (punto anterior).

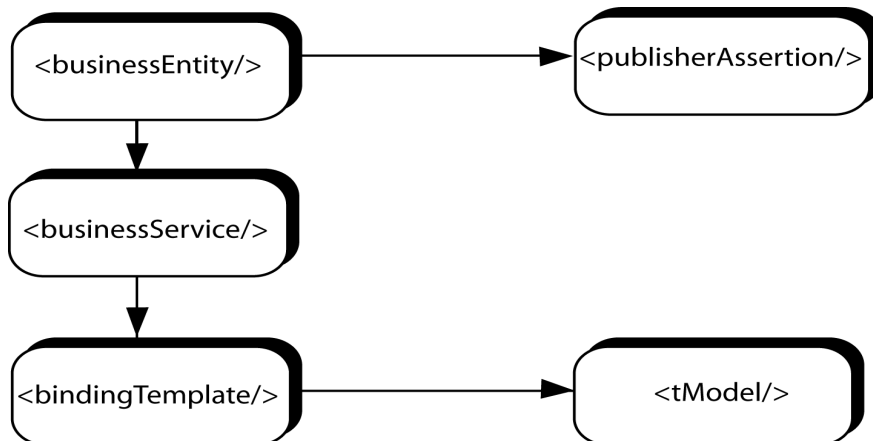


Figura 7: Estructuras de datos definidas en UDDI y sus relaciones.

5.2.1 Localización de UDDI

El registro “universal” UDDI, no está físicamente en una máquina, o en una compañía, sino que está mantenido por un consorcio de varias empresas, tales como IBM, SAP o Microsoft, que continuamente realizan réplicas de los datos entre sus servidores, manteniendo sincronizadas las informaciones de todos ellos, formando así una red de ordenadores con idéntico contenido UDDI.

Esto hace que tras la publicación de un web service sobre uno de los registros la información se transfiera automáticamente hacia el resto de las bases de datos que mantienen la información. La búsqueda se realiza de la misma forma, no importa la dirección de UDDI que se consulte (si es una dirección de UDDI pública y no es un registro de pruebas), ya que al estar los datos replicados, la información recibida será la misma (o debería serlo) consulte el servidor que consulte.

Esta distribución permite hacer todas las operaciones sobre un mismo registro UDDI, es decir, las publicaciones de soporte, de servicios, búsquedas, etc.. se realizan sobre uno de los registros y serán los propios registros quienes se encarguen de gestionar esta información para transmitirla entre ellos. Es lo que se llama entorno de nubes UDDI (UDDI cloudscape) o registro de negocios UDDI (UDDI Business Registry, UBR).

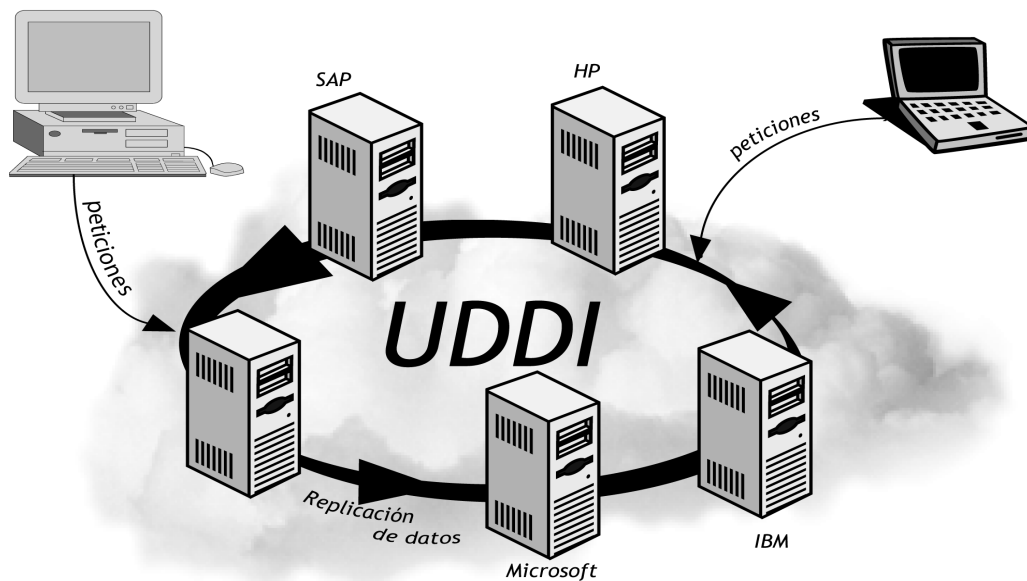


Figura 8: UDDI cloudscape.

UDDI es más que UBR, ya que la especificación UDDI nos permite tener registros privados a nivel de intranet, registros de prueba, o incluso generar nuestra propia red de nodos UDDI. Todos estos datos privados, lógicamente, no son replicados hacia el UBR.

Existen también algunos registros donde los programadores pueden realizar sus pruebas antes de pasar al UBR. Por ejemplo la dirección del UDDI de pruebas de SAP es <http://udditest.sap.com/>. En estos registros se puede incorporar información sin necesidad de permiso alguno, al contrario que en el UBR, donde para realizar esta inserción se debe estar registrado.

5.2.2. Información almacenada en UDDI

UDDI nos permite almacenar distintos tipos de datos, y su evolución natural se dirige hacia un sistema completo de informaciones referentes a las empresas, a sus direcciones, a su estructura interna y a sus servicios.

Cuando se piensa que UDDI dará informaciones sobre web services, parece lógico pensar se que mantenga una gran base de datos conteniendo una serie de documentos WSDL con todos los servicios que estén registrados, pero nada más lejos de la realidad. Lo que se hace es almacenar información primaria de las empresas (ordenadas por categorías) y dentro de esta información, es donde se pueden encontrar los enlaces (referencias) a los distintos documentos WSDL, que darán la especificación de los web services que se ofrecen.

Los registros UDDI mantienen cinco grandes tipos de información:

- BusinessEntity (Entidad de negocio) que define las páginas blancas
- BusinessService (Servicio de negocio) que define las páginas amarillas
- BindingTemplate (Patrón de enlace) que define las páginas verdes
- tModel
- publisherAssertion

Estas informaciones están interrelacionadas de la siguiente manera, cada entidad de negocio puede tener uno o mas servicios de negocio, que a su vez pueden tener uno o más patrones de enlace. Por último cada enlace tiene un Tmodel:

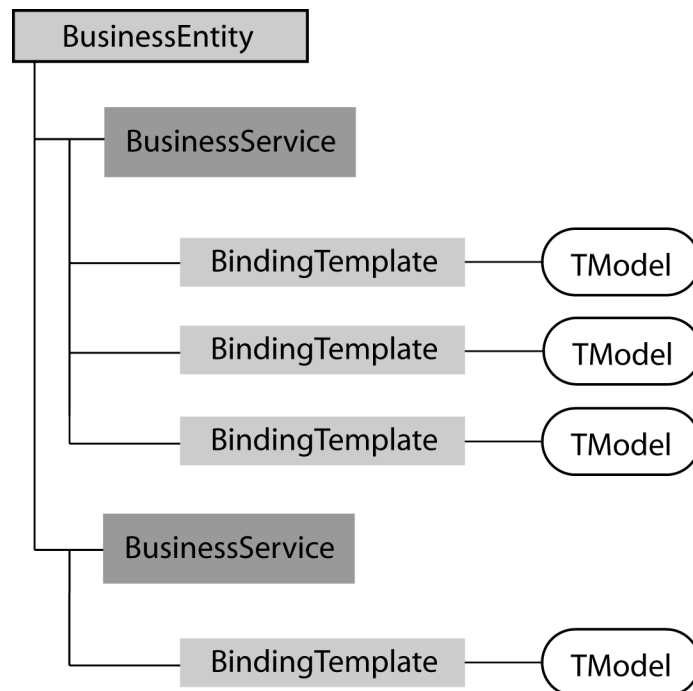


Figura 9: Relación entre las distintas informaciones del registro UDDI.

Un elemento que está presente en varios de estos tipos de datos (businessEntity, businessService y bindingTemplate) y que se tiene que tener siempre en cuenta es el identificador único universal (Universally Unique Identifier UUID), que se utilizará para referenciar cada uno de estos elementos de manera unívoca. Su asignación se realiza durante la primera inserción del dato en el registro, y se garantiza que es única en todo el UBR. Se genera utilizando un algoritmo especificado por el estándar ISO/IEC 11578:1996. En la generación del UUID entran en juego el día y hora de generación, dirección IP, dirección hardware y un número aleatorio, que son almacenados en una cadena hexadecimal de ciento veintiocho bits.

5.2.2.1. BusinessEntity (Páginas blancas)

Contienen información sobre los negocios de la empresa, como pueden ser teléfonos de contacto o direcciones. La mayoría de las veces, esta sección tiene datos redundantes en distintos idiomas para llegar al mayor número de clientes posibles (no olvidemos que los datos serán accesibles a escala mundial).

Además de esta información se pueden encontrar todo tipo de datos que ayudan a la localización de la empresa, como números de faxes, correos electrónicos o direcciones de páginas web.

Otra de las características de esta sección, es que almacena también nombres por los que puede ser conocida la empresa; por ejemplo, si nuestro negocio se llama “Thomas & Helmet unlimited enterprise” es posible que sea conocida simplemente por T&H o por THUE, si los clientes no recuerdan el nombre exacto, es posible perder una búsqueda y con ello un cliente, así estas acepciones permitirán mayor flexibilidad para las exploraciones, también puede incluirse cualquier tipo de identificador como de números de registro social, números *DUNS* (Dun & Bradstreet Numbers) o identificadores fiscales que ayuden a la localización de la empresa.

El XML Schema correspondiente a esta estructura es el siguiente:

```
<element name="businessEntity" type="uddi:businessEntity" />
```

```
<complexType name="businessEntity">
```

```

<sequence>

<element ref="uddi:discoveryURLs" minOccurs="0" />

<element ref="uddi:name" minOccurs="0" maxOccurs="unbounded" />

<element ref="uddi:description" minOccurs="0" maxOccurs="unbounded" />

<element ref="uddi:contacts" minOccurs="0" />

<element ref="uddi:businessServices" minOccurs="0" />

<element ref="uddi:identifierBag" minOccurs="0" />

<element ref="uddi:categoryBag" minOccurs="0" />

</sequence>

<attribute name="businessKey" type="uddi:businessKey" use="required"
/>

<attribute name="authorizedName" type="string" use="optional" />

<attribute name="operator" type="string" use="optional" />

</complexType>

```

En las siguientes descripciones, se habla de funciones como *find_business* o *save_business*. Estas funciones y su utilización concreta se verán más adelante en este mismo capítulo.

La descripción de los elementos más importantes se detalla a continuación:

- **authorizedName**
Atributo es aportado por el controlador que realiza la inserción en el registro. No debe ser suministrado en la información de los mensajes *save_business*. Es una cadena de 64 caracteres.
- **businessServices**
Se trata de un elemento que puede ser incluido tantas veces como se quiera. Es una estructura con las descripciones de los servicios. Si en un periodo de días fijado por el operador no se registran servicios, y la estructura está vacía, se procede al borrado de los datos de la estructura *<businessEntity>*.
- **businessKey (Obligatorio)**
Es un identificador único en todo el registro, y es el UUID de la entrada *<businessKey>* en el registro UDDI. Se almacena en modo hexadecimal con una longitud de 128 bits. Funciona como atributo del elemento y es generado por el nodo operador del registro UDDI. No es obligatorio en la primera inserción (se transmite vacío) de los datos, ya que como es lógico, aún no se tiene esta cadena, pero sí es obligatoria su transmisión cuando se efectúan modificaciones de los datos.
- **discoveryURLs**
Se trata de una estructura opcional que servirá para indicar una lista de URL, que apuntarán a servicios de búsqueda basados en archivo.
- **operator**

Cadena de 48 caracteres que es la certificación del operador de registro UDDI que se encarga de realizar el control de la copia maestra del registro `<businessEntity>`. No debe ser suministrado en la información de los mensajes `save_business`, ya que lo controla el nodo generador del registro.

- `name` (Obligatorio)
Será el nombre con el que se grabará la entidad. Es el elemento que se devuelve en los mensajes de búsqueda del tipo `find_business`. Es una cadena de 128 caracteres comprensible para los humanos.
- `description`
Se trata de un elemento optativo pero muy útil. Permite poner pequeñas descripciones (cadena de 255 caracteres) acerca del negocio. Se pueden poner tantas como se quiera (salvando los 255 caracteres por cada entrada), por lo que se aprovecha para incluir descripciones en varios idiomas pudiendo indicar el mismo mediante el atributo `lang`, por ejemplo `xml:lang="es"`.
- `contacts`
Elemento opcional de tipo estructura que mantiene una lista de contactos.
- `identifierBag`
Estructura de carácter opcional. Dicha estructura tiene la forma de una lista de parejas de valores, esto es, parejas de nombre-valor. Se almacenan los valores de identificadores `<businessEntity>` para poder ser utilizados en búsquedas del tipo `find_business`.
- `categoryBag`
Al igual que el elemento anterior, esta estructura de carácter opcional, está constituida por una lista de parejas nombre-valor. En este caso es utilizada para clasificar al `<businessEntity>` en distintas categorías, como por ejemplo tipo de producto o localización geográfica. Al igual que el `<identifierBag>`, se utiliza para las búsquedas del tipo `find_business`.

5.2.2.2. **BusinessService (Páginas amarillas)**

Estas estructuras contienen la información referente a los servicios que ofrece el negocio, áreas de trabajo, localización geográfica, etc.

Para las categorías de negocio, se usan cuatro clasificaciones (aunque en la versión dos se han añadido clasificaciones nuevas, éstas aún no son muy usadas), que son:

- Productos y servicios (UN/SPSC – ECMA Universal Standard Products and Services Classification, clasificación universal estándar de servicios y productos). Es utilizado a escala mundial. Para más información: <http://www.unspsc.org>. El `tModelName` correspondiente es: `tModelName:unspsc-org:unspsc:3-1`.
- Localización geográfica. (ISO 3166) . Incluye información sobre los códigos geográficos. Se usa también a escala mundial. Para más información sobre este tipo de clasificación: <http://www.din.de/gremien/nas/nabd/iso3166ma>. El `tModelName` correspondiente es: `tModelName:iso-ch:3166:1999`.
- Industria de los Estados Unidos (NAICS - North American Industry Classification System. sistema de clasificación de la industria de Norte América). Para más información acerca de este sistema de clasificación: <http://www.census.gov/epcd/www/naics.html>. El `tModelName` correspondiente es: `tModelName:ntis-gov:naics:1997`.
- Otros. Es una clasificación para la cual no existe especificación de uso ni de formas preestablecidas, ya que cada operador UDDI puede gestionar las suyas propias. También se utilizan para realizar agrupaciones de servicios de semejante propósito, pero que no se adaptan correctamente a ninguna de las clasificaciones anteriores. En este caso, el `tModelName` correspondiente es: `tModelName:uddi-org:general_keywords`.

La implementación de este apartado se realiza mediante parejas de nombre - valor, esta manera de realizarla, permite enlazar rápidamente con la información de las páginas blancas.

El XML Schema correspondiente a esta estructura es el siguiente:

```
<element name="businessService" type="uddi:businessService" />

<complexType name="businessService">
  <sequence>
    <element ref="uddi:name" minOccurs="0" maxOccurs="unbounded" />
    <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded" />
    <element ref="uddi:bindingTemplates" minOccurs="0" />
    <element ref="uddi:categoryBag" minOccurs="0" />
  </sequence>
  <attribute name="serviceKey" type="uddi:serviceKey" use="required" />
  <attribute name="businessKey" type="uddi:businessKey" use="optional" />
</complexType>
```

Un detalle de sus elementos:

- serviceKey (Obligatorio)

Es el UUID del servicio, es decir es la clave única por la que se puede referenciar el elemento *<businessService>*. La clave es generada por el operador la primera vez que se inserta la información correspondiente al servicio, y no debe ser proporcionado en esta operación (se transmite vacío), pero sí debe ser informado cuando se trate de, por ejemplo, modificaciones. Es una cadena hexadecimal de 128 caracteres.

- description

Cero o más elementos con la descripción del servicio. Se suele utilizar para incluir descripciones del servicio en varios idiomas pudiendo indicar el mismo mediante el atributo *lang*. Es una cadena de 255 caracteres (por cada elemento).

- businessKey (Obligatorio en ocasiones)

Este elemento puede ser opcional o no, dependiendo de las circunstancias. Si la estructura *<businessKey>* se transmite junto a la entidad superior que la contiene (*<businessEntity>*), y ésta está perfectamente informada, entonces no es necesario; pero si se transmite como estructura independiente, sin ninguna información acerca del *<businessEntity>* al que pertenece, entonces sí debe ir informada. Es el UUID que crea la relación padre-hijo entre el elemento *<businessService>* y el *<businessEntity>* que lo contiene. Como todo UUID, es una cadena hexadecimal de 128 caracteres.

- bindingTemplate (Obligatorio)

Se trata de una estructura que contiene la información de la descripción técnica del servicio. En el siguiente apartado se detalla en profundidad dicha estructura de datos.

- name (Obligatorio)

Es el nombre del servicio; se trata de una cadena de cuarenta caracteres, especificando el nombre del servicio de forma comprensible para los humanos.

- categoryBag

Se trata de una estructura con forma de lista de parejas de valores del tipo nombre – valor.

Esta estructura es de carácter opcional, está constituida por una lista de parejas nombre-valor. En este caso es utilizada para clasificar al *<businessEntity>* en distintas categorías, como por ejemplo tipo de producto o localización geográfica. Se utiliza para las búsquedas del tipo *find_service*.

5.2.2.3. BindingTemplate (Páginas verdes)

Contiene las especificaciones y contenidos técnicos de los servicios que se ofrecen, así como la información necesaria para programar los clientes.

Mediante las páginas verdes es posible obtener la información necesaria para poder efectuar comercio electrónico con la empresa, además de los datos técnicos que se necesitan para codificar los clientes que se encargarán de manejar las transacciones comerciales. Los servicios pueden a su vez catalogarse en nuevas categorías.

Su XML Schema es:

```
<element name="bindingTemplate" type="uddi:bindingTemplate" />

<complexType name="bindingTemplate">

  <sequence>

    <element ref="uddi:description" minOccurs="0"
maxOccurs="unbounded" />

    <choice>

      <element ref="uddi:accessPoint" />

      <element ref="uddi:hostingRedirector" />

    </choice>

    <element ref="uddi:tModelInstanceDetails" />

  </sequence>

  <attribute name="serviceKey" type="uddi:serviceKey" use="optional"
/>

  <attribute name="bindingKey" type="uddi:bindingKey" use="required"
/>

</complexType>
```

Donde

- serviceKey

Es el identificador único de *<businessService>*, es decir su UUID. Si el elemento *<bindingTemplate>* se proporciona dentro de una entidad superior (*<businessService>*) totalmente informada, este identificador puede omitirse, pero si se suministra como entidad independiente (sin ningún *<businessService>*), debe indicarse esta cadena de identificación, puesto que servirá de enlace con su entidad superior, define la relación padre hijo.

- bindingKey (Obligatorio)

Es la clave única de la estructura, corresponde a su UUID, y como tal, es una cadena hexadecimal de 128 bits. Como en anteriores casos, esta cadena la proporciona el operador UDDI, por lo que en la primera inserción de los datos, este atributo debe ir en blanco. Para operaciones tales como la actualización de datos se debe proporcionar siempre.

- description

Cadena de 255 caracteres, sirve para la descripción técnica del punto de entrada al servicio. Al igual que en ocasiones anteriores, este elemento suele estar en varios idiomas y podemos usar el atributo *lang*, y puede haber cero o más elementos.

- accessPoint (Obligatorio en ocasiones)

Este elemento debe existir siempre que no exista un *<hostingRedirector>*, y además tiene que ser único, debe haber uno y sólo uno. Es un texto de 255 caracteres que servirá para indicar el punto de entrada al servicio. El modo en el que se da este punto de entrada es muy diverso, ya que acepta desde direcciones web hasta teléfonos. Para ayudar en su análisis, este elemento tiene un atributo llamado *URLType*, que sirve para especificar el tipo de enlace que se informa. Como posibles tipos *URLType* tenemos:

- http: el punto de entrada es una dirección web. Ej. <http://algun.lugar.com/miServicio>
- https: es semejante al anterior, solo que en este caso la conexión se realiza usando SSL (Secure Socket Layer), es decir cifrada. Ej <https://algun.lugar.com/miServicioSeguro>
- ftp: El punto de acceso se realiza a partir de una dirección FTP (File Transfer Protocol). Ej. <ftp://algun.lugar.com/miServicioFtp>
- mailto: el punto de acceso es un correo electrónico. Ej. <mailto:alguien@algun.lugar.com>
- phone: Indica que el punto de entrada es un número de teléfono que será respondido por una persona, o en su caso por un sistema automático de tonos o palabras. Ej. 34-999115599
- fax: El punto de entrada al servicio, corresponde a un número que está atendido por un fax. Ej. 34-999559911
- other: Sirve para indicar cualquier otro tipo de direcciones. Si se da este *URLType*, al menos un *<tModel>* de la colección *<tModelInstanceInfo>* debe indicar el formato o tipo de transporte particular.

- hostingRedirector (Obligatorio en ocasiones)

Es obligatorio si no se ha especificado un *<accessPoint>*. Sirve para indicar una redirección del *<bindingTemplate>* actual hacia otro distinto, es decir, actuará como puntero hacia otro *<bindingTemplate>*. Los datos que debemos usar son los del destino de la dirección y no los de la estructura que contiene la redirección. Este elemento puede aparecer por distintas causas, por ejemplo que el servicio se haya cambiado de dirección o que se quiera mantener distintos descriptores de servicio para una sola instancia del servicio. *<hostingRedirector>* no tiene elementos hijos, su único atributo se llama *<bindingKey>* y el valor que contiene es el UUID necesario para obtener el *<bindingDetail>* con los datos necesarios para usar el servicio.

- tModelInstanceDetails

Es una estructura que mantiene una lista de uno o más *<tModelInstanceInfo>*. Los *<tModelInstanceInfo>* mantienen un enlace con los elementos *<tModel>*. Todos ellos como conjunto forman una especie de firma digital que puede usarse en búsquedas. Cuando alguien registra un *<bindingTemplate>*, éste contiene una o más referencias a un *<tModel>*, y todas ellas forman un conjunto que puede ser utilizado total o parcialmente en búsquedas, ya que se puede buscar un *<bindingTemplate>* que tenga referencias a un determinado grupo de *<tModel>* o a un *<tModel>* concreto, por lo que las búsquedas se agilizan.

La estructura *<tModelInstanceInfo>* es la siguiente:

- tModelKey (Obligatorio)

Es un atributo que representa el identificador único que referencia al `<tModel>` que estamos describiendo. Es un UUID (cadena hexadecimal de 128 caracteres).
- description

Elemento que sirve de descripción de la función del `<tModel>` dentro del servicio, es una cadena de 255 caracteres y normalmente se usan varias entradas para informar las descripciones en distintos idiomas.
- instanceDetails

Elemento formado por una estructura de datos que se usa para dar más información acerca del uso de la instancia del `<tModel>`, por ejemplo si se necesitan ciertos parámetros o algún protocolo de acuerdo en la comunicación (*handshakes*).

5.2.2.4 tModel

Los elementos `<tModel>`, son unas estructuras de datos que contienen información técnica acerca del servicio al que representan. A diferencia de las estructuras anteriores, los `<tModel>` no son únicos de un elemento `<bindingTemplate>` o de un `<businessEntity>`, sino que como se ha visto en uno de los puntos anteriores, los `<tModel>` se usan mediante referencias. Un elemento `<tModel>` almacena datos sobre los datos, es lo que se denomina *metadata*.

Principalmente los `<tModel>` tienen tres usos:

- representación técnica de las especificaciones de los servicios (technical fingerprint, huella digital técnica)
- organizador
- modulador de búsquedas

La actuación como representación técnica se podría decir que es su principal misión. Es posible representar por ejemplo los protocolos y formatos utilizados en transmisiones por ondas, indicando todos los parámetros necesarios para que la transmisión se realice correctamente. Si se quiere realizar una nueva descripción sobre la forma de realizar un tipo de comunicaciones, se puede incluir un nuevo `<tModel>` en el registro UDDI. Los documentos que contienen la información sobre cómo se debe realizar las tareas que implica ese `<tModel>`, vienen referenciadas por unos enlaces que se almacenan en la estructura `<overviewDoc>` dentro del propio `<tModel>`, es decir, no están físicamente en el registro.

Si se tiene un servicio que es compatible con las directrices marcadas por un `<tModel>`, se puede anunciar su compatibilidad simplemente añadiendo una nueva entrada al elemento `<tModelInstanceDetails>` dentro de la estructura `<bindingTemplate>`. Esto ayuda a la búsqueda de web services que sean compatibles con una o varias especificaciones técnicas.

Como organizador, podemos encontrar sus referencias en el `<identifierBag>`, en el `<categoryBag>`, en `<address>` y en `<publiserAssertion>` (esta estructura se verá en el apartado 5.2.2.5).

Como modulador de búsquedas (*find qualifier*) actuará cuando su presencia modifique la respuesta de la búsqueda. Con ellos es posible variar el comportamiento del API `find_XXX`. Por ejemplo, para ordenar los nombres de una búsqueda por orden alfabético se debe especificar el modulador `sortByNameAsc`.

Su XML Schema es el siguiente:

```
<element name="tModel" type="uddi:tModel" />

<complexType name="tModel">

<sequence>

<element ref="uddi:name" />

<element ref="uddi:description" minOccurs="0" maxOccurs="unbounded" />
```

```

<element ref="uddi:overviewDoc" minOccurs="0" />
<element ref="uddi:identifierBag" minOccurs="0" />
<element ref="uddi:categoryBag" minOccurs="0" />
</sequence>
<attribute name="tModelKey" type="uddi:tModelKey" use="required" />
<attribute name="operator" type="string" use="optional" />
<attribute name="authorizedName" type="string" use="optional" />
</complexType>

```

Donde:

- **tModelKey (Obligatorio)**
Es un atributo que servirá como identificador único del *<tModel>*, en este caso el UUID es una cadena de 255 caracteres. Cuando se introduce por primera vez los datos del *<tModel>* en el registro, este atributo se debe enviar en blanco, pero para las sucesivas modificaciones es obligatorio.
- **authorizedName**
Es un atributo que es generado por el operador, por lo que no debe ser suministrado. Se trata de una cadena de 64 caracteres.
- **operator**
Es el nombre certificado del nodo operador UDDI que gestiona la copia maestra del *<tModel>*. El valor de este atributo no se debe suministrar, puesto que es calculado en el momento que se inserta el elemento en el registro. Es una cadena de 48 caracteres.
- **name(Obligatorio)**
Es el nombre del *<tModel>*. Este nombre es el que se buscará en los mensajes *find_tModel*. Se trata de una cadena de 128 caracteres y tiene que ser entendible por un humano.
- **description**
Cero o más descripciones del *<tModel>*. Se suele dar una entrada por idioma pudiendo indicar el mismo mediante el atributo *lang*, por ejemplo *xml:lang="de"*, y cada una de las entradas puede tener hasta 255 caracteres.
- **overviewDoc**
Es una estructura en la que se detallan los lugares en los que se puede encontrar los documentos relativos al *<tModel>*.
- **identifierBag**
Se trata de estructura con forma de lista de parejas del tipo nombre-valor. Es semejante a la de la estructura *<businessEntity>*.
- **categoryBag**
Es una estructura con forma de lista de parejas del tipo nombre-valor. Se utiliza para clasificaciones. Es semejante a la de la estructura *<businessEntity>*.
Para comprender un poco mejor esta estructura se propone el siguiente ejemplo de un *<tModel>*:

```
<?xml version="1.0"?>
```

```

<tModel tModelKey="...">
  <name>http://www.acme.com/facturas</name>
  <description xml:lang="es">
    Lista de facturas.
  </description>
  <description xml:lang="en">
    Invoice list.
  </description>
  <overviewDoc>
    <description xml:lang="es">
      Documento WSDL relativo a la interfaz del servicio
    </description>
    <description xml:lang="en">
      WSDL Service Interface Document
    </description>
    <overviewURL>
      http://www.acme.com/services/INVO-interface.wsdl#EEN
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="UUID:C22CF21A-9672-5523-9A47-45FA5624CAB0"
      keyName="uddi-org:types" keyValue="wsdlSpec"/>
    <keyedReference tModelKey="UUID:BD837A36-CAB0-35AA-3226-AAF362A6B377"
      keyName="Relación de facturas de un proveedor"
      keyValue="426768942"/>
  </categoryBag>
</tModel>

```

5.2.2.5 PublisherAssertion

Mantiene las relaciones entre distintos elementos *<businessEntity>* (entidades de negocio). Mediante este elemento se pueden ver las relaciones entre dos negocios, o dos ramas de un mismo negocio. Para que el público pueda ver estas relaciones, ambas empresas han tenido que realizar previamente un *<publisherAssertion>* de forma separada y en cada una de sus estructuras *<businessEntity>*, apuntándose mutuamente (en la empresa A será de empresa A hacia B y en la B la relación será de B hacia A), esto evita confusiones y fingir relaciones que realmente no existen, es decir no es posible decir que se tienen negocios con la empresa Bertelsmann, si ésta no lo ratifica mediante otra estructura *<publisherAssertion>*.

Su esquema se detalla a continuación:

```
<element name="publisherAssertion" type="uddi:publisherAssertion" />

<complexType name="publisherAssertion">

  <sequence>

    <element ref="uddi:fromKey" />

    <element ref="uddi:toKey" />

    <element ref="uddi:keyedReference" />

  </sequence>

</complexType>
```

Donde:

- fromKey (Obligatorio)
Es el *<businessKey>* (UUID) perteneciente a la empresa que actuará como emisor de la relación que se va a definir.
- toKey (Obligatorio)
Es el *<businessKey>* (UUID) perteneciente a la empresa que actuará como receptor de la relación que se va a definir. Esta empresa deberá realizar otro *<publisherAssertion>* para que la relación sea efectiva.
- keyedReference (Obligatorio)
Es una estructura en la que se definen las relaciones entre las empresas implicadas en la estructura *<publisherAssertion>*. Esta estructura da su información a través de tres atributos y no tiene ningún elemento hijo. Los atributos son *tModelKey*, que referencia a un *tModel* (lo más normal es usar el *tModel* uddi:uddi.org:relationships), *keyName* que será el nombre de la relación y *keyValue* (atributo obligatorio) que definirá el tipo de relación en sí.

El XML Schema de *<keyedReference>*:

```
<element name="keyedReference" type="uddi:keyedReference" />

<complexType name="keyedReference">

  <attribute name="tModelKey" type="uddi:tModelKey" use="optional" />

  <attribute name="keyName" type="string" use="optional" />

  <attribute name="keyValue" type="string" use="required" />

</complexType>
```

Como estas relaciones son un poco complicadas de ver, valga un ejemplo de un mensaje de adición de un nuevo *<publisherAssertion>* entre la compañía A y la compañía B. En este caso se define una relación de padre hijo entre dos empresas, la empresa A cuyo *<businessKey>* (UUID) es A6A7FCB1... y la empresa B con el *<businessKey>* B6B74451... en la que el padre es la empresa A.

```
<add_publisherAssertions xmlns="urn:uddi-org:api_v3" >

  <authInfo>slkadfjsadofpe</authInfo>

  <publisherAssertion>

    <fromKey>A6A7FCB1-9D88-11D6-91B6-0003479A7335</fromKey>

    <toKey>B6B74451-9D88-12F6-9ff4-0003129B7524</toKey>

    <keyedReference

      tModelKey="uddi:uddi.org:relationships"

      keyName="Nuevas relaciones"

      keyValue="parent-child" />

  </publisherAssertion>

</add_publisherAssertions>
```

La relación entre dos empresas no se hará pública mientras no existan dos *<publisherAssertion>*, uno de relación de la empresa A hacia la B y otro de la empresa B hacia la A. Además las relaciones deben corresponderse. En el momento en el que falte uno de ellos, la relación desaparecerá de la información pública.

Entre los elementos definidos en este punto y los elementos del documento WSDL , se podría realizar la relación representada en la figura 4.

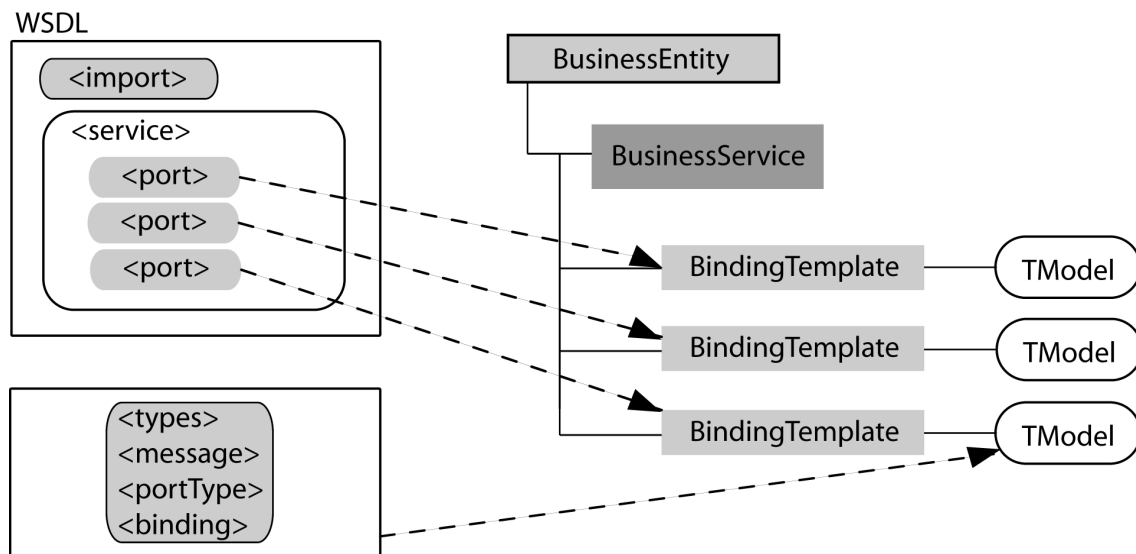


Figura 10: Relación de elementos UDDI u elementos WSDL.

5.2.3. Mensajes de control del registro

El estándar UDDI define una API de mensajes, mediante la cual se permite la comunicación con el registro. Esta API define ciertas llamadas divididas en varias categorías, mediante las cuales se obtendrá un control absoluto del registro. Algunas de las llamadas son accesibles a todo el público, otras será necesario proporcionar una clave de acceso, y otras serán totalmente inaccesibles (como programador y/o usuario).

Tanto los datos proporcionados por el programador y/o usuario como por el registro en sus respuestas están constituidos mediante XML, formando parte de la sección `<Body>` del `<Envelope>` del mensaje SOAP.

Los distintos tipos de mensajes se pueden clasificar según su acceso, según su utilidad, según su alcance, etc, en este caso la clasificación se realizará mediante la finalidad del mensaje, dividiéndolos en cinco grupos:

- Mensajes de búsqueda general
- Mensajes de búsqueda específica
- Mensajes de publicación
- Mensajes de borrado
- Otros tipos

El registro del negocio se realiza mediante un documento XML que contiene las partes que interesa introducir en la base de datos. Una vez introducida la información en uno de los registros ésta se replica hacia el resto de los nodos, consiguiendo así una información uniforme en todas las bases de datos que den servicio UDDI.

Tanto la petición de información como la respuesta con la misma se producen usando SOAP

5.2.3.1 Mensajes de búsqueda general.

Mediante este conjunto de mensajes se puede obtener información general del registro, como una lista de elementos `tModel` que contengan un determinado servicio. En todos ellos, el elemento del `Body` comienza por la palabra `find`, continuado por la palabra de la cual deseamos obtener información. El mensaje de respuesta contiene distintos elementos en cada caso, dependiendo del mensaje enviado al registro. Las llamadas disponibles son:

- `find_business`

El documento respuesta devuelve un `<Body>` conteniendo un y sólo un elemento `<businessList>`. Mediante una búsqueda es posible obtener cero o más elementos `<businessInfo>` que estarán contenidos dentro del elemento `<businessList>`. La consulta se puede realizar especificando una categoría de negocio, un identificador de negocio, un `<tModel>` (con los datos que se conozcan) o una expresión regular de consulta.
- `find_service`

En el mensaje de respuesta, el `<Body>` contiene un y sólo un elemento `<serviceList>`, conteniendo cero o más elementos `<businessService>` que concuerden con los criterios de búsqueda. Para realizar la búsqueda se le debe proporcionar el UUID del `<businessEntity>` que se quiera inspeccionar, además se debe añadir para la búsqueda el nombre del servicio, la categoría del mismo o un `<tModel>`.
- `find_binding`

Mediante el UUID de un `<businessService>` se podrá obtener un y sólo un elemento `<bindingDetail>` conteniendo cero o más elementos `<bindingTemplate>`.
- `find_tModel`

La búsqueda se realiza partiendo de una categoría, un nombre o un identificador UUID. La respuesta contiene un y sólo un elemento llamado `<tModelList>` que contiene cero o más elementos que cumplen los criterios de búsqueda.

- `find_relatedBusinesses`

Dado el UUID de un elemento `<businessEntity>`, esta petición devuelve un y sólo un elemento llamado `<relatedBusinessesList>` conteniendo una lista de UUID correspondientes a los UUID de las `<businessEntity>` que tienen relación con la dada en la búsqueda.

5.2.3.2 Mensajes de búsqueda específica.

Mediante las llamadas de la sección anterior se podrán realizar búsquedas generales, pero muchas veces estos datos no son suficientes. Se utilizan entonces llamadas para la obtención de información específica. Las llamadas de esta sección comienzan por `get_` y terminan por `Detail`.

- `get_businessDetail`

Dados uno o más UUID correspondientes a diferentes `<businessEntity>`, devuelve un y sólo un elemento `<businessDetail>` conteniendo los documentos `<businessEntity>` correspondientes a cada uno de los UUID proporcionados que coincidieran en el registro.

- `get_serviceDetail`

Dados uno o más UUID referentes a estructuras `<businessService>`, esta función devuelve un y sólo un elemento `<serviceDetail>` que contiene a su vez, un elemento `<businessService>` por cada uno de los UUID proporcionados como parámetros, que coincidieran en el registro.

- `get_bindingDetail`

Dados uno o más UUID correspondientes a distintos elementos `<bindingTemplate>`, se obtiene un elemento `<bindingDetail>` conteniendo un conjunto de elementos `<bindingTemplate>`, uno por cada uno de los UUID que coincidieran en el registro. La especificación recomienda el uso de caché local para los documentos `<bindingTemplate>`, de manera que cuando se repitan las llamadas no haga falta consultar al registro UDDI, sino leer la caché. En caso de fallo en la caché (que no exista el dato) o que falle la llamada usando los datos contenidos por ella, se realiza una nueva petición `get_bindingDetail` para conseguir nuevos datos.

- `get_tModelDetail`

Dados uno o más UUID correspondientes a distintos documentos `<tModel>`, se obtiene un y solamente un elemento `<tModelDetail>` conteniendo una lista de `<tModel>`, uno por cada uno de los UUID proporcionados que coincidieran en el registro.

5.2.3.3 Mensajes de publicación.

Cada uno de los documentos referentes a las distintas estructuras UDDI se van a guardar utilizando alguno de los mensajes expuestos en este punto. Mediante ellos se obtiene un control sobre las publicaciones en el registro.

Las llamadas de almacenamiento requieren un elemento especial (*authentication token*), que servirá de autenticación de los datos transmitidos. Dicho elemento se obtiene mediante el uso del mensaje `get_authToken`.

Estas peticiones no pueden ser realizadas por cualquier usuario, sino que deben realizarse por usuarios autorizados. Para la validación de usuario existe una petición que se verá en detalle más adelante (el mensaje `get_authToken`). Todas las llamadas de almacenamiento de datos comienzan por `save_` y a continuación se muestra la lista.

- **save_business**
 Dados un elemento de autenticación y uno o más elementos del tipo *<businessEntity>*, permite la grabación (o la inserción si no existía previamente) de los datos correspondientes. Hay que tener en cuenta que la modificación de los datos de un *<businessEntity>* puede acarrear la modificación de otras estructuras; exactamente afecta a los documentos *<publisherAssertion>*, a los *<bindingTemplate>* y a los *<businessService>*. Tras la grabación de los datos, el registro devuelve un mensaje con el elemento *<businessDetail>*, especificando los resultados finales del registro UDDI.
- **save_service**
 Dados un elemento de autenticación y uno o más elementos *<businessService>*, permite la grabación o modificación (si existía anteriormente) de los datos correspondientes, mediante los datos pasados como parámetros. Además se debe tener en cuenta que la modificación de estos datos puede repercutir en documentos del tipo *<bindingTemplate>*. Este mensaje, tiene como efecto, la devolución de un mensaje con los datos correspondientes a la información final del registro.
- **save_binding**
 Dados un elemento de autenticación y uno o más elementos *<bindingDetail>*, permite la grabación de los datos proporcionados, bien insertándolos o actualizando los existentes. Este mensaje se puede usar para actualizar las relaciones con los documentos *<bindingTemplate>*. Devuelve un documento *<bindingDetail>* con los resultados finales de la operación.
- **save_tModel**
 Mediante un elemento de autenticación y uno o más elementos *<tModel>*, permite insertar los datos obtenidos como parámetros o actualizarlos en caso de que existieran previamente. Tras el proceso se devuelve un documento del tipo *<tModelDetail>* con los resultados finales obtenidos en el registro.
- **add_publisherAssertions**
 Dado un elemento de autenticación y un documento *<publisherAssertion>*, se añade la información del documento pasado por parámetro a la colección de elementos de relación de la empresa. El elemento *<publisherAssertion>* genera la relación entre dos empresas, por lo que hasta que no exista un *<publisherAssertion>* por parte de las dos partes involucradas correspondiéndose mutuamente, no se hará pública dicha relación. Devuelve un documento del tipo *<dispositionReport>*.
- **set_publisherAssertions**
 Dado un elemento de autenticación, y uno o más elementos *<publisherAssertion>*, se pueden actualizar de una sola vez todos los elementos *<publisherAssertion>* contenidos en la lista de entrada, y eliminando aquellos que no estén contenidos. Este mensaje devuelve un documento conteniendo los *<publisherAssertion>* finales resultantes de la operación.

5.2.3.4 Mensajes de borrado.

Con este conjunto de mensajes se pueden borrar los documentos previamente salvados mediante los mensajes de publicación anteriormente vistos. Para poder usar cualquiera de estos mensajes, se debe proporcionar al servidor, un elemento de autenticación que en secciones posteriores se verá como conseguirlo.

- **delete_business**
 Dado un elemento de autenticación y uno o más UUID de elementos *<businessEntity>*, mediante este mensaje se procede a la eliminación de todos los elementos *<businessEntity>* del registro que coincidan con los dados. La eliminación de estos elementos acarrea también

la eliminación de todos los elementos `<businessService>`, `<bindingTemplate>` y `<publisherAssertions>` que estén relacionados con los elementos pasados por parámetro. Este mensaje devuelve un documento del tipo `<dispositionReport>`.

- `delete_service`

Dado un elemento de autenticación y uno o más UUID de elementos `<businessService>` o más, causan el borrado del registro de los elementos que coincidan con los elementos de la lista dada como parámetro. Como respuesta del registro se obtiene un mensaje del tipo `<dispositionReport>`.

- `delete_binding`

Dado un elemento de autenticación y un elemento UUID o más del tipo `<bindingTemplate>`, se consigue el borrado de los elementos `<bindingTemplate>` del registro que coincidan con los proporcionados como parámetros.

- `delete_tModel`

Dado un elemento de autenticación y uno o más UUID de elementos `<tModel>`, este mensaje “borra” los elementos que coincidan con los proporcionados en la lista dada como parámetro. Los elementos son transformados en invisibles, esto es, no son físicamente borrados, sino marcados como no accesibles. Mientras se encuentren en este estado, podrán ser accedidos mediante el mensaje `<get_tModelDetail>` y `<get_registeredInfo>` pero no mediante `<find_tModel>`, esto permite seguir trabajando a las empresas que conozcan su existencia con ellos, pero no permite su búsqueda.

- `delete_publisherAssertions`

Dado un elemento de autenticación válido y uno o más elementos UUID de elementos `<publisherAssertion>`, borrará los elementos `<publisherAssertion>` de la colección de relaciones de la empresa que está publicando. Los elementos correspondientes de las compañías que tienen relación con los `<publisherAssertion>` borrados, quedan inalterados en sus colecciones, pero la relación deja de existir por faltar una de las partes.

5.2.3.5 Otros mensajes.

- `get_authToken`

Mediante el paso como parámetros de un nombre de usuario y una clave, se obtiene un elemento de autenticación, entregado por parte del nodo operador UDDI, y servirá para el uso del API de publicación y borrado.

- `discard_authToken`

Sirve para salir del sistema de publicación (*logout*), para realizar la desconexión del sistema de publicación (y borrado). Dado un elemento de autenticación válido, el sistema procede a su invalidación. De esta forma, si se intentara publicar algún nuevo elemento, daría error por no estar correctamente validado, por lo que para realizar esta acción debería volver a usar el mensaje `get_authToken`.

- `get_registeredInfo`

Dado un elemento de autenticación, se obtiene una lista de `<businessEntity>` y `<tModel>` que pueden ser manipulados por esas credenciales de autenticación.

5.2.4 Identificadores

En apartados anteriores, se vio que en UDDI se pueden marcar los datos mediante identificadores que ayuden a su localización. Es aconsejable el uso de identificadores para la empresa, por ejemplo dentro del elemento `<businessEntity>` o dentro del `<tModel>`. No es obligatorio su uso, pero las empresas pueden

utilizar números de seguridad social o números GLN (Global Location Number, número de localización global) u otros medios de localización.

Para poder utilizar estos identificadores, deben ser soportados por el registro. Se pueden obtener los identificadores soportados por un registro UDDI, mediante la consulta *find_tModel* siguiente:

```
<find_tModel>

  <categoryBag>

    <keyedReference

      tModelKey="uddi:uddi.org:categorization:types"

      keyValue="identifier" />

    </categoryBag>

</find_tModel>
```

Dos de los identificadores más utilizados son los números DUNS y GLN. Información sobre identificadores DUNS se puede encontrar en <http://www.dnb.com/> y sobre GLN en <http://www.ean-int.org/locations.html>

Un ejemplo del uso de un número DUNS sería en un elemento *<identifierBag>* dentro de *<businessEntity>*

```
<identifierBag>

  <keyedReference

    tModelKey="uddi:ubr.uddi.org:identifier:dnb.com:D-U-N-S"
    keyName="D-U-N-S:My Company"
    keyValue="12-345-6789" />

</identifierBag>
```

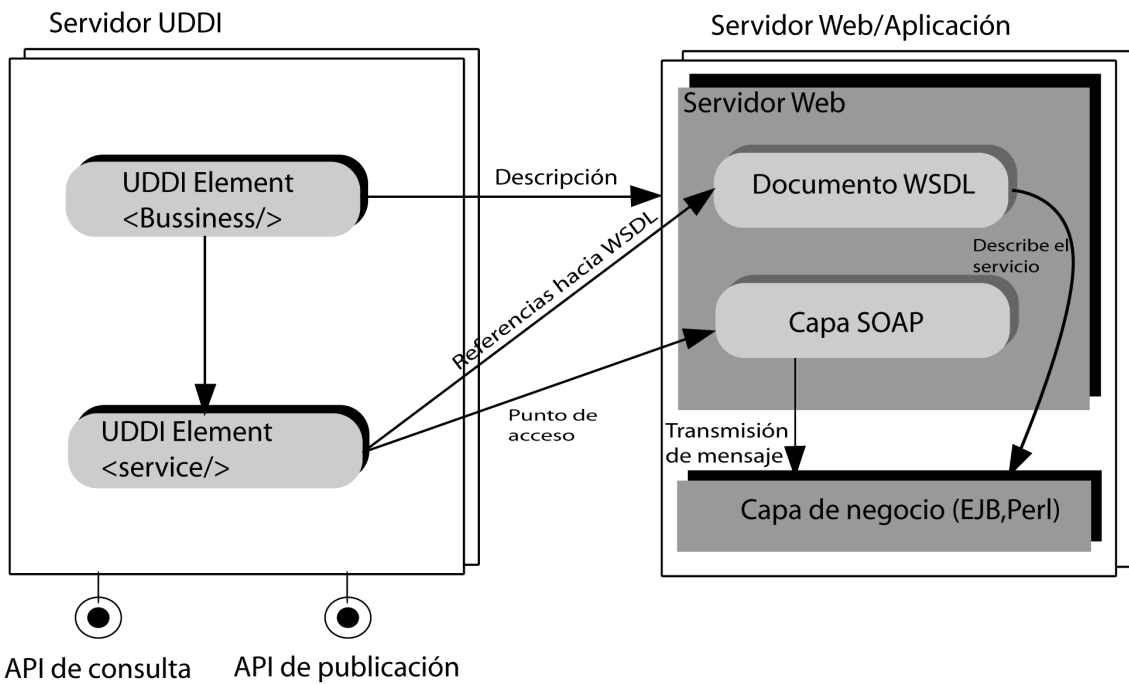


Figura 11: Representación gráfica de dependencias UDDI.

5.2.5. Exploradores UDDI

Para las consultas de los registros UDDI, es muy normal el uso de programas denominados exploradores UDDI, que se encargarán de crear los mensajes vistos en esta sección y presentar las respuestas generadas por registro. En el paquete de desarrollo de servicios web de Java (JWS DP) incluido en el CDROM que acompaña a la guía, se incluye un explorador de éste tipo.

Otra forma de consultar los registros es mediante el uso de páginas web preparadas para tal uso, como por ejemplo: <http://uddi.microsoft.com/default.aspx> o <http://www.soapclient.com/uddisearch.html>.

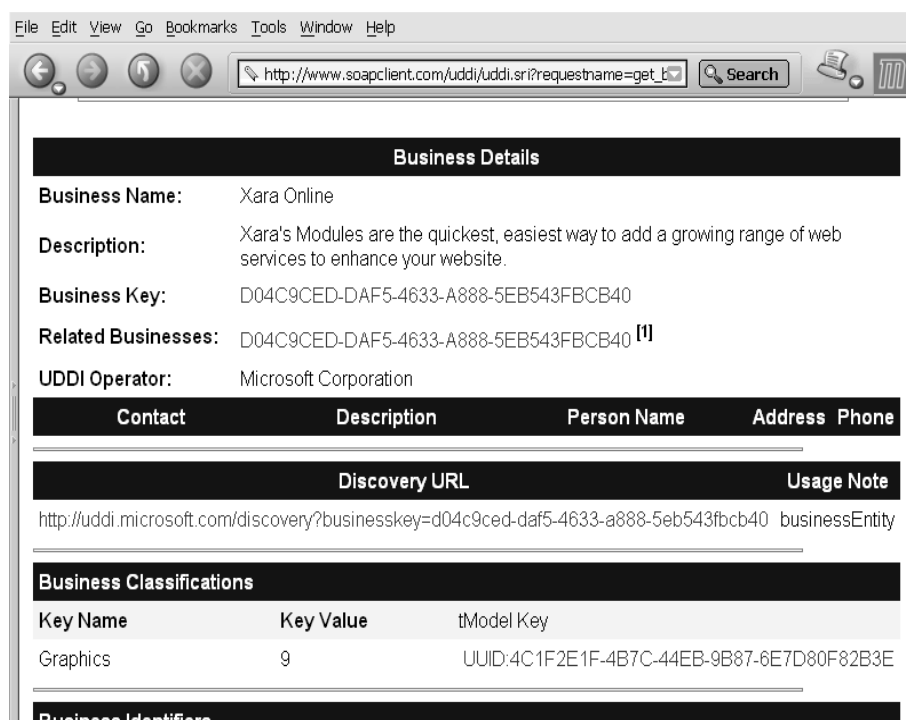


Figura 12: Consulta en un registro UDDI mediante web.

5.3 WSIL

El lenguaje WSIL (Web Service Inspection Language, lenguaje de inspección de servicios web), es otra manera de localizar web services. Como se vio, el UDDI está centralizado, y pese a que se pueden usar UDDI privados, puede que el lector encuentre en este método algunas carencias o dificultades en su uso (como tener que instalar una base de datos). WSIL no es un competidor de UDDI, no está para remplazarlo, sino para completarlo, ya que en algunas ocasiones será más útil el uso de WSIL y en otras ocasiones lo natural será el uso de UDDI, tanto es así que se pueden definir elementos en un documento WSIL que hagan referencia a una entrada de un registro UDDI, o a un documento HTML y viceversa.

WSIL funciona de forma descentralizada, no existe un registro universal en el que buscar, sino que cada nodo puede tener su(s) propio(s) documento(s) WSIL con las especificaciones necesarias para la búsqueda de web services.

Además WSIL se apoya de manera importante en el fichero de descripción WSDL, dejando de lado la información referente a la empresa, mientras que en UDDI, la empresa es la entidad sobre la que se gira toda la información (el elemento raíz es `<businessEntity>`). Esto permite realizar un documento de especificación que será algo semejante a una tarjeta de presentación con los servicios disponibles por parte de dicho documento.

5.3.1 Estructura de WSIL

Cuando se diseñó WSIL, se pensó en un sistema de búsqueda y hallazgo de web services que fuera simple y extensible. En esencia, un documento WSIL es una lista de punteros hacia documentos WSDL que son los encargados de describir el servicio, y al usar XML como soporte base, se puede hacer que sea extensible mediante el uso de *namespaces* (ver capítulo dos), por lo que las metas del diseño están cubiertas. Este esquema de diseño permite una rápida adaptación de los contenidos del fichero, tanto en su creación como en su mantenimiento.

Al final del capítulo se incluye el XML Schema del documento WSIL, pero un detalle de lo que será dicho documento es el siguiente:

```
<wsil:inspection>
```

```

<wsil:abstract xml:lang=""? ... /> *

<wsil:service> *

  <wsil:abstract xml:lang=""? ... /> *

  <wsil:name xml:lang=""? ... /> *

  <wsil:description referencedNamespace="uri" location="uri"?> *

    <wsil:abstract xml:lang=""? ... /> *

    <!-- elementos de extension --> ?

  </wsil:description>

</wsil:service>

<wsil:link referencedNamespace="uri" location="uri"?/> *

  <wsil:abstract xml:lang=""? ... /> *

  <!-- elementos de extension --> ?

</wsil:link>

</wsil:inspection>

```

Se puede ver la definición de las seis etiquetas que se utilizan en este tipo de documentos:

1. inspection

Es la etiqueta que sirve de raíz en un documento WSIL. Representa el elemento que contendrá a todos los demás.
2. abstract

Es un pequeño texto dirigido a los humanos que usen el documento, indicando aclaraciones, modos de uso o cualquier cosa que considere necesaria el diseñador del documento. Puede haber cero o más elementos de este tipo, y de igual forma que en la especificación UDDI, los textos dirigidos a los humanos pueden realizarse en varios idiomas, pudiendo indicar el mismo mediante el atributo *lang*, por ejemplo *xml:lang="es"*. Dependiendo de la localización de la etiqueta, el texto será referente a un elemento o a otro, ya que como se puede observar en el esquema, es posible tener este tipo de marcas en varios lugares.
3. service

Es el elemento donde se define realmente la entrada al localizador del servicio.
4. name

Se refiere al nombre que se dará al servicio, y al igual que en la etiqueta *<abstract>*, es posible indicar el nombre en varios idiomas, que pueden ser identificados mediante su atributo *lang*. Este elemento puede aparecer cero o más veces, y no se asegura que sea único.
5. description

Es junto con el elemento *<link>* la parte más importante del documento WSIL, ya que indica el punto de entrada a la descripción del servicio. Como la entrada a la descripción se puede dar de varias formas, es necesario el uso del atributo *referencedNamespace*, en el que se indicará el *namespace* del documento referenciado. Para el caso más normal (WSDL), el valor de este atributo es <http://schemas.xmlsoap.org/wsdl/>. Otro atributo de este elemento es el *localization*; es un atributo optativo que permite dar un enlace hacia la descripción. El enlace debe ser un URL válido, y el documento debe ser accesible

mediante el mecanismo de acceso primario del URL especificado, esto es, si es un URL de un ftp, se debe poder acceder mediante un GET, o si es un URL de tipo HTTP, se podrá acceder mediante un HTTP GET. Si fueran necesarios pasos de parámetros, tener en cuenta alguna otra consideración o simplemente dar unas pequeñas aclaraciones sobre la información a la que se va a acceder, se usarán elementos de extensión, donde se podrá advertir los detalles pertinentes.

6. link

Este elemento nos permite enlazar el documento WSIL con otros documentos del mismo tipo o cualquier otro método de localización de web services como por ejemplo un registro UDDI. Mediante el atributo *referencedNamespace* se define el *namespace* de la fuente de datos agregada, si es, por ejemplo otro fichero WSIL, el valor de este atributo será `http://schemas.xmlsoap.org/ws/2001/10/inspection/`. El atributo optativo *location*, al igual que en el apartado anterior, permite definir un mecanismo por el cual se puede obtener la fuente de los datos que se quieren enlazar mediante el mecanismo primario de su URL. Si no existe este atributo o no se puede acceder simplemente mediante el mecanismo primario, se debe añadir un elemento de extensión en el que se indica la información necesaria para la obtención de los datos que han sido enlazados.

Un ejemplo de enlace con otro documento WSIL, que será accedido mediante HTTP GET:

```
<?xml version="1.0"?>

<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">

  <link referencedNamespace=

    "http://schemas.xmlsoap.org/ws/2001/10/inspection/"

    location="http://acme.com/compras/inspection.wsil"/>

</inspection>
```

5.3.2 Publicación del documento

WSIL es un método descentralizado de localización de servicios, por ello se plantea el problema de la búsqueda del documento WSIL. Antes, con UDDI, se conectaba a un registro y de allí se obtenía la información necesaria, pero ahora no existe tal registro. La especificación WS-Inspection ofrece dos posibilidades, describiendo su búsqueda y su localización. Estas dos soluciones son mediante:

- nombre fijo
- documento enlazado

5.3.2.1 Nombre fijo

Al realizar la búsqueda del fichero con el documento WSIL, es posible encontrarse con dos problemas, uno es el nombre del fichero y otro la localización. El documento tiene que ser fácilmente accesible (mayor posibilidad de captar clientes), por lo que se debe buscar la manera en la que el usuario tenga mayores posibilidades de obtener dicho documento. Por ejemplo, si todos los documentos se llaman igual, sólo quedaría por saber la dirección. Así es, los documentos de publicación WSIL, deben llamarse *inspection.wsil* y deben de colocarse en los puntos de entrada más comunes.

Aunque la recomendación es llamar a los archivos con la información WSIL *inspection.wsil*, no es obligatorio, el fichero puede llamarse de cualquier modo aunque se desaconseja esta opción por hacer más difícil su localización.

Por ejemplo si se tiene una web de cines (miscines.com), con unos servicios determinados, se puede hacer accesible el archivo con el documento WSIL desde la dirección raíz de la web (`http://www.miscines.com/inspection.wsil`), o puede darse el caso de que se ofrezcan servicios en diversos

puntos, por ejemplo un servicio de venta en <http://www.miscines.com/ventas> y un servicio de críticas cinematográficas en <http://www.miscines.com/criticas>, se podría en este caso, crear un archivo *inspection.wsil* en cada uno de los URL de acceso, y hacer que el archivo accesible desde el URL raíz, sea simplemente un archivo que enlace con cada uno de estos mediante el elemento `<link>`, como se vio en un ejemplo del apartado 5.2.

El problema que se plantea al usar este método es que se tiene que investigar en la web del proveedor, para ver si existe el fichero *inspection.wsil* o no.

5.3.2.2 Documento enlazado

El lenguaje de marcas HTML posee una etiqueta llamada META que permite al autor dar información acerca del documento, y no sobre el formato de documento como la mayoría de las etiquetas de este lenguaje. Toda la información que se dé mediante estas marcas no será visualizada, pero si se podrá ver si se decide ver el código fuente de la página. Por ejemplo una marca META muy utilizada es para indicar el autor de la página:

```
<META name="Author" content="Joaquin Ternet">
```

Además estas marcas sirven de referencia a los *crawlers* de los motores de búsqueda de web como Google.

```
<META name="keywords" lang="es"
```

```
content="programaci&oacute;n, fuentes, utilidades, recursos">
```

En este caso se usará como enlace a los puntos donde se encuentren los distintos documentos WSIL. Si se usara este método en el ejemplo de la web de cine, se podrían incluir dentro de la página principal los siguientes META:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

```
<HTML>
```

```
<HEAD>
```

```
<META name="serviceInspection" content="localservices.wsil">
```

```
<META name="serviceInspection" content="
http://www.miscines.com/ventas/inspection.wsil">
```

```
<META name="serviceInspection" content="
```

```
http://www.miscines.com/criticas/inspection.wsil">
```

```
....
```

```
</HEAD>
```

```
<BODY>
```

```
...
```

```
</BODY>
```

```
</HTML>
```

Como se puede apreciar, ya no hace falta que el nombre del fichero WSIL sea *inspection.wsil*, sino que se puede poner el que se quiera, aunque no deja de ser una buena costumbre utilizar siempre como nombre el estándar de la especificación.

El problema que plantea este método es que las etiquetas META no se pensaron para esta finalidad, y aunque puede ser útil su uso de esta forma, no deja de ser un uso incorrecto de la etiqueta.

5.3.4 Ejemplo de documento WSIL

Para finalizar el capítulo se presenta un documento WSIL que contiene un acceso a dos documentos WSDL. Uno de ellos contiene los servicios encargados del tema de localización de comercios, y el otro es la versión dos del mismo (documentos localizador.wSDL y localizadorV2.wSDL respectivamente).

```
<service>

  <name>Localizador</name>

  <abstract>Localizador de las tiendas con productos
Acme.</abstract>

  <description
referencedNamespace="http://schemas.xmlsoap.org/wSDL/"

    location="http://acme.com/servicios/localizador.wSDL">

    <description
referencedNamespace="http://schemas.xmlsoap.org/wSDL/"

      location="http://acme.com/servicios/localizadorV2.wSDL">

    </description>

  </service>
```

Por otra parte se tiene un documento WSDL localizado en el servidor de FTP, que contiene servicios de conversión entre monedas. Para acceder a él se genera una nueva entrada `<service>`.

```
<service>

  <abstract>Servicios de cambio entre monedas</abstract>

  <description
referencedNamespace="http://schemas.xmlsoap.org/wSDL/"

    location="ftp://ftp.acme.com/util/conversor.wSDL"/>

</service>
```

además se enlaza con uno de los documentos WSIL que se tenían ya realizados para los suministros. Esto quiere decir que el documento tendrá un elemento `<service>` con dos elementos `<description>` y otro elemento `<link>`.

```
<link referencedNamespace=

  "http://schemas.xmlsoap.org/ws/2001/10/inspection/"

  location="http://acme.com/suministros/inspection.wsil">

  <abstract>Acme Suministros</abstract>

</link>
```

Si además se tiene información sobre el servicio de suministros ya introducida en un registro UDDI, también se puede hacer uso de ella. Para esto se introduce un nuevo elemento `<link>`. En este caso se necesitan unos elementos de extensión para poder recuperar la especificación.

Se añade un elemento `<serviceKey>` que permitirá acceder al servicio en concreto, como valor de este elemento tiene que darse el UUID del servicio. Además se ha informado un URL en el elemento `<discoveryURL>`. Mediante este elemento, que es optativo, es posible expresar un URL por la que se obtendrá la estructura `<businessEntity>` mediante una petición HTTP GET :

```
<link referencedNamespace="urn:uddi-org:api">
  <abstract>Acme Suministros</abstract>
  <wsiluddi:serviceDescription
    location="http://reguddi.org/uddi/inquiryapi">
    <wsiluddi:serviceKey>4AA427F0-2C66-1A1B-9A37-BA120FF33F72
      </wsiluddi: serviceKey >
    <wsiluddi:discoveryURL useType="businessEntity">
      http:// reguddi.org/uddi?3A542AD0-2C66-1A1B-9A37-
BA120FF33F72
    </wsiluddi:discoveryURL>
  </wsiluddi: serviceDescription >
</link>
```

El resultado final será:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  xmlns:wsiluddi="http://schemas.xmlsoap.org/ws/2001/10/inspection/uddi/"
  xmlns:wsilwsdl="http://schemas.xmlsoap.org/ws/2001/10/inspection/wsdl/"
">
  <abstract>Productos Acme</abstract>
  <service>
    <name>Localizador</name>
    <abstract>Localizador de las tiendas con productos
Acme.</abstract>
    <description
referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
      location="http://acme.com/servicios/localizador.wsdl">
```

```

    <description
referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
        location="http://acme.com/servicios/localizadorV2.wsdl">
    </description>
</service>
<service>
    <description
referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
        location="ftp://ftp.acme.com/util/conversor.wsdl"/>
</service>
<link referencedNamespace=
"http://schemas.xmlsoap.org/ws/2001/10/inspection/"
        location="http://acme.com/suministros/inspection.wsil">
    <abstract>Acme Suministros</abstract>
</link>
<link referencedNamespace="urn:uddi-org:api">
    <abstract>Acme Suministros</abstract>
    <wsiluddi:serviceDescription
        location="http://reguddi.org/uddi/inquiryapi">
    <wsiluddi:serviceKey>4AA427F0-2C66-1A1B-9A37-BA120FF33F72
        </wsiluddi:serviceKey >
    <wsiluddi:discoveryURL useType="businessEntity">
        http:// reguddi.org/uddi?3A542AD0-2C66-1A1B-9A37-
BA120FF33F72
    </wsiluddi:discoveryURL>
    </wsiluddi:serviceDescription >
</link>
</inspection>

```

Si en el elemento que accede al registro UDDI se quisiera recuperar todos los servicios en lugar de uno en concreto, es posible hacerlo recuperando todos de uno en uno o bien recuperar el elemento

<*businessEntity*> con todos sus servicios, para ello habría que cambiar la etiqueta <*serviceKey*> por <*businessKey*>:

```
<link referencedNamespace="urn:uddi-org:api">
  <abstract>Acme Suministros</abstract>
  <wsiluddi:businessDescription
    location="http://reguddi.org/uddi/inquiryapi">
    <wsiluddi:businessKey>3A542AD0-2C66-1A1B-9A37-BA120FF33F72
      </wsiluddi:businessKey>
    <wsiluddi:discoveryURL useType="businessEntity">
      http:// reguddi.org/uddi?3A542AD0-2C66-1A1B-9A37-
BA120FF33F72
    </wsiluddi:discoveryURL>
  </wsiluddi:businessDescription >
</link>
```

Ejemplos prácticos

6.1. Introducción

En este capítulo se verán una serie de modelos prácticos sobre la programación de web services. Si bien la intención de esta guía no es enseñar al lector a programar servicios web, sino mostrar su funcionamiento, si es lógico ver algunos ejemplos sobre los temas tratados anteriormente.

Aunque el lector no conozca ninguno de los lenguajes de programación empleados para la construcción de los ejemplos, es interesante su lectura por ser sencillos y fáciles de comprender incluso desconociendo la sintaxis de estos lenguajes (aunque es muy recomendable tener nociones básicas de programación), por lo que servirán de referencia para poder realizar los primeros servicios en el lenguaje de programación favorito del lector.

Como podrá comprobar, la mayoría de los lenguajes actuales ofrecen herramientas o módulos para alejar al programador de la tediosa tarea de realizar los mensajes SOAP vistos anteriormente, permitiendo un desarrollo más rápido y sencillo, dejando al programador centrarse en tareas más específicas del negocio.

La lista de implementaciones actuales de SOAP para distintos lenguajes de programación puede obtenerse de la dirección <http://www.software.org/directory/4/implementations>.

Como no podía ser de otra manera, existen implementaciones para los lenguajes de programación más comunes.

Como un servicio web se puede programar en multitud de lenguajes, se planteaba un pequeño problema a la hora de escribir este capítulo: ¿qué lenguaje usar para los ejemplos? Se ha optado por hacer ejemplos en Perl y en Java mayoritariamente (aunque se realizan también clientes en Delphi y Visual Basic), por ser dos lenguajes cuyos entornos de desarrollo, al igual que sus herramientas para generación de web services, son gratuitas (en cuanto a Delphi se puede obtener una versión de evaluación en la web de Borland). Además, el uso de estos lenguajes, permite desarrollar indistintamente en entornos Windows o en Unix, ya que existen implementaciones para ambos.

El hecho de elegir estos entornos de trabajo, no significa que los ejemplos mostrados no puedan portarse a otros sistemas operativos o lenguajes (salvando las distancias de configuración).

Se vio en el primer capítulo, que en el acceso a un web service, entran en juego varias partes. Lo primero que se necesita es un servicio, un programa que sea capaz de realizar algún tipo de operación cuando recibe una petición, este código será el que forme el servidor. Por otra parte se tiene un cliente que desea acceder al servicio. Por último se necesita un medio de unir las dos partes.

El protocolo de comunicación con el web service más extendido es el HTTP, es por ello por lo que será el adoptado en este capítulo (aunque puede usarse cualquier otro), por lo que se necesita también un servidor de páginas web. Una buena opción es el servidor Apache (www.apache.org), por ser gratuito y por existir tanto en Windows como en Unix, aunque si el lector dispone de *Internet Information Server* (o cualquier otro), puede usarlo sin ningún problema más que conocer las rutas equivalentes a las que se usarán en el libro, y sus ficheros de configuración.

6.2. Hola mundo Perl

Desde que se escribiera por primera vez en 1967 usando BCPL, en un libro que trate de lenguajes de programación, no puede faltar un ejemplo que se encargue de saludar al mundo. Esta guía no va a ser la excepción, y se explicará en esta sección como realizar un pequeño web service en Perl que devuelva el mensaje “Hola mundo”.

Lo primero que se necesita es el intérprete Perl completamente configurado. Se puede obtener de Internet (www.perl.org) tanto los binarios como el código fuente (si posee un entorno de desarrollo C es preferible usar el código fuente y compilarlo según sus necesidades, sobre todo si trabaja en un sistema Unix). Además es MUY aconsejable (tanto si se decide usar los binarios precompilados, como si se usa el código fuente) leer detenidamente los ficheros de configuración (sobre todo el `lib/Config.pm` y el

lib/CPAN/Config.pm) y realizar los cambios necesarios, poniendo especial cuidado con las rutas. Por último, también es aconsejable el añadir el directorio de los ejecutables a la variable PATH del sistema.

Para el manejo de los mensajes SOAP, utilizaremos una de las herramientas más extendidas en Perl, el SOAP Lite (www.soaplite.com).

Una de las maneras más rápidas y cómodas de obtener este módulo, es a través del uso del CPAN (Comprehensive Perl Archive Network, Red de todos los archivos de Perl). Perl ofrece una *shell* para poder acceder a esta red de forma cómoda, permitiendo su instalación inmediatamente; pero en cualquier caso, también se puede descargar el módulo usando otros medios como FTP o HTTP y agregarlo más tarde a Perl.

Para obtenerlo mediante CPAN lo primero que debe hacer es conectarse a la red CPAN, para ello escriba el siguiente comando:

```
perl -MCPAN -e shell
```

obteniendo la salida

```
cpan shell -- CPAN exploration and modules installation (v1.48)
```

```
ReadLine support enabled
```

Para comprobar si el módulo ya está instalado puede ejecutar:

```
cpan> i /SOAP::Lite/
```

Si el módulo ya está instalado, obtendrá una salida en pantalla semejante a:

```
Module id = SOAP::Lite
```

```
CPAN_USERID KULCHENKO (Paul Kulchenko <paulclinger@yahoo.com>)
```

```
CPAN_VERSION 0.55
```

```
CPAN_FILE K/KU/KULCHENKO/SOAP-Lite-0.55.zip
```

```
MANPAGE SOAP::Lite - Client and server side SOAP
implementation
```

```
INST_FILE /usr/local/lib/site_perl/SOAP/Lite.pm
```

```
INST_VERSION 0.55
```

mientras que si no está aún instalado, la salida será semejante a:

```
Module id = SOAP::Lite
```

```
CPAN_USERID KULCHENKO (Paul Kulchenko <paulclinger@yahoo.com>)
```

```
CPAN_VERSION 0.55
```

```
CPAN_FILE K/KU/KULCHENKO/SOAP-Lite-0.55.zip
```

```
INST_FILE (not installed)
```

Si no se encuentra instalado, proceda a su instalación mediante el comando CPAN:

```
cpan> install SOAP::Lite
```

tras el cual, si es la primera vez que se usa, le pedirá unos valores para la configuración del sistema, y si ya ha sido utilizado en alguna ocasión, procederá a la descarga del código fuente de SOAP Lite.

Si el usuario ha tenido problemas con la descarga del código mediante CPAN, puede hacerlo utilizando métodos tradicionales (FTP o HTTP) y realizar manualmente las tareas que el *shell* CPAN realiza automáticamente:

```
perl Makefile.PL

make

make test

make install
```

Al finalizar la descarga, se procede a la configuración del módulo, aunque es totalmente válida la configuración por defecto (ver que este activado el protocolo HTTP).

XMLRPC::Lite, UDDI::Lite, and XML::Parser::Lite are included by default.

Installed

transports can be used for both SOAP::Lite and XMLRPC::Lite.

Client (SOAP::Transport::HTTP::Client) [yes]

Client HTTPS/SSL support

(SOAP::Transport::HTTP::Client, require OpenSSL) [no]

Client SMTP/sendmail support (SOAP::Transport::MAILTO::Client) [yes]

Client FTP support (SOAP::Transport::FTP::Client) [yes]

Standalone HTTP server (SOAP::Transport::HTTP::Daemon) [yes]

Apache/mod_perl server (SOAP::Transport::HTTP::Apache, require Apache) [yes]

FastCGI server (SOAP::Transport::HTTP::FCGI, require FastCGI) [no]

POP3 server (SOAP::Transport::POP3::Server) [yes]

IO server (SOAP::Transport::IO::Server) [yes]

MQ transport support (SOAP::Transport::MQ) [no]

JABBER transport support (SOAP::Transport::JABBER) [yes]

MIME messages [required for POP3, optional for HTTP]

(SOAP::MIMEParser) [yes]

SSL support for TCP transport (SOAP::Transport::TCP) [yes]

Compression support for HTTP transport (SOAP::Transport::HTTP) [no]

Do you want to proceed with this configuration? [yes]

6.2.1. Servidor Perl

Dentro de la cadena de elementos que entran en juego en una transacción usando web services, el servidor será el encargado de realizar las operaciones de negocio, o lo que es lo mismo, será el encargado de dar una respuesta ante una petición. Podrá observar el lector, que el hecho de trabajar como servidor no supone un gran cambio en el código fuente respecto a lo que sería trabajar como un programa independiente, por lo que muchas veces es recomendable la codificación de las reglas de negocio como programa independiente, y una vez comprobado que funciona correctamente, transformarlo en web service, ya que como programa independiente será más sencillo de depurar y el ciclo de prueba es menor.

Realice ahora el código del programa que se encargará de saludar al mundo. En la primera versión, se incluirán unas líneas de código para probarlo sin utilizar aún SOAP, es decir, sin realizar el enlace con el servidor web (sin realizar el *deployment*). Estas líneas de prueba tienen una doble finalidad, por un lado le servirá para asegurarse de que el código funciona correctamente y por otro lado le servirá para ver que realmente no hay que hacer grandes cambios para transformar los programas independientes en servicios.

Mediante cualquier editor de texto (texto sin formato), cree el fichero *hola.pl* cuyo código fuente es el siguiente:

```
#!/usr/bin/perl

# Saludos al mundo

package Holamundo;

# funciones

sub dimeHola{

    return "Hola mundo \n";

}

sub dimeAdios{

    return "Adiós mundo injusto \n";

}

print dimeHola;

print dimeAdios;
```

Para probar este primer programa, ejecute:

```
perl hola.pl
```

Y la salida debe ser:

```
Hola mundo
```

```
Adiós mundo injusto
```

Si por alguna causa hubiera dado error, es posible que no esté bien configurada la variable de sistema PATH, y que no tenga acceso al intérprete Perl. Compruebe que se han añadido las rutas del intérprete a la variable PATH, y que estas rutas son correctas.

Una vez probado y comprobado que el programa funciona correctamente, puede realizar el código para poder enlazarlo con el servidor web. La manera más rápida de realizar esta tarea es mediante un CGI (Common Gateway Interface, Interfaz común de salida).

Para ello genere el archivo *hola.cgi* a partir del código realizado anteriormente. El código del nuevo archivo es el siguiente:

```
#!/usr/bin/perl

# Saludos al mundo

use SOAP::Transport::HTTP;

SOAP::Transport::HTTP::CGI

-> dispatch_to('Holamundo')

-> handle;

package Holamundo;

sub dimeHola{

    return "Hola mundo \n";

}

sub dimeAdios{

    return "Adiós mundo injusto \n";

}
```

Es muy importante no omitir la primera línea del código, por que pese a que es un comentario, le sirve de referencia al servidor web para saber que programa debe procesar ese CGI y, en ocasiones, para obtener la localización del interprete, en este caso, la dirección donde está instalado el Perl.

A la hora de procesar la petición HTTP, el archivo CGI tiene que ser accesible por el servidor web, por lo que debe guardarlo en alguno de los directorios donde éste almacene los archivos de tipo CGI. Para encontrar alguno de estos directorios, puede buscar en los archivos de configuración del servidor que en el caso del servidor Apache será un fichero llamado *srm.conf*, donde se puede encontrar una línea semejante a:

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
```

En este servidor, normalmente, el directorio donde se encuentran estos ficheros de configuración, es */etc/apache/conf* o en */usr/local/apache/conf* en sistemas Unix y en *%directorioInstalación%\conf* en sistemas Windows.

Además debe comprobar que el directorio donde se alojarán los ficheros CGI, tenga permisos de ejecución. Para ello en el archivo *access.conf* (también en el directorio *conf* de Apache) debe haber unas líneas semejantes a:

```
<Directory /usr/lib/cgi-bin>

    AllowOverride None

    Options ExecCGI FollowSymLinks

</Directory>
```

Por último el servidor tiene que ser capaz de ejecutar código CGI escrito en Perl. Para ello debe tener instalado el módulo `mod_perl`, siendo indiferente la versión del mismo (actualmente están disponibles las versiones 1.0 y 2.0). Puede ver si está accesible o no revisando el archivo de configuración (*httpd.conf* en Apache). En él debería encontrar una línea como:

```
LoadModule perl_module /usr/lib/apache/1.3/mod_perl.so
```

Si no existe es muy probable que no esté instalado. El hecho de que esté la línea, pero que esté comentada, no significa siempre que esté instalado el módulo. A la hora de configurarlo hay que tener cuidado en escribir correctamente la ruta donde se encuentra el instalado.

En caso de no tener el módulo instalado, se puede obtener en la web de Apache, y añadirlo posteriormente a la instalación del servidor.

Una vez se haya asegurado de que la configuración es correcta, ya puede copiar el archivo *hola.cgi* al directorio que ha configurado para la utilización de programas CGI. Tras copiar el archivo compruebe que éste tiene permisos de lectura y ejecución. En estos momentos ya está configurado el web service y preparado para ser utilizado.

6.2.3. Cliente Perl

Una vez creado el servidor, se necesita un cliente para poder probarlo. Este programa será el encargado de conectarse al servidor web y enviarle los mensajes SOAP que se generarán para la petición de ejecución de alguna tarea por parte del servidor.

No importa el lenguaje que se haya utilizado en la generación del código del servidor, pero en este caso, el cliente se realizará también en Perl. En el siguiente apartado se demostrará mediante el uso de Java que no importa realmente los lenguajes elegidos.

Cree un nuevo fichero llamado *holaCliente.pl* con el siguiente código fuente:

```
#!/usr/bin/perl -w

# holaCliente.pl - Cliente del servicio holaMundo

use SOAP::Lite;

print "\nPrimera llamada: hola\n";

print "Respuesta del servicio: ";

print SOAP::Lite

-> uri('http://localhost/Holamundo')

-> proxy('http://127.0.0.1/cgi-bin/hola.cgi')

-> dimeHola

-> result . "\n\n";

print "\nSegunda llamada: adiós\n";

print "Respuesta del servicio: ";

print SOAP::Lite

-> uri('http://localhost/Holamundo')

-> proxy('http://127.0.0.1/cgi-bin/hola.cgi')

-> dimeAdios
```

```
-> result . "\n\n";
```

Mediante *proxy()*, se informa la dirección sobre la que se realizarán las peticiones, por lo que si el lector ejecuta el cliente en una máquina distinta de la que aloja al servidor web, deberá variar la dirección del proxy, pudiendo especificarse tanto como un nombre válido, como una IP válida.

En *uri()* se debe especificar la clase que tiene atender la petición, ya que es posible que una misma dirección *proxy* atienda varios servicios a la vez.

Al ejecutar este programa debe obtener la siguiente salida:

```
Primera llamada: hola
```

```
Respuesta del servicio: Hola mundo
```

```
Segunda llamada: adiós
```

```
Respuesta del servicio: Adiós mundo injusto
```

Si se produjeran errores, habría que mirar que estuviera bien instalado el SOAP Lite, y que éste fuera accesible al servidor web.

6.2.4. Cliente en Java

Para comprobar que realmente pueden realizarse clientes con casi cualquier lenguaje de programación (en teoría con cualquiera), se realizará ahora un cliente mediante Java. Para poder desarrollar con Java se necesita el SDK (Software Development Kit) de Sun. En este caso sirve la edición estándar que se puede obtener en la propia web de Sun dedicada a Java (<http://java.sun.com>).

Una vez instalado, se requerirán también unas librerías que permitan trabajar con mayor comodidad con mensajes SOAP. Las librerías que se utilizarán en los ejemplos, son las correspondientes a Apache SOAP, que se pueden obtener en <http://xml.apache.org/soap/index.html>. Una vez instaladas en el sistema, deberá incluirlas en la variable de sistema CLASSPATH. La forma de realizarlo varía de un sistema operativo a otro, e incluso entre las distintas *shell*. En Windows sería mediante:

```
set CLASSPATH = %CLASSPATH%;%HOME_SOAPL%\soap.jar
```

y en Unix (shell Bourne)

```
CLASSPATH = $CLASSPATH:$HOME_SOAPL/soap.jar
```

```
export CLASSPATH
```

Una vez configurada esta variable, podrá acceder a estas clases desde Java. Así pues ya se puede proceder a la compilación del código fuente del cliente Java. Cree un nuevo directorio llamado *holapclient* (si no usa *package* en Java no se necesario) y en el genere un archivo llamado *HolaPerlCliente.java* que corresponderá al cliente. Su código es el siguiente:

```
package holapclient;
```

```
import java.net.URL;
```

```
import org.apache.soap.Fault;
```

```
import org.apache.soap.rpc.Response;
```

```
import org.apache.soap.rpc.Call;
```

```
import org.apache.soap.Constants;
```

```
import org.apache.soap.rpc.Parameter;
```

```
public class HolaPerlCliente {  
    public HolaPerlCliente() {  
    }  
  
    public static void main(String[] args) throws Exception {  
        HolaPerlCliente holaPerlClientel = new HolaPerlCliente();  
  
        System.out.println("\n\nLlamada al servicio\n\n");  
  
        System.out.println(getResponse());  
    }  
  
    public static String  getResponse() throws Exception{  
        URL url = new URL ("http://127.0.0.1/cgi-bin/hola.cgi");  
        Call call = new Call ();  
        call.setTargetObjectURI("http://localhost/Holamundo");  
        call.setMethodName("dimeHola");  
        String strReturn = "Respuesta del servicio: ";  
        Response rsp = call.invoke(url, "");  
        if (rsp.generatedFault()) {  
            Fault fault = rsp.getFault();  
            strReturn += " Error en la llamada.\n";  
            strReturn += "-> Código de error = " + fault.getFaultCode() +  
            "\n";  
            strReturn += "-> Descripción del error = " +  
            fault.getFaultString()+ "\n";  
        } else {  
            strReturn += rsp.getReturnValue().getValue();  
        }  
        return strReturn;  
    }  
}
```

```
}

```

Puede observar que la semejanza (en los contenidos) con el código anteriormente realizado en Perl para la creación del cliente es muy elevada, el hecho es que se llama al mismo servicio, por lo que los parámetros se configuran de modo semejante, con los mismos valores, pero en este caso usando Java.

Es cierto que la cantidad de código de este ejemplo es mucho mayor que el de Perl, pero en este caso, se ha añadido al código la lógica necesaria para controlar los posibles errores que se pudieran dar en la comunicación con el servicio.

Compile y ejecute el programa de la siguiente manera (si se está trabajando en Windows, debe cambiar la barra “/” por “\”):

```
javac holaplclient/HolaPerlCliente.java

```

```
java holaplclient.HolaPerlCliente

```

La salida que se debería obtener es:

```
Llamada al servicio

```

```
Respuesta del servicio: Hola mundo

```

En este ejemplo sólo se ha generado el código necesario para realizar la llamada a la función *dimeHola*, pero la llamada a la función *dimeAdios* se realizaría de modo semejante, simplemente debería sustituir:

```
call.setMethodName("dimeHola");

```

por

```
call.setMethodName("dimeAdios");

```

Para hacer más versátil el código de este ejemplo, también se podría haber pasado como parámetros tanto la dirección, como el uri o el nombre del método, y de esta forma hacer que este cliente fuera “universal” (siempre y cuando las llamadas devolvieran una cadena de texto). En este caso, para ejecutar el programa desde línea de comando, debería proporcionar los parámetros de dirección, uri y método a invocar (en este orden). Las modificaciones que habría que hacer en el código fuente son mínimas:

...

```
// en el main

```

```
System.out.println(getResponse(args[0], args[1], args[2]));

```

...

```
public static String getResponse(String surl, String suri, String
smetodo) throws Exception{

```

```
    URL url = new URL (surl);

```

```
    Call call = new Call ();

```

```
    call.setTargetObjectURI(suri);

```

```
    call.setMethodName(metodo);

```

Y se invocaría :

```
java holaplclient.HolaPerlCliente http://127.0.0.1/cgi-bin/hola.cgi
http://localhost/Holamundo dimeHola

```

6.3. *euroConversor Perl*

Ya se ha visto cómo se puede generar un web service mediante Perl y cómo usarlo mediante programas realizados con otros lenguajes. El problema del programa anterior es que el hecho que sólo diga “hola mundo”, no es algo que se pueda considerar muy útil, así pues, se verá ahora otro ejemplo en Perl, pero esta vez un poco más completo; en este caso será necesario el paso de parámetros. Este nuevo servicio web será un conversor de monedas. Como aún somos varios los que no manejamos bien el euro, el conversor hará cambios entre pesetas y euros, aunque fácilmente el lector podrá adaptarlo a nuevas conversiones o añadir mejoras como el redondeo (que en este ejemplo no se realiza).

6.3.1. Servidor Perl

Tal y como se hizo en el primer ejemplo Perl, haga primeramente un programa independiente. Cree un nuevo archivo y llámelo *euro.pl*. En él se introducirá la lógica necesaria para realizar el cambio, así como unas líneas de prueba, que más tarde se utilizarán para la generación del código del cliente. El contenido del fichero *euro.pl* es el siguiente:

```
#!/usr/bin/perl -w

# euro.pl - Conversor de euros a pesetas y viceversa

package EuroConversor;

#rutina de conversión de euros a pesetas

sub euroToPts {

    my $euro = shift(@_);

    my $pts=$euro*166.386;

    return $pts;

}

#rutina de conversión de pesetas a euros

sub ptsToEuro {

    my $pts = shift(@_);

    $euro=$pts/166.386;

    return $euro;

}

# líneas de código para realizar pruebas

my $cantidad = shift;
```

```

my $modo = shift;

if ($modo eq 'p'){

    print "\n Ha elegido euro -> pesetas \n";

    my $pts = euroToPts ($cantidad);

    print "\n" . $cantidad . " euros son " . $pts . " pesetas\n\n";

} else {

    print "\ Ha elegido pesetas -> euro \n";

    my $euro = ptsToEuro ($cantidad);

    print "\n" . $cantidad . " pesetas son " . $euro . " euros\n\n";

}

```

Para probarlo ejecute:

```
perl euro.pl <cantidad> <modo>
```

Donde el parámetro <cantidad> representa el importe de dinero que se desea convertir, y modo es 'p' para transformar la cantidad de euros a pesetas y cualquier otra opción para transformar la cantidad de pesetas a euro. Una vez comprobado que funciona correctamente debe transformarlo en un módulo, para lo cual cree un archivo llamado *EuroConversor.pm* e introduzca las siguientes líneas (que como verá son bastante parecidas a las del archivo *euro.pl*):

```

#!/usr/bin/perl -w

# EuroConversor.pm - Conversor de euros a pesetas y viceversa

package EuroConversor;

#rutina de conversión de euros a pesetas

sub euroToPts {

    shift;

    my $euro = shift;

    my $pts=$euro*166.386;

    return $pts;

}

#rutina de conversión de pesetas a euros

```



```

sub ptsToEuro {
    my ($class, $pts) = @_; #puede realizarse de la forma anterior

    $euro=$pts/166.386;

    return $euro;
}

1;

```

Este módulo será accedido por el CGI escrito en Perl, por lo que se tiene que almacenar en algún directorio que Perl use para guardar sus librerías. Si no conoce ninguno de esos directorios, puede consultar a Perl mediante:

```
perl -e print@INC
```

Si no tuviera privilegios suficientes para la escritura en ninguno de estos directorios, puede guardar el archivo en cualquier otro directorio, pero en este caso, tendrá que indicarle al CGI donde encontrar el módulo. Por ejemplo si guarda el archivo en un directorio llamado *perlModules* dentro del directorio */homes/user001*, la línea que tendría que incluir en el CGI sería:

```
use lib '/homes/user001/perlModules' ;
```

Para la generación del archivo CGI, cree un nuevo documento llamado *euro.cgi* e introduzca las siguientes líneas:

```

#!/usr/bin/perl -w

# Cambio de moneda

# quitar comentario a la siguiente línea si hiciera falta

# use lib '/homes/user001/perlModules' ;

use SOAP::Transport::HTTP;

SOAP::Transport::HTTP::CGI

-> dispatch_to('EuroConversor')

-> handle;

```

Este archivo CGI, debe copiarlo en el directorio que utilice el servidor web para albergar los programas CGI (en el mismo directorio que utilizó en el primer ejemplo).

En este caso el código del programa CGI es mucho menor, puesto que está separada la lógica de negocio (la conversión entre monedas) en un módulo totalmente independiente. El parámetro que se pasa a *dispatch_to()*, debe ser el nombre del *package* que hemos realizado anteriormente, en nuestro caso será “*EuroConversor*”.

6.3.2. Cliente Perl

Una vez realizado el servidor, proceda a la generación del código del cliente. Éste se basa en el programa inicial que hizo de prueba, sólo que en este caso las llamadas a las conversiones son remotas, por lo que no incluye el código de implementación de éstas. Además, por cuestiones de comodidad, se añadirá un “menú” de selección, y así ya no será necesario introducir los datos de los parámetros por línea de comando.

En este caso hay que reseñar que las dos posibles llamadas al servicio (conversión euro a peseta y peseta a euro) esperan un parámetro, que representa la cantidad a cambiar. La información de éste parámetro se realiza de modo semejante a como se haría en una función.

```
-> euroToPts($cantidad).
```

Cree un nuevo documento y llámelo *euroCliente.pl*. El código a introducir en él es el siguiente:

```
#!/usr/bin/perl

#euroCliente.pl - Cliente de conversión de euros

use SOAP::Lite;

print "\n*****Conversión de moneda*****\n";

my $modo = "";

print "\n\nPulse 'p' para euro a peseta o 'e' para peseta a euro\n\n";

print "Su elección: ";

$modo = <STDIN>;

print "\nCantidad de dinero a convertir: ";

my $cantidad = <STDIN>;

if ($modo == "p"){

    print "\n Ha elegido euro -> pesetas \n";

    #rutina de conversión de euros a pesetas

    my $pts = SOAP::Lite

        -> uri('http://localhost/EuroConversor')

        -> proxy('http://localhost/cgi-bin/euro.cgi')

        -> euroToPts($cantidad)

        -> result;

    print "\n" . $cantidad . " euros son " . $pts . " pesetas\n\n";

} else {

    #rutina de conversión de pesetas a euros

    print "\ Ha elegido pesetas -> euro \n";

    my $euro = SOAP::Lite
```

```

-> uri('http://localhost/EuroConversor')

-> proxy('http://localhost/cgi-bin/euro.cgi')

-> ptsToEuro($cantidad)

-> result;

    print "\n" . $cantidad . " pesetas son " . $euro . " euros\n\n";
}

```

Para ejecutarlo use el comando:

```
perl euroCliente.pl
```

El resultado de la ejecución de este programa será:

```

*****Conversion de moneda*****

Pulse 'p' para euro a peseta o 'e' para peseta euro

Su elección: p

Cantidad de dinero a convertir: 10

```

```
Ha elegido euro -> pesetas
```

```
10 euros son 1663.86 pesetas
```

6.3.3. Cliente Java

Al igual que en el primer ejemplo, se hará en esta sección un cliente en Java para ver como se realiza el paso de parámetros hacia el web service. El código es muy semejante al del cliente del primer ejemplo, con la diferencia que en éste se proporciona un vector de parámetros (clase *Parameter*). La línea correspondiente a este paso es:

```
param.addElement (new Parameter("cantidad", String.class, pts,
Constants.NS_URI_SOAP_ENC));
```

En este caso se tendrá que definir previamente la variable pts.

Cree un nuevo archivo en un directorio llamado *euoplclient* y llámelo *EuroPerlCliente.java* e introduzca el código completo de este cliente Java que se muestra a continuación:

```

package euoplclient;

import java.net.URL;

import java.util.Vector;

import org.apache.soap.Fault;

import org.apache.soap.rpc.Response;

```

```
import org.apache.soap.rpc.Call;

import org.apache.soap.Constants;

import org.apache.soap.rpc.Parameter;

public class EuroPerlCliente {

    public EuroPerlCliente () {

    }

    public static void main(String[] args) throws Exception {

        EuroPerlCliente holaPerlClientel = new EuroPerlCliente ();

        System.out.println("\n\nLlamada al servicio\n\n");

        System.out.println(getResponse(args[0]));

    }

    public static String  getResponse(String pts) throws Exception{

        URL url = new URL ("http://192.168.1.1/cgi-bin/euro.cgi");

        Call call = new Call ();

        call.setTargetObjectURI("http://localhost/EuroConversor");

        call.setMethodName("ptsToEuro");

        // parámetros

        Vector param = new Vector ( );

        param.addElement (new Parameter("cantidad", String.class, pts,
        Constants.NS_URI_SOAP_ENC));

        call.setParams (param);

        String strReturn = "Respuesta del servicio: ";

        Response rsp = call.invoke(url, "");

        if (rsp.generatedFault()) {

            Fault fault = rsp.getFault();
```

```

    strReturn += " error en la llamada.\n";

    strReturn += "-> Código de error = " + fault.getFaultCode() +
"\n";

    strReturn += "-> Descripción del error = " +
fault.getFaultString()+ "\n";

} else {

    strReturn += rsp.getReturnValue().getValue();

}

return strReturn;
}
}

```

Tras compilarlo, se puede ejecutar dando como parámetro el importe a cambiar. Si lo ejecuta dando como cantidad 166:

```
java euoplclient.EuroPerlCliente 166
```

La salida que debería obtener es:

Llamada al servicio

Respuesta del servicio: 0.99768007

6.3.4. Cliente Delphi

Se verá ahora otro ejemplo de cliente para el servicio de conversión, en este caso se realizará en Delphi. Delphi proporciona unas herramientas muy potentes para la generación de servicios web (sobre todo en su última versión, la 7) que más adelante se usarán, pero en este caso se va a realizar el proyecto del cliente a mano, es decir sin uso de importaciones de documentos WSDL.

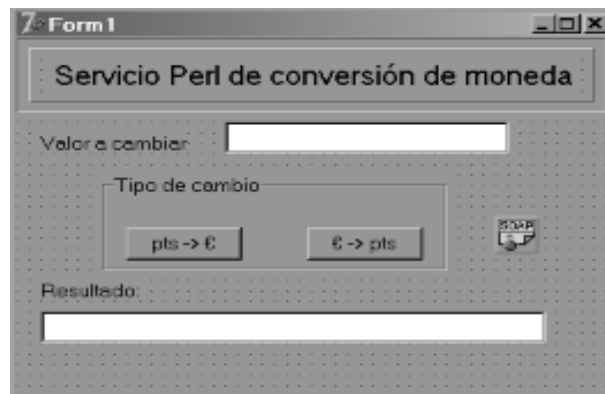


Figura 13: Edición de la ventana del cliente Delphi

Para comenzar el nuevo proyecto, elija File > New > Application o Archivo > Nuevo > Aplicación (dependiendo de la versión se tenga de Delphi). Sobre la ventana de edición gráfica añada los elementos necesarios para que quede como la figura 13.

El componente que puede parecer más extraño al lector será el THTTTPRIO (ver figura 2), que se encuentra en la pestaña correspondiente a los web services. Este componente se encargará de hacer el

trabajo sucio de las conexiones y serializado del mensaje. En este componente debe asignar la dirección a la que accederá para la obtención del servicio, en este caso hay que informar la propiedad *URL* con *http://direccionDestino:puertoDestino/cgi-bin/euro.cgi*.

El THTTPRIO puede operar de dos formas distintas, bien mediante la propiedad *URL* o bien mediante la propiedad *WSDLLocation*, que nos permite indicar directamente un documento WSDL.

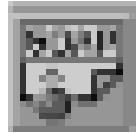


Figura 14: Detalle del componente THTTPRIO

Edite las propiedades de las cajas de texto y dé como nombre a la superior *valorEdt* y a la inferior llámela *cambioEdt*. En cuanto a los botones, llame *ptsAEBtn* al botón que servirá para pasar de pesetas a euros y *eAPtsBtn* al que servirá para pasar de euros a pesetas.

Cree una nueva unidad mediante File > New > Unit o Archivo > Nuevo > Unidad, y llámela *EuroIF*. El código de la unidad se muestra a continuación.

```
unit EuroIF;

interface
uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;
type
    Euro = interface(IInvokable)
        ['{00000000-0000-0000-0000-000000000000}']
        function ptsToEuro(const cantidad: WideString): WideString;
        stdcall;
        function euroToPts(const cantidad: WideString): WideString;
        stdcall;
    end;

function GetEuroServer(HTTPRIO: THTTPRIO = nil): Euro;

implementation

function GetEuroServer(HTTPRIO: THTTPRIO): Euro;
```

```

var
    RIO: THTTTPRIO;

begin
    Result := nil;

    if HTTPRIO = nil then
        RIO := THTTTPRIO.Create(nil)
    else
        RIO := HTTPRIO;

    try
        Result := (RIO as Euro);
    finally
        if (Result = nil) and (HTTPRIO = nil) then
            RIO.Free;
        end;
    end;

initialization
    InvRegistry.RegisterInterface(TypeInfo(Euro),
    'http://localhost/EuroConversor', 'UTF-8');

    InvRegistry.RegisterDefaultSOAPAction(TypeInfo(Euro), '');

end.

```

Ahora debe enlazar esta interfaz con el componente THTTTPRIO que añadió en el diseño y con el resto de la aplicación. Para ello, en la unidad principal, debe añadir la entrada *euroIF* en la sección *uses*.

```

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms, Dialogs, euroIF, StdCtrls, InvokeRegistry, Rio, SOAPHTTPClient,
    ExtCtrls;

```

Y por último, genere la lógica para la llamada del servicio y los eventos de los botones en la unidad principal:

```

type

```

```
private
    { Private declarations }
public
    function GetService: Euro;
end;

(...)

var
    euroFrm: TeuroFrm;
implementation

{$R *.dfm}
// obtención del servicio
function TeuroFrm.GetService: Euro;
begin
    Result := HTTPRIO1 as Euro;
end;

// evento click del botón pesetas a euros
procedure TeuroFrm.ptsAEBtnClick(Sender: TObject);
begin
    cambioEdt.Text := GetService.ptsToEuro(valorEdt.Text);
end;

// evento click del botón euros a pesetas
procedure TeuroFrm.eAPtsBtnClick(Sender: TObject);
begin
    cambioEdt.Text := GetService.euroToPts(valorEdt.Text);
end;
```


end.

El programa totalmente terminado y funcionando tendría un aspecto semejante a:

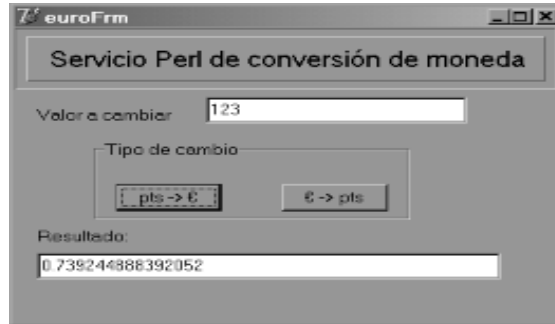


Figura 15: Cliente Delphi en acción

6.3.5. Cliente Visual Basic

Como último ejemplo de cliente para esta aplicación se realizará un pequeño código en Visual Basic. Lo primero que necesita es instalar el SDK para XML de Microsoft que puede encontrar en el CDROM que acompaña la guía y el entorno de desarrollo de Visual Basic.

En este ejemplo se formará el mensaje SOAP a enviar, es decir, escribiremos exactamente el mensaje que se transmitirá. Para ello genere un nuevo proyecto, añada un botón al formulario y como código de este botón añada lo siguiente:

```
Private Sub Command1_Click()

Dim xmlObj, httpObj

Set xmlObj = CreateObject("MSXML2.DOMDocument")

xmlObj.loadXML "<s:Envelope
xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/1999/XMLSchema'><s:Body><m:ptsToEuro
xmlns:m='http://localhost/EuroConvertor'><cantidad
xsi:type='xsd:string'>1000</cantidad></m:ptsToEuro></s:Body></s:Envelope>"

MsgBox xmlObj.xml, , "Mensaje SOAP de petición"

Set httpObj = CreateObject("Microsoft.XMLHTTP")

httpObj.open "POST", "http://192.168.1.1/cgi-bin/euro.cgi"

httpObj.send (xmlObj)

While httpObj.readyState <> 4

Wend

MsgBox httpObj.responseText, , "Mensaje SOAP respuesta"
```

End Sub

Al igual que en ocasiones anteriores, debe tener cuidado con las direcciones y con el URI.

Al ejecutar este programa y pulsar sobre el botón, deberá mostrar una ventana con el texto correspondiente al mensaje de petición, al igual que muestra la figura 4.

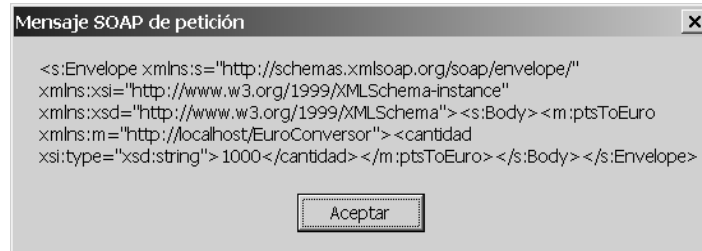


Figura 16: Mensaje de petición.

Tras unos instantes, mostrará el aviso con la respuesta.



Figura 17: Mensaje de respuesta

Si el lector no posee el entorno de programación para Visual Basic, puede crear un archivo de texto con el código (exceptuando la primera y la última línea) y guardarlo como *clienteVB.vbs*. Al hacer doble click sobre icono de este nuevo fichero, se ejecutará y obtendrá los mismos resultados que obtendría con Visual Basic.

6.4. Hola Mundo Java

Una vez vista la manera de realizar servicios web mediante el lenguaje Perl, se verán ahora unos ejemplos utilizando Java. La cantidad de código y operaciones es bastante mayor que la de Perl, e incluso más engorrosa, pero Java tiene otros puntos fuertes (que quedan fuera del alcance de este libro).

Para aquellos que estén acostumbrados a trabajar en entornos J2EE, decir que es totalmente factible el uso de Enterprise Java Beans para la generación de web services, y que se puede realizar el desarrollo de una aplicación web completa, que soporte tanto páginas *jsp* como web service. Existen muchas empresas volcadas en el mundo Java (como BEA <http://www.bea.com>) que integran web services desde el principio

en todos sus productos, ofreciendo entornos completos de desarrollo orientados hacia ellos (Bea Weblogic Workshop).

El hecho de que el libro sea una pequeña guía de iniciación a los web services, obliga a que los ejemplos utilizados en este apartado sean pequeñas clases capaces de responder a las peticiones del cliente, pero decir que para aquellos que estén interesados en construir verdaderas aplicaciones web, puede descargarse directamente de Sun el paquete de desarrollo para web services que éstos ofrecen (<http://java.sun.com/webservices/downloads/webservicespack.html>), que incluye entre otras utilidades interesantes, un servidor de registro UDDI sobre base de datos Xindice (también proporcionada) y un explorador de servicios web.

Antes de pasar a realizar la codificación de los servicios, debe configurar el sistema para que sea capaz de funcionar correctamente.

Para trabajar con Java lo primero que necesita es el SDK, que está disponible en la página web de Sun relacionada con Java (<http://java.sun.com>), que ya lo tendrá instalado si realizó los ejemplos anteriores de clientes en Java. Es aconsejable configurar algunos aspectos como el CLASSPATH, o añadir el directorio de los ejecutables a la variable de entorno PATH.

Por otra parte, necesitará un servidor web capaz de trabajar con *servlets* (y *Java Server Pages*), como por ejemplo el servidor Tomcat, que es parte de un proyecto de Apache llamado *Jakarta*. Este servidor está disponible en <http://jakarta.apache.org/tomcat/>. Al igual que en otras ocasiones, puede obtener tanto los binarios (Windows y Unix) como el código fuente; aunque siempre será mejor obtener el código fuente y compilar sobre el ordenador en el que va a trabajar, pero por el contrario, será mucho más fácil utilizar los binarios precompilados, ya que solamente hay que descomprimirlos, para poder trabajar con ellos. Además se le debe dotar a este servidor la posibilidad de trabajar con Apache SOAP, por lo se realizará un despliegue (*deploy*) de la aplicación *Apache SOAP Admin* sobre el servidor. En el caso de utilizar Tomcat vale con copiar el archivo *soap.war* que viene con el paquete Apache SOAP, en el directorio *webapps* del servidor. Para otros servidores, es recomendable leer las instrucciones de despliegue de aplicaciones que acompañan tanto a la documentación del Apache SOAP como a la del propio servidor, sobre todo si se tiene pensado utilizar EJB, ya que algunos servidores tienen configuraciones especiales en estos casos.

Asimismo necesitará algunas librerías para el lado del cliente. Entre ellas se encuentra también Apache SOAP, que se utilizó en el primer ejemplo a la hora de codificar el cliente Java y anteriormente para configurar el servidor. Por último, dependiendo de la funcionalidad del cliente, necesitará algunas librerías como *mail.jar*, *activation.jar*, y un analizador XML (XML parser) compatible con los *namespaces*, por ejemplo el Apache Xerces, que incluye SAX 2 y nivel DOM 2. Este analizador puede obtenerse en <http://xml.apache.org/xerces-j/>.

Para que todas estas clases sean accesibles por Java debe incorporarlas a la variable de entorno CLASSPATH. En Windows sería mediante:

```
set CLASSPATH = %SOAP_LIB%\xerces.jar;%CLASSPATH%

set CLASSPATH = %CLASSPATH%;%SOAP_LIB%\activation.jar

set CLASSPATH = %CLASSPATH%;%SOAP_LIB%\mail.jar

set CLASSPATH = %CLASSPATH%;%SOAP_LIB%\soap.jar
```

En Bourne shell se haría mediante:

```
CLASSPATH = $SOAP_LIB/xerces.jar;$CLASSPATH

CLASSPATH = $CLASSPATH;$SOAP_LIB/activation.jar

CLASSPATH = $CLASSPATH;$SOAP_LIB/mail.jar

CLASSPATH = $CLASSPATH;$SOAP_LIB/soap.jar
```

Se debe tener cuidado en el orden en el que se realiza la incorporación de las clases dentro del CLASSPATH, ya que es posible que tenga instalado algún otro analizador XML no compatible con los

namespaces, y el sistema utilice éste en lugar del Xerces, obteniendo resultados inesperados. Por esto se recomienda añadir el Xerces como primera entrada dentro del CLASSPATH, como puede verse en los ejemplos anteriores.

Si al compilar la clase falla alguna librería, es aconsejable revisar los posibles fallos comenzando por el CLASSPATH del sistema, ya que es posible que falte algún componente, o bien que el orden de incorporación de las clases en esta variable, obligue a utilizar una clase no deseada.

También puede instalar todas estas clases usando CVS (Concurrent Versions System, sistema de versiones concurrentes), ya que es más rápido y cómodo. Por ejemplo para obtener de este modo Xerces, escribiría:

```
set CVSROOT=:pserver:anoncvs@cvs.apache.org:/home/cvspublic
cvs login (password: anoncvs)
cvs checkout -d xerces_j xml-xerces/java
```

6.4.1. Servidor Java

Al igual que se hizo con el programa servidor Perl, se realizará primero un pequeño programa independiente para asegurarse de que funciona correctamente, y luego se transformará en el servidor. Para ello cree la clase *HolaServer* en un fichero llamado *HolaServer.java* en un directorio llamado *holawebservice*. En esta clase se incluirán las funciones que atenderán las peticiones del cliente. El código de la clase es el que se muestra a continuación:

```
package holawebservice;

public class HolaServer {

    public HolaServer() {

    }

    public static void main(String[] args) {

        HolaServer hs = new HolaServer();

        System.out.println(getHola("Joan"));

        System.out.println(getAdios("Joan"));

    }

    public static String getHola(String aQuien){

        return "Hola " + aQuien + ", soy tu servicio web."

    }

    public static String getAdios(String aQuien){

        return "Lo siento " + aQuien + " pero he de abandonar este mundo.";

    }

}
```

Para probarlo, compílolo y ejecute:

```
java holawebsevice.HolaServer
```

La salida obtenida debe ser:

```
Hola Joan, soy tu servicio web.
```

```
Lo siento Joan pero he de abandonar este mundo.
```

Una vez compilado y comprobado que funciona correctamente como programa independiente, retoque el código para que funcione como servidor. Se puede observar en el código siguiente, que en el servidor no hace falta la ayuda de ninguna clase de manejo de SOAP, ya que de esto se encargará totalmente Apache SOAP. El código del servidor debe ser:

```
package holawebsevice;

public class HolaServer {

    public static String getHola(String aQuien){

        return "Hola " + aQuien + ", soy tu servicio web";

    }

    public static String getAdios(String aQuien){

        return "Lo siento " + aQuien + " pero he de abandonar este mundo.";

    }

}
```

Una vez compilado, ha de hacer que esta clase (fichero *.class) sea accesible al servidor Tomcat, por lo que debe copiarla en alguno de los directorios que estén incluidos en su CLASSPATH (el CLASSPATH del servidor). Normalmente podrá utilizar alguno de los tres directorios siguientes: *tomcat_dir_base/classes*, *tomcat_dir_base/common/classes* o *tomcat_dir_base/server/classes*, dependiendo de la configuración del servidor y de la utilidad de las clases.

En estos momentos, aunque haya copiado el fichero dentro del CLASSPATH del servidor, éste no tiene conocimiento del servicio, para esto tendrá que registrarlo. El registro del web service, se realiza mediante el envío de un mensaje SOAP al servicio de administración SOAP, que es el que se encarga de gestionarlo. Este administrador es una aplicación que se ha instalado anteriormente cuando se copió el archivo *soap.war* en el servidor (ver punto anterior).

En todo momento puede ver que servicios están registrados en el servidor mediante el comando:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter list
```

donde *http://localhost:8080/soap/servlet/rpcrouter* es el URL del encaminador (router) de llamadas remotas de Apache. Es posible que al realizar esta acción, en algunos casos (dependiendo de la configuración del servidor) sea necesario suministrar un nombre de usuario y una clave. Si no se obtiene ningún tipo de respuesta, compruebe que el servidor está realmente en marcha.

En el caso que no exista ningún servicio registrado, la salida del comando anterior será:

```
Deployed Services:
```

si ya hay alguno, la salida mostraría algo semejante:

```
Deployed Services:
```

```
urn:mimetest
```

```
http://www.soapinterop.org/Bid
```

```
urn:attser
```

```
urn:fileUpload
```

Todos los comandos que se realizan desde la línea de comando mediante la clase *ServiceManagerClient*, pueden realizarse accediendo al servicio de administración del servidor, que está en la dirección `http://localhost:8080/soap/admin/index.html`. Desde esta página, se pueden controlar los servicios, añadiendo nuevos, listando los existentes o borrándolos.

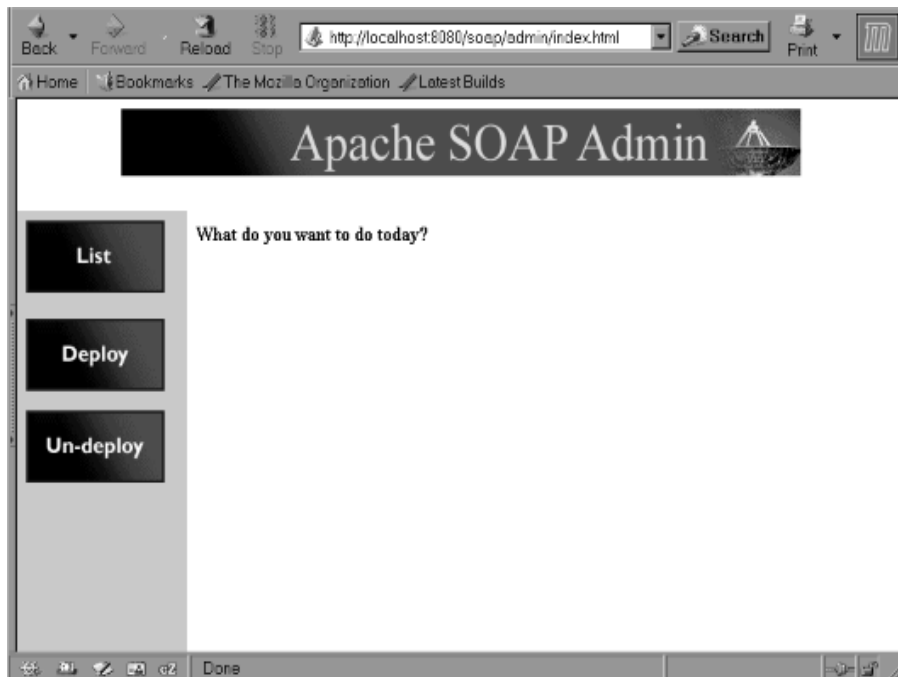


Figura 18: Índice del administrador SOAP.

El archivo de descripción del servicio, como no podía ser de otra forma, está realizado mediante XML. En él se detallan aspectos obligatorios como el nombre de la clase que atenderá el servicio y aspectos optativos como si ésta es estática o no.

Para este primer servicio cree un fichero descriptor llamado *holaDeployment.xml* cuyo contenido será el mostrado a continuación:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment "
    id="urn:holawebservice">
    <isd:provider type="java"
        scope="Application"
        methods="getHola getAdios">
    <isd:java class="holawebservice.HolaServer" />
```

```

</isd:provider>

<isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>

</isd:service>

```

Donde el *id* será el nombre del servicio. El atributo *scope* representa el tiempo que debe durar “viva” la instancia de la clase, pudiendo ser :

- Request: la instancia se “mata” tras concluir la llamada
- Session: se destruye al acabar la sesión HTTP
- Application: perdura hasta que termina el *servlet* que da el servicio

En el atributo *methods* se informarán (separados por espacios en blanco), los métodos que proporciona este servicio. El elemento `<java>`, contiene el atributo *class*, que será el que servirá para especificar la clase que debe atender el servicio.

Existen otros atributos como por ejemplo *MustUnderstand* para especificar si se debe comprender o no el elemento (este atributo se vio en el capítulo dos), u otros para detallar si la clase es estática o no, pero su utilización no es obligatoria.

Si se utilizaran otro tipo de servicios, como por ejemplo servicios realizados mediante Enterprise Java Beans (EJB) o Bean Scripting Framework (BSF), se tienen que usar descriptores diferentes, pero que no se verán en esta guía.

Para realizar el envío de la información del registro del servicio al administrador SOAP, use de nuevo la clase *ServiceManagerClient*, pero esta vez con otros parámetros:

```

java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy holaDeployment.xml

```

Si todo ha ido correctamente, al hacer de nuevo el listado de los servicios registrados, debería salir el servicio hola mundo. Para comprobarlo consulte nuevamente los servicios registrados usando el administrador web o desde línea de comando. Deberá aparecer el servicio *holawebservice*.

```

java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter list

```

Deployed Services:

```

urn:mimetest

http://www.soapinterop.org/Bid

urn:holawebservice

urn:attser

urn:fileUpload

```



Figura 19: Lista de servicios accesibles.

Para comprobarlo desde el administrador vale con pulsar sobre el botón **List**. Puede obtener detalles de cada uno de los servicios registrados, pulsando sobre ellos.



Figura 20: Detalle del servicio.

Para obtener el descriptor del servicio mediante línea de comando, se realiza mediante la opción *query*, por ejemplo, para ver los detalles del servicio de esta forma, se ejecutará el comando:


```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter query urn:holawebbservice
```

La salida obtenida:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
id="urn:holawebbservice" checkMustUnderstands="false">

  <isd:provider type="java" scope="Application" methods="getHola
getAdios">

    <isd:java class="holawebbservice.HolaServer" static="false"/>

  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faul
tListener>

</isd:service>
```

6.4.2. Cliente Java

Cuando las aplicaciones se complican, y se comienza a trabajar con datos que no son simples, tales como objetos, entonces se deben utilizar interfaces específicos para la generación del cliente, o registrar nuevas clases para la serialización de los datos. Como en este caso sólo se trabaja con cadenas de texto, no es necesario. Este cliente no dista mucho de los vistos anteriormente, si bien en este caso no su URL no se refiere a un CGI, sino al del *router* RPC de SOAP.

Para crear este cliente, cree un nuevo fichero llamado *HolaJavaCliente.java* e introduzca el código mostrado a continuación:

```
package holajavac;

import java.net.URL;

import java.util.Vector;

import org.apache.soap.Fault;

import org.apache.soap.rpc.Response;

import org.apache.soap.rpc.Call;

import org.apache.soap.Constants;

import org.apache.soap.rpc.Parameter;

public class HolaJavaCliente {

  public static void main(String[] args) throws Exception {

    HolaJavaCliente jcl = new HolaJavaCliente();

    System.out.println("\n\nLlamada al servicio\n\n");

    System.out.println(getResponse());
```

```
}

public static String  getResponse() throws Exception{

    String aQuien = "Manolo";

    URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");

    Call call = new Call ();

    call.setTargetObjectURI("urn:holawebservice");

    call.setMethodName("getHola");

    Vector param = new Vector ( );

    param.addElement (new Parameter("aQuien", String.class, aQuien,
    Constants.NS_URI_SOAP_ENC));

    call.setParams (param);

    String strReturn = "Respuesta del servicio: ";

    Response rsp = call.invoke(url, "");

    if (rsp.generatedFault()) {

        Fault fault = rsp.getFault();

        strReturn += " error en la llamada.\n";

        strReturn += "-> Código de error = " + fault.getFaultCode() +
"\n";

        strReturn += "-> Descripción del error = " +
fault.getFaultString()+ "\n";

    } else {

        strReturn += rsp.getReturnValue().getValue();

    }

    return strReturn;

}

}
```

Tras compilarlo y ejecutarlo mediante

```
java holajavac.HolaJavaCliente
```

debe obtener:

Llamada al servicio

Respuesta del servicio: Hola Manolo, soy tu servicio web.

Si se han obtenido errores en la ejecución, compruebe que realmente está bien escrita tanto la dirección del *router* RPC como el nombre del servicio.

Es posible ver los mensajes SOAP que está recibiendo el mensaje, mediante una herramienta incorporada en las clases Apache SOAP. Esta utilidad se llama *TcpTunnelGui* y al estar realizada en Java, funciona tanto en sistemas Windows como en Unix. Lo que hace es escuchar los mensajes sobre la máquina local para después enviarlos a su destino. La forma de invocar esta utilidad es:

```
java org.apache.soap.util.net.TcpTunnelGui puerto_de_escucha
host_a_redireccionar puerto_host
```

Por ejemplo, si se quieren escuchar los mensajes que enviaremos a la dirección `http://www.acme.com`, y queremos escuchar en el puerto 82 de nuestra máquina, se ejecutaría:

```
java org.apache.soap.util.net.TcpTunnelGui 82 http://www.acme.com 80
```

En el CD se incluye otra aplicación llamada *tcpTrace* cuya misión es idéntica a la de *TcpTunnelGui*. *tcpTrace* sólo funciona en sistemas Windows.

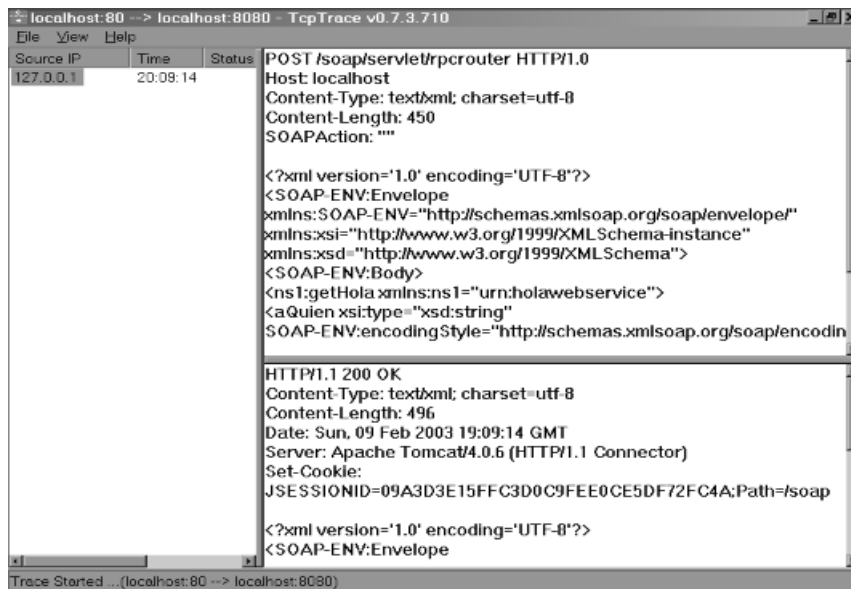


Figura 21: Programa *tcpTrace* funcionando.

6.4.3. Cliente Perl

Nuevamente comprobará que el uso de un lenguaje en concreto para generar un servicio, no obliga a la utilización del mismo lenguaje para la creación del cliente. En este caso el cliente será escrito en Perl.

Al igual que se hizo con Java, en el caso del servidor Perl, solamente se probará en el ejemplo, una de las funciones del servicio. Además en este caso se generará código para el control de errores.

Cree un nuevo fichero llamado *holajCliente.pl* e inserte las líneas siguientes:

```
use SOAP::Lite;

print "\nPrimera llamada: hola\n";

print "Respuesta del servicio: ";

my $soapMess = SOAP::Lite
-> uri("urn:holawebservice")
-> proxy("http://127.1:8080/soap/servlet/rpcrouter");

my $result = $soapMess->getHola("Joan");

unless ($result->fault) {
    print $result->result() . "\n\n";
} else {
    print join ', ',
        $result->faultcode,
        $result->faultstring;
}
```

Para probarlo, ejecute el comando

```
perl holajCliente.pl
```

La salida de la ejecución a este programa será:

```
Primera llamada: hola
```

```
Respuesta del servicio: Hola Joan , soy tu servicio web.
```

6.4.4. El WSDL

Ya se vio en capítulos anteriores que el documento WSDL es una herramienta que ofrece unas ventajas considerables a la hora de generar servicios web. En este caso se va a crear un WSDL del servicio hola mundo de Java, para utilizarlo más adelante en la generación de un cliente Delphi.

Sería recomendable que el lector comparara cada parte del siguiente documento, con la teoría sobre WSDL que se vio en el capítulo cuatro, ya que al haber creado también el código, puede resultar un ejemplo muy educativo.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="urn:holawebservice"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:intf="urn:holawebservice"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```

xmlns:wsdlssoap="http://schemas.xmlsoap.org/wsdlssoap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"

<types>

  <schema targetNamespace="urn:holawebsevice"
    xmlns="http://www.w3.org/2001/XMLSchema"

    <complexType name="ArrayOf_SOAP-ENC_string">

      <complexContent>

        <restriction base="SOAP-ENC:Array">

          <attribute ref="SOAP-ENC:arrayType"
            wsdl:arrayType="xsd:string[]" />

        </restriction>

      </complexContent>

    </complexType>

    <element name="ArrayOf_SOAP-ENC_string" nillable="true"
      type="intf:ArrayOf_SOAP-ENC_string" />

  </schema>

</types>

<wsdl:message name="getAdiosRequest">

  <wsdl:part name="aQuien" type="SOAP-ENC:string" />

</wsdl:message>

<wsdl:message name="getAdiosResponse">

  <wsdl:part name="return" type="SOAP-ENC:string" />

</wsdl:message>

<wsdl:message name="getHolaRequest">

  <wsdl:part name="aQuien" type="SOAP-ENC:string" />

</wsdl:message>

<wsdl:message name="getHolaResponse">

  <wsdl:part name="return" type="SOAP-ENC:string" />

</wsdl:message>

```

```

<wsdl:portType name="HolaServer">
  <wsdl:operation name="getHola" parameterOrder="aQuien">
    <wsdl:input message="intf:getHolaRequest"/>
    <wsdl:output message="intf:getHolaResponse"/>
  </wsdl:operation>
  <wsdl:operation name="getAdios" parameterOrder="aQuien">
    <wsdl:input message="intf:getAdiosRequest"/>
    <wsdl:output message="intf:getAdiosResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="HolaServerSoapBinding" type="intf:HolaServer">
  <wsdlsoap:binding style="rpc" transport=
    "http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getHola">
    <wsdlsoap:operation soapAction="" style="rpc"/>
    <wsdl:input>
      <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:holawebservice" use="encoded"/>
    </wsdl:input>
    <wsdl:output>
      <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:holawebservice" use="encoded"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="getAdios">
    <wsdlsoap:operation soapAction="" style="rpc"/>
    <wsdl:input>

```

```

        <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:holawebservice" use="encoded"/>

        </wsdl:input>

        <wsdl:output>

                <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:holawebservice" use="encoded"/>

                </wsdl:output>

        </wsdl:operation>

</wsdl:binding>

        <wsdl:service name="HolaServerService">

                <wsdl:port binding="intf:HolaServerSoapBinding"
name="HolaServer">

                        <wsdlsoap:address
location="http://192.168.1.2:8080/soap/servlet/rpcrouter"/>

                </wsdl:port>

        </wsdl:service>

</wsdl:definitions>

```

6.4.5. Cliente Delphi

A lo largo de la guía se ha dicho muchas veces que los documentos WSDL sirven, entre otras cosas, para agilizar la codificación de los clientes. En esta ocasión se va a generar el cliente a partir del WSDL que se ha creado en la sección anterior.

Inicie para ello un proyecto nuevo en Delphi mediante File > New > Application o Archivo>Nuevo> Aplicación. Una vez que obtenida la nueva aplicación, importe el documento WSDL. Para ello elija el menú File > New >Other... o Archivo > Nuevo > Otros...

En este momento aparecerá un cuadro de diálogo en el que se muestran distintas posibilidades.

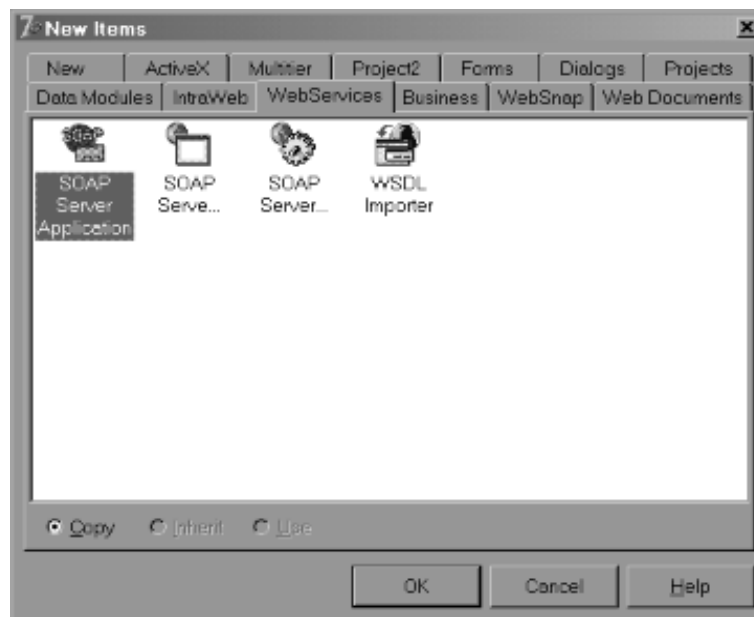


Figura 22: Selección del asistente de importación de WSDL.

Debe elegir la opción *WSDL Importer* (importador de WSDL). Tras seleccionar esta opción, se abre una nueva ventana en la que el asistente le pregunta sobre la localización del documento. En este caso será el fichero generado anteriormente, que estará localizado en algún lugar del disco duro, pero podría también optar por hacer búsquedas mediante UDDI, como en la figura doce.

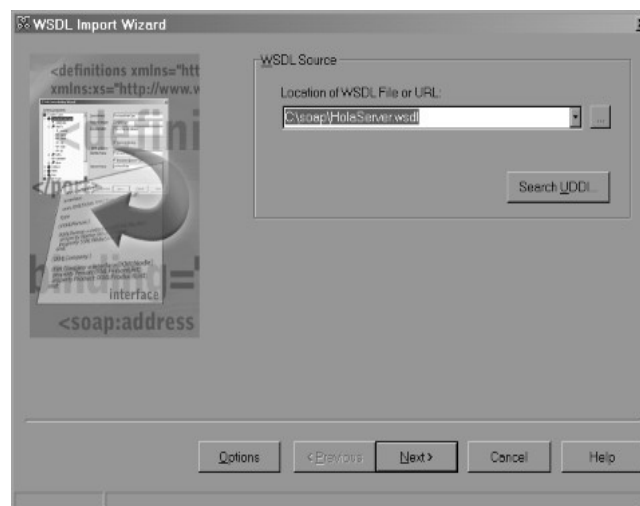


Figura 23: Localización del documento WSDL

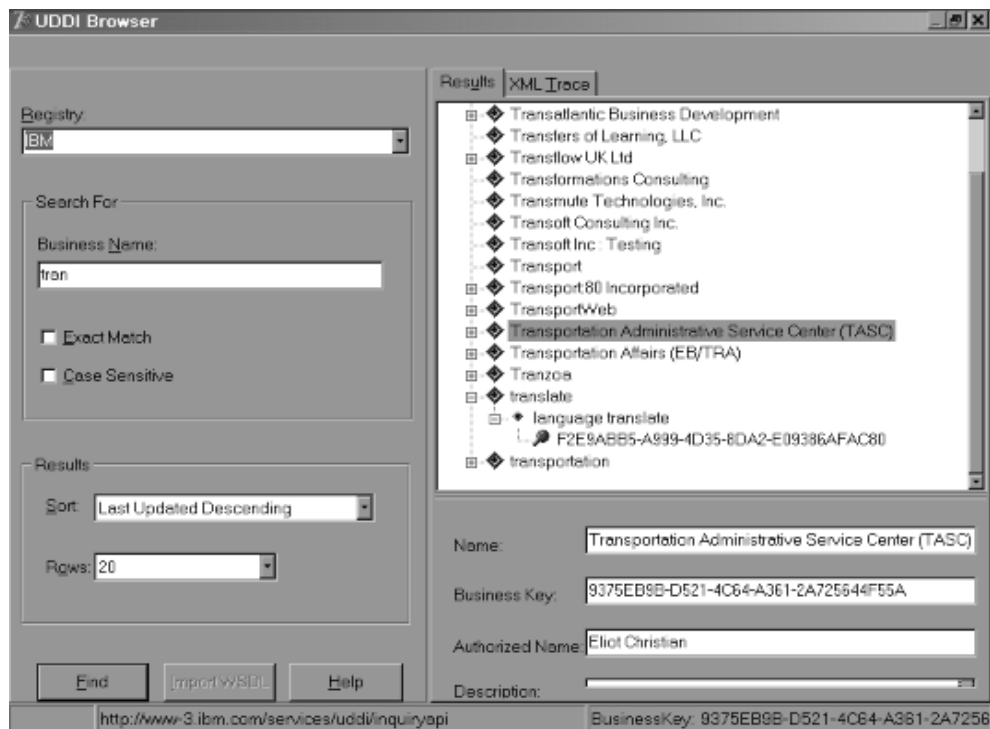


Figura 24: Selección de WSDL mediante UDDI.

Tras seleccionar la fuente de origen del documento WSDL, Delphi muestra la información acerca de los métodos que hay accesibles, y realiza una vista preliminar del código que se generará de forma automática.

Si pulsa sobre el botón **Aceptar**, Delphi creará una nueva unidad con el código mostrado en la pantalla del asistente.

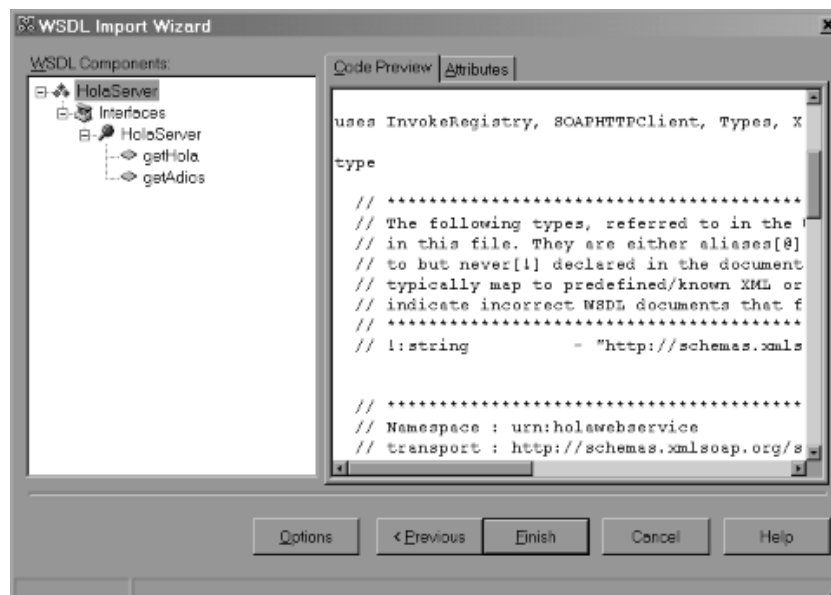


Figura 25: Vista preliminar del código automático generado por Delphi.

El código generado por Delphi es:

```
//
*****
** //

// The types declared in this file were generated from data read from
the

// WSDL File described below:

// WSDL      : C:\soap\HolaServer.wsdl

// Encoding  : UTF-8

// Version   : 1.0

// (29/01/2003 17:47:32 - 1.33.2.5)

//
*****
** //

unit HolaServer1;

interface

uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;

type

    //
    *****
    ** //

    // The following types, referred to in the WSDL document are not
being represented

    // in this file. They are either aliases[@] of other types
represented or were referred

    // to but never[!] declared in the document. The types from the
latter category
```

```

// typically map to predefined/known XML or Borland types; however,
they could also

// indicate incorrect WSDL documents that failed to declare or
import a schema type.

//
*****
** //

// !:string          - "http://schemas.xmlsoap.org/soap/encoding/"

//
*****
** //

// Namespace : urn:holawebservice
// transport : http://schemas.xmlsoap.org/soap/http
// style      : rpc
// binding    : HolaServerSoapBinding
// service    : HolaServerService
// port       : HolaServer
// URL        : http://127.0.0.1:8080/soap/servlet/rpcrouter

//
*****
** //

HolaServer = interface(IInvokable)

['{EE417C49-FA39-CBDA-20F4-DC47588AF72D}']

function getHola(const aQuien: WideString): WideString; stdcall;

function getAdios(const aQuien: WideString): WideString; stdcall;

end;

function GetHolaServer(UseWSDL: Boolean=System.False; Addr: string='';
HTTPRIO: THTTPRIO = nil): HolaServer;

implementation

function GetHolaServer(UseWSDL: Boolean; Addr: string; HTTPRIO:
THTTPRIO): HolaServer;

const

defWSDL = 'C:\soap\HolaServer.wsdl';

```

```
defURL = 'http://127.0.0.1:8080/soap/servlet/rpcrouter';
defSvc = 'HolaServerService';
defPrt = 'HolaServer';

var
    RIO: THTTTPRIO;

begin
    Result := nil;
    if (Addr = '') then
    begin
        if UseWSDL then
            Addr := defWSDL
        else
            Addr := defURL;
    end;
    if HTTPRIO = nil then
        RIO := THTTTPRIO.Create(nil)
    else
        RIO := HTTPRIO;
    try
        Result := (RIO as HolaServer);
        if UseWSDL then
        begin
            RIO.WSDLLocation := Addr;
            RIO.Service := defSvc;
            RIO.Port := defPrt;
        end else
            RIO.URL := Addr;
    finally
```

```

    if (Result = nil) and (HTTPRIO = nil) then

        RIO.Free;

    end;

end;

initialization

    InvRegistry.RegisterInterface(TypeInfo(HolaServer),
'urn:holawebservice', 'UTF-8');

    InvRegistry.RegisterDefaultSOAPAction(TypeInfo(HolaServer), '');

end.

```

Para poder hacer uso de esta nueva unidad, hay que proceder como se hizo anteriormente con el cliente para Perl, a grandes rasgos, hay que hacer accesible a este interfaz mediante su inclusión en la zona *uses* de la unidad principal, y tras ello generar la lógica de los botones para realizar las llamadas a las que da acceso el servicio. Puede encontrar el código fuente del ejemplo completo en el CDROM que acompaña la guía.

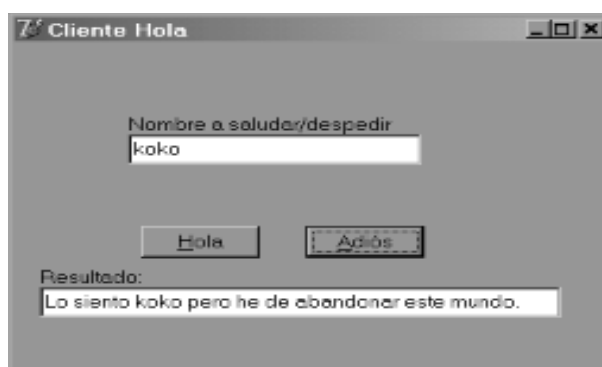


Figura 26: Cliente Delphi en acción.

6.5. Attachments

Con el nombre de *attachment* (atadura, anexo) se conoce a todos aquellos datos que de alguna forma, van ligados al documento. Un ejemplo de su uso es en los correos electrónicos cuando se envía por ejemplo una fotografía, ésta es un *attachment*.

Al igual que en los mensajes SOAP, en los correos electrónicos sólo se puede enviar información en modo texto, ya que dicha información puede pasar por varios servidores antes de llegar a su destino, y puede que alguno de estos servidores, sea antiguo, o no soporte algún tipo de codificación; es por esto por lo que se buscó la forma de hacer compatible el uso del correo con las nuevas necesidades de transmisión de datos binarios. Algún lector quizá recuerde los comandos *uuencode* y *uudecode* de Unix, con los que se conseguía transformar datos binarios en texto plano ASCII y viceversa.

La aparición del estándar MIME (Multipurpose Internet Mail Extensions, extensiones multipropósito para correo)(RFC2045- RFC2049) supuso un gran avance en la transmisión de datos anexados. MIME da las pautas de cómo realizar este envoltorio de datos binarios sobre texto plano, y añadirlo al mensaje a modo de complemento. Este anexo se enviará de manera conjunta con el resto del mensaje.

Esta es la opción que se ha tomado en la transmisión de algunos datos en SOAP (ficheros por ejemplo, aunque teóricamente puede ser cualquier tipo de datos, incluso de usuario), ya que aunque hay formas de codificar los datos binarios en texto (base64 por ejemplo), son métodos bastante lentos, sobretodo con ficheros de gran tamaño, ya que el analizador SOAP, debe examinarlos previamente.

En este último ejemplo, se realizará un servicio que acepte dos parámetros, uno de ellos será el nombre que se quiere dar al fichero en el servidor y el segundo será el fichero en si (realmente será una instancia de la clase *DataHandler*). De esta forma, cuando el servidor reciba la petición, lo que hará será guardar el fichero enviado usando el nombre que se le ha dado como parámetro. Además se le pondrá al fichero la extensión *bak* (caprichos del autor).

6.5.1. Servidor

El uso de Apache SOAP, le proporciona la ventaja de poder tratar los ficheros de forma semejante a cualquier otro tipo de dato nativo, ya que posee serializadores que se encargan de todo el trabajo. En la declaración de la función puede ver los dos tipos que se usarán como parámetros.

```
public boolean guardaFichero(String nombre, DataHandler handler)
throws Exception {
```

En este caso *DataHandler* es la clase que “contiene” el fichero, y deberá ser el cliente el que se encargue de enviar no una instancia de clase *File*, sino de *DataHandler*.

Una vez que se tenga en el servidor la instancia *javax.activation.DataHandler*, y de saber que no es nula, se obtendrá el *javax.activation.DataSource* que necesita para la creación de la instancia *org.apache.soap.util.mime.ByteArrayDataSource*. En la creación del objeto *ByteArrayDataSource* necesitará saber la naturaleza del fichero que ha sido enviado; esto se puede conseguir mediante el método *getContentType()* de la clase *DataHandler*. Una vez creado el objeto *ByteArrayDataSource*, se está en condiciones de guardar el fichero en disco. Esto se hará mediante el método *writeTo()* de la clase *ByteArrayDataSource*, que tiene como parámetro un *java.io.FileOutputStream*.

Cree un nuevo fichero y llámelo *GFServer.java*. Este fichero hará de servidor. Añada las líneas de código siguientes:

```
package gf.srv;

import java.io.FileOutputStream;

import javax.activation.*;

import org.apache.soap.util.mime.*;

public class GFServer {

    public GFServer() {

        init();

    }

    private void init () {

        // añadir inicializadores de servicio

    }

    public boolean guardaFichero(String nombre, DataHandler handler)
throws Exception {

        boolean rtnval = true;
```

```

    try {
        if (handler != null) {
            String fname = nombre + ".bak"; // creamos el nombre del
fichero

            DataSource ds = handler.getDataSource( ); // obtenemos el
DataSource del fichero

            ByteArrayDataSource bsource =
                new ByteArrayDataSource(ds.getInputStream( ),
                    handler.getContentType( ));

            bsource.writeTo(new FileOutputStream(fname)); // se escribe a
disco
        }

        return rtnval;
    }

    catch (Exception e){
        rtnval = false;
    }

    return rtnval;
}
}

```

Una vez compilada la clase, se debe colocar en un directorio de forma que sea accesible por el servidor, tal y como se realizó en el ejemplo anterior.

Para registrarlo debe crear un fichero XML con la descripción del servicio. Para ello genere un documento nuevo llamado *ficheroDeployment.xml* y añada las siguientes líneas:

```

<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
            id="urn:archMime">
    <isd:provider type="java"
                scope="Request"
                methods="guardaFichero">
        <isd:java class="gf.srv.GFServer" static="false"/>
    </isd:provider>
</isd:service>

```

Para realizar el registro debe ejecutar:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy
ficheroDeployment.xml
```

Puede comprobar que se ha registrado correctamente mediante:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter list
```

6.5.2. Cliente

En el cliente se debe generar la instancia *DataHandler* que se enviará como parte del mensaje SOAP. Para generar dicha instancia, necesita previamente un *ByteArrayDataSource* que haya sido creado con el fichero que se quiere enviar, esto se hace en las líneas.

```
DataSource ds = new ByteArrayDataSource(new File("\\texto.txt"),
null);
```

```
DataHandler dh = new DataHandler(ds);
```

En este caso se enviará el fichero llamado *texto.txt* que está en el directorio raíz del sistema Windows (el hecho de poner dos barras invertidas es que en Java este carácter se usa para escapar otros caracteres, por lo que se tiene que escapar a si mismo para poder ser evaluado). Si se está probando en un sistema Unix, recuerde que el separador de directorios no es “\” sino “/”, y deberá escribir solamente una, no como en Windows. El segundo parámetro del constructor de la clase *ByteArrayDataSource* es una cadena con el tipo del contenido, pero en este caso no hace falta, por lo que se le ha dado valor nulo.

El primer parámetro introducido es el nombre que se le dará al fichero en el servidor (se podrían incluir directorios), y en este caso se le ha dado el valor de prueba.

Por lo demás el código es bastante semejante a los vistos anteriormente.

Para crear el cliente, proceda a generar un nuevo fichero llamado *GFCliente.java* con el siguiente contenido:

```
package gf.cli;

import java.io.File;

import java.util.Vector;

import java.net.URL;

import org.apache.soap.util.mime.*;

import org.apache.soap.*;

import org.apache.soap.encoding.*;

import org.apache.soap.encoding.soapenc.*;

import org.apache.soap.rpc.*;

import javax.activation.*;

import javax.mail.internet.*;

import org.apache.soap.util.xml.*;
```



```
public class GFCliente {

    public GFCliente() {

    }

    public static void main(String[] args) throws Exception{

        URL url = new URL("http://localhost:8080/soap/servlet/rpcrouter");

        Call call = new Call( );

        call.setTargetObjectURI("urn:archMime");

        call.setMethodName("guardaFichero");

        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        Vector params = new Vector( );

        params.addElement (new Parameter("nombre", String.class, "prueba",
Constants.NS_URI_SOAP_ENC));

        DataSource ds = null;

        try {

            ds = new ByteArrayDataSource(new File("\\texto.txt"), null);

        }

        catch (Exception e){

            e.printStackTrace();

            System.out.println("Error en la apertura del fichero indicado.
Compruebe que existe");

        }

        DataHandler dh = new DataHandler(ds);

        params.addElement(new
Parameter("handler", javax.activation.DataHandler.class, dh, null));

        call.setParams(params);

    }

}
```

```
Response resp;

try {
    resp = call.invoke(url, "");
}

catch (SOAPException e) {
    System.out.println(e.getMessage( ));

    return;
}

if (!resp.generatedFault( )) {
    Parameter ret = resp.getReturnValue( );

    if (((Boolean)ret.getValue()).booleanValue()){
        System.out.println("La transmisión se realizó correctamente");
    }

    else{
        System.out.println("Hubo problemas al guardar el fichero");
    }
}

else {
    String strError = null;

    Fault fault = resp.getFault();

    strError = "Error en la llamada.\n";

    strError += "-> Código de error = " + fault.getFaultCode() +
"\n";

    strError += "-> Descripción del error = " +
fault.getFaultString()+ "\n";

    System.out.println(strError);
}
}
}
```

Para probar este cliente, compílelo y ejecútelo con el comando:

```
java gf.cli.GFCliente
```

Si la transmisión fue correcta, tendremos el nuevo fichero en el directorio elegido y con el nombre indicado, además la salida obtenida debe ser:

```
La transmisión se realizó correctamente
```

Si se obtuvieron fallos en la transmisión, compruebe que el archivo de la ruta especificada en el cliente realmente existe.

6.6. Otras consideraciones

Aunque en estos ejemplos se ha utilizado HTTP como protocolo transporte principal de mensajes, ya se comentó que nada impide el uso de otro tipo de protocolos como por ejemplo FTP. Para hacerse una idea de los protocolos que se pueden usar por ejemplo en Perl, se puede pedir información a CPAN sobre los transportes disponibles para SOAP

```
kappa:~# perl -MCPAN -e shell
```

```
cpan shell -- CPAN exploration and modules installation (v1.48)
```

```
ReadLine support enabled
```

```
cpan> i /SOAP::TRANSPORT/
```

```
(...)
```

```
Module SOAP::Transport::HTTP::ForkingDaemon (K/KU/KULCHENKO/SOAP-Lite-0.46.tar.gz)
```

```
Module SOAP::Transport::HTTP::Server (K/KB/KBROWN/SOAP-0.28.tar.gz)
```

```
Module SOAP::Transport::HTTPX (D/DY/DYACOB/SOAP-Lite-SmartProxy-0.11.tar.gz)
```

```
Module SOAP::Transport::IO (K/KU/KULCHENKO/SOAP-Lite-0.55.zip)
```

```
Module SOAP::Transport::JABBER (K/KU/KULCHENKO/SOAP-Lite-0.55.zip)
```

```
Module SOAP::Transport::LOCAL (K/KU/KULCHENKO/SOAP-Lite-0.55.zip)
```

```
Module SOAP::Transport::MAILTO (K/KU/KULCHENKO/SOAP-Lite-0.55.zip)
```

```
Module SOAP::Transport::MQ (K/KU/KULCHENKO/SOAP-Lite-0.55.zip)
```

```
Module SOAP::Transport::POP3 (K/KU/KULCHENKO/SOAP-Lite-0.55.zip)
```

```
Module SOAP::Transport::TCP (K/KU/KULCHENKO/SOAP-Lite-0.55.zip)
```

Ante tanta cantidad de protocolos... ¿cuál es el mejor? Depende de la utilidad del servicio. Por ejemplo si es un servicio que tiene que ser de respuesta inmediata no sería lógico el uso de SMTP para su transmisión, mientras que si no importa la velocidad y se envían ficheros de tamaño considerable si parece lógico usar SMTP. En cada caso habrá que valorar los pros y los contras de usar uno u otro método.

Igualmente, los archivos que en el apartado anterior se han enviado como un añadido al mensaje, se podrían haber enviado perfectamente como una cadena de texto dentro del propio <Body>, simplemente

haciendo una codificación base64 del dato, pero esto implica que el motor SOAP deberá procesar todo este fichero para incluirlo dentro del mensaje, mientras que si se usa una transmisión con *attachment*, no es necesario analizar esta información, por lo que se realizará en menor tiempo y consumiendo menos recursos. También puede que en ciertas circunstancias sea más útil el uso de esta otra técnica, por lo que no se la debe perder de vista.

Por último reseñar que hay que tener muy en cuenta el tema de las direcciones, ya que al trabajar con web services, el cliente tiene que enviar su petición a una dirección y un puerto concreto, y en ocasiones no se informa bien este apartado, dando como resultado que el cliente no es capaz de conseguir una respuesta correcta. Además el URI debe también coincidir, por lo que cuando sea posible se debería usar WSDL para evitar estos fallos.

Encaminamiento

7.1. Introducción

Como se ha visto en capítulos anteriores, la comunicación con los web services se hace mediante mensajes SOAP, que son fácilmente interpretables, esto deja al descubierto una falta de seguridad importante, ya que dichos mensajes pueden ser interceptados y leídos. Existen varios mecanismos para ocultar los datos a posibles lecturas (o al menos dificultar) por parte de terceras personas, ajenas a la transacción. La posibilidad de guiar el mensaje por unos nodos determinados es uno de estos posibles mecanismos. Este método permite, por ejemplo, conocer de antemano el nivel de seguridad que estos nodos poseen, y permitir que los distintos actores (nodos) efectúen algún tipo de operación sobre el mensaje, proporcionándole además un valor añadido. Aunque pueda parecer suficiente esta forma de emisión de mensajes para procurar su seguridad, se verá que en ocasiones, no lo es, por lo que en el capítulo siguiente se explicarán otros métodos para ocultar datos mediante encriptación.

Esta capacidad de elegir previamente el camino de nodos que debe seguir el mensaje es denominado routing (rutado) o encaminamiento

7.2. WS-Routing

Dentro del entorno de trabajo de mensajes, parece lógico que el rutado se produzca también a nivel de mensaje, y a ser posible, dentro del propio documento SOAP. El lugar ideal para la implementación de estas capacidades es dentro de la sección `<Header>` (cabecera) dentro del `<Envelope>` del documento SOAP.

Un aspecto que debe tenerse muy en cuenta, es que en el estándar WS-Routing se definen mensajes de un solo sentido, por lo que para conseguir comunicaciones en los dos sentidos, hay que dotar al mensaje no solamente del camino de emisión, sino que también en su caso, hay que definir el camino de recepción, o dicho de otra forma, hay que definir el camino de ida y el de vuelta.

Antes de continuar, merece la pena describir algunos conceptos, que se encontrará en este protocolo.

- **WS-Routing sender (web service – emisor del mensaje rutado)**
Se refiere a la aplicación que genera el mensaje que será encaminado y enviado. El emisor inicial es el primer emisor.
- **WS-Routing receiver (web service – receptor del mensaje encaminado)**
Es la aplicación encargada de recibir el mensaje rutado que ha sido enviado mediante algún protocolo de transmisión, y se encarga de procesarlo. El último receptor es el destinatario final del mensaje.
- **WS-Routing message path (web service – camino del mensaje)**
Conjunto de emisores y receptores del mensaje rutado que componen conjuntamente el camino que seguirá el mensaje. Se definen dos tipos de camino, el de emisión o petición (*forward*) y el de recepción o respuesta (*reverse*).
- **Forward WS-Routing message path (web service – camino de ida del mensaje)**
Camino recorrido por el mensaje desde el emisor inicial hasta el receptor final, incluyendo los elementos (cero o más, sin límite) que actúan como intermediarios entre ellos.
- **Reverse WS-Routing message path (web service – camino de retorno del mensaje)**
Camino que utilizarán el o los mensajes de respuesta para retornar al emisor inicial; estos mensajes los emitirá el receptor final del primer mensaje transferido. El camino de retorno es generado de forma automática durante la transmisión inicial del mensaje.

- **WS-Routing intermediary (web service – intermediario de rutado)**
Aplicación que es capaz de actuar tanto como emisor y como receptor. Su utilidad es la de procesar y reenviar el mensaje continuando con la ruta indicada por el emisor inicial. Los intermediarios pueden ser de dos tipos, túnel o proxy.
- **WS-Routing tunnel (web service – túnel de rutado)**
La aplicación que actúa como túnel, es en realidad una comunicación ciega entre dos canales. Una vez activa, pasa a no ser considerado parte de esta comunicación. Para que un intermediario actúe como túnel, debe ser explícitamente indicado.
- **WS-Routing proxy (web service – proxy de rutado)**
Es el intermediario que toma parte activa en la comunicación. En este caso, el intermediario realiza algún tipo de acción en el transcurso de la comunicación del mensaje entre emisor inicial y receptor final. En algún momento tiene el papel de receptor, y más tarde de emisor.
- **WS-Routing gateway (web service – puerta de rutado)**
Receptor que se encarga de realizar las traducciones de la cabecera del rutado del mensaje hacia un protocolo no soportado por SOAP, por ejemplo es posible transformar el mensaje para ser utilizado mediante IRC o CORBA.

En la figura 1 se puede ver un esquema de una transmisión rutada entre cuatro nodos. En ella el nodo A es el emisor inicial del mensaje encaminado, y el nodo D es el receptor final. Los nodos B y C son los intermediarios del encaminamiento, y en algún momento de la comunicación actuarán como emisor y receptor, por ejemplo, el nodo B será el receptor del mensaje proveniente de A y emisor del mensaje que tiene como destino C.

7.2.1. Mecanismos de rutado

El modo de proporcionar la ruta al sistema es mediante la adición de una nueva entrada en la cabecera del mensaje. En ella se escriben las rutas de envío y en ocasiones las de recepción.

La ruta de envío tiene un emisor, un punto de recepción del mensaje y ningún o varios elementos intermediarios. En cuanto a la ruta de retorno, su definición es totalmente optativa. Normalmente, la ruta es totalmente especificada por el emisor, pero bajo ciertas circunstancias, es posible que los intermediarios introduzcan nuevos intermediarios; pero jamás podrán variarse ni el emisor inicial ni el receptor final definidos en un principio.

La función de los intermediarios es la de añadir valor al mensaje a través de procesos propios o ajenos, como por ejemplo poder interactuar con otros servicios, generar una caché de mensajes, etc. también pueden ser utilizados para conseguir accesos a otras redes, por ejemplo a través de cortafuegos de aplicación.

Cuando los intermediarios introducen a su vez nuevos intermediarios en las rutas hay que tener mucho cuidado para no crear bucles infinitos. En el caso de posibles bucles infinitos, se puede incluir una etiqueta `<Expires>` (se verá en el próximo capítulo) o devolver el mensaje con un error.

Veamos un pequeño ejemplo en el que se tiene un emisor, en este caso `www.A.com` y se quiere enviar el mensaje a través de dos nodos (`www.B.com` y `www.C.com`) hasta el nodo final que será el `www.D.com`, gráficamente puede representarse como:

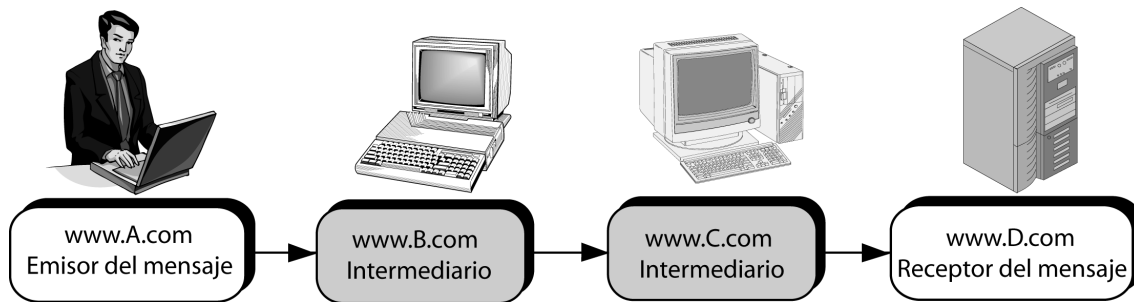


Figura 27: Representación del recorrido del mensaje

El código del mensaje SOAP que se generaría para esta transmisión sería semejante al siguiente listado, en el que se puede observar tanto la ruta de envío, como el destino y emisor del mensaje:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2001/06/soap-envelope">
  <SOAP-ENV:Header>
    <m:path xmlns:wspth="http://schemas.xmlsoap.org/rp/">
      <m:action>http://www.acme.com/procesos</m:action>
      <m:to>soap://www.D.com/fin/trayecto</m:to>
      <m:fwd>
        <m:via>soap://www.B.com</m:via>
        <m:via>soap://www.C.com</m:via>
      </m:fwd>
      <m:from>mailto:alguien@A.com</m:from>
      <m:id>uuid:a47e37d2-32f2-41d1-bd2d-db3708719146</m:id>
    </m:path>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <!-- Resto de elementos de Body-->
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

7.2.2. Los elementos

Puede verse en el ejemplo anterior, que se realiza el rutado mediante la inserción de un elemento `<path>` dentro del elemento `<Header>` del propio `<Envelope>` del documento. Los elementos que se pueden encontrar dentro de esta nueva etiqueta son:

- `action`

Es un elemento obligatorio dentro de los mensajes encaminados. Es generado en el emisor original y no puede ser modificado durante la transmisión del mensaje.

Mediante este elemento se marca el propósito del mensaje, de la misma forma que lo hace `SOAPAction` en una transmisión HTTP SOAP (puede verse en el punto 3.2 del tema sobre el lenguaje SOAP). Su valor es un URI. Existe un URI especial que marca que el mensaje ha sido producido por un error, este valor es `“http://schemas.xmlsoap.org/soap/fault”`, es decir, en todos los mensajes de error que se generen durante la transmisión de un mensaje rutado debe aparecer este URI como valor del elemento `<action>`, aunque el error no haya sido producido directamente por alguna causa relacionada con el encaminamiento.

- `from`

Debe ser generado siempre en el emisor original, y no puede ser cambiado a lo largo de la transmisión. Su valor corresponde a la entidad o humano responsable de la emisión del documento, y está en formato URI. Lo normal es especificar este valor en la forma `“mailto:”`, aunque no es necesario. Este valor no debe ser utilizado nunca como elemento de seguridad ni elemento que valide de mensaje, ya que es fácilmente falsificable. Si se necesitan usar elementos de seguridad o de validación se usarán otros métodos que se verán en el capítulo relacionado con la seguridad.

- `fwd`

Este elemento contiene el camino que deberá seguir el mensaje en su transmisión. Este itinerario se marca mediante elementos `<via>`, que irán posicionados en el mismo orden en el que se irán accediendo.

- `id`

Al igual que en otras ocasiones, este elemento es único y es el identificador. En cada nueva transmisión se debe crear uno distinto y único, por esto se usan identificadores de gran número de bits (128), e incluso se usan técnicas de *hash* como MD5 o DES, para obtener este identificador a partir del contenido del mensaje.

- `relatesTo`

Con él se puede indicar que el mensaje que lo contiene está relacionado con el mensaje que poseía como identificador el valor que ostenta ahora `<relatesTo>`. Dicho de otro modo, el valor que tiene este elemento, es el mismo valor que el del `<id>` del mensaje con el que guarda relación. No se especifica aquí la relación que existe entre los dos mensajes, sino simplemente que están relacionados. No hace falta que los dos mensajes (mensaje origen y mensaje relacionado) sigan las mismas rutas. Sólo puede ser generado por el primer emisor.

- `rev`

Elemento opcional que funciona de modo semejante a como lo hace `<fwd>`, solamente que en este caso, esta ruta se utilizará para el mensaje de respuesta. Sólo puede ser generado por el primer emisor.

- `to`

Sirve para indicar el destinatario final del mensaje. No es obligatorio su uso (aunque si recomendado) en el mensaje de envío; en caso de no existir, el destinatario será el último elemento `<via>` definido dentro de `<fwd>`. En el mensaje de respuesta no debe aparecer este elemento. Su valor viene dado en forma de URI, y lo debe imponer el emisor inicial; de ninguna forma se debe cambiar por parte de los intermediarios.

- `via`

Ya se ha visto que este elemento es a partir del cual se forman las rutas tanto de envío de petición (`<fwd>`), como de respuesta (`<rev>`). Pueden ser insertados por los intermediarios, y son eliminados una vez procesados, por ejemplo, si en `<fwd>` se tiene un primer elemento `<via>` que se dirige hacia `www.acme.com`, una vez que este nodo procesa la petición, lo debe eliminar, de forma que el primer elemento `<via>` ya no será el.

Si el elemento `<via>` no tiene valor alguno, entonces será el protocolo de transporte utilizado quien se encargue de proporcionar el destino. Normalmente se usa sin valor en mensajes de retorno, de respuesta.

El elemento `<via>` posee un atributo llamado `vid`, que sirve para identificar la conexión usada en la petición para poder ser utilizada en la respuesta, de modo que se consigue una comunicación en los dos sentidos por el mismo canal. Hay que tener en cuenta que la conexión TCP usada en el envío, puede haberse cerrado cuando se de la respuesta. Por otra parte, cuando se utiliza TCP se debe proporcionar, tanto a los emisores como a los receptores, unos tiempos de espera tras los cuales deben cerrar la conexión. Estos tiempos son por defecto de dos minutos (RFC 793), por lo que si el tiempo de respuesta es superior a estos dos minutos, es posible obtener un error.

El atributo `vid` se suele emplearse cuando en la ruta de respuesta aparece un `<via>` sin valor, entonces, el nodo intermediario que recibe este mensaje será el encargado de introducirlo. Su uso no es obligatorio, y solamente se introduce en el primer elemento de la ruta de retorno.

A la hora de realizar la ruta, el algoritmo comprueba si existe aún algún elemento `<via>` en la pila `<fwd>` (elementos de ruta de envío), en caso de que exista, se reenvía hacia ese destinatario y en caso de que no exista, se revisa la existencia del elemento `<to>` (destinatario final). En caso de existir `<to>`, éste será el destinatario final, y si no existiera, el destinatario final sería la máquina que estuviera procesando el mensaje en ese momento, ya que sería el último elemento `<via>` dentro de `<fwd>`.

En el primer ejemplo se mostró el mensaje correspondiente al envío entre los nodos del ejemplo A y B, si se analiza, se podrá ver que dentro del elemento `<fwd>` se encuentra la ruta para el envío del mensaje, y por otra parte, en el elemento `<rev>` sólo hay escrito un `<via/>`. Al estar este elemento vacío, la ruta de retorno se irá cumplimentando durante la ruta de emisión. Esto es posible verlo en el mensaje emitido del nodo B al C representado en el siguiente listado:

`<Env:Envelope`

```
xmlns:Env="http://schemas.xmlsoap.org/soap/envelope/">
```

```
<Env:Header>
```

```
<m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
```

```
<m:action>http://www.acme.com/procesos</m:action>
```

```
<m:to>soap://www.D.com/fin/trayecto</m:to>
```

```
<m:fwd>
```

```
<m:via>soap://www.C.com</m:via>
```

```
</m:fwd>
```

```
<m:rev>
```

```
<m:via/>
```

```

        <m:via m:vid="cid:123ABC@www.B.com" />
    </m:rev>

    <m:from>mailto:alguien@A.com</m:from>

    <m:id> uuid:a47e37d2-32f2-41d1-bd2d-db3708719146</m:id>

</m:path>
</Env:Header>

<Env:Body>

    ...

</Env:Body>

</Env:Envelope>

```

En el mensaje correspondiente al envío entre C y D, es posible observar que el elemento `<fwd>` está actualmente vacío, esto es por que no existen más intermediarios. El nodo D, será el destinatario final del mensaje. También puede verse cómo se han ido generando los elementos de la ruta de retorno en cada envío.

```

<Env:Envelope

  xmlns:Env="http://schemas.xmlsoap.org/soap/envelope/">

  <Env:Header>

    <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">

      <m:action>http://www.acme.com/procesos</m:action>

      <m:to>soap://www.D.com/fin/trayecto</m:to>

      <m:fwd>

      </m:fwd>

      <m:rev>

        <m:via>soap://C.com/proc/it</m:via>

        <m:via/>

        <m:via m:vid="cid:123ABC@B.com" />

      </m:rev>

      <m:from>mailto:alguien@A.com</m:from>

      <m:id> uuid:a47e37d2-32f2-41d1-bd2d-db3708719146</m:id>

    </m:path>

```

```

</Env:Header>

<Env:Body>
    ...
</Env:Body>
</Env:Envelope>

```

En los mensajes de respuesta se observa un mecanismo semejante al de la petición, solo que en este caso, la ruta de retorno que se formó durante el primer envío, se utilizará como ruta de nueva envío. Una forma de saber que es un mensaje de respuesta, es porque no existe el elemento *<to>*, además el elemento *<from>* permanece invariable. Puede observarse así mismo, que aparece un elemento *<relatesTo>* para indicar el mensaje con el que tiene relación. El mensaje correspondiente a la comunicación entre los nodos C y B sería el siguiente:

```

<Env:Envelope
  xmlns:Env="http://schemas.xmlsoap.org/soap/envelope/">
  <Env:Header>
    <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
      <m:action>http://www.acme.com/Respuesta</m:action>
      <m:fwd>
        <m:via/>
        <m:via m:vid="cid:123ABC@B.com"/>
      </m:fwd>
      <m:rev>
        <m:via> soap://www.C.com/proc/it2</m:via>
        <m:via>soap://www.D.com/proc/pp2</m:via>
      </m:rev>
      <m:from> mailto:alguien@A.com </m:from>
      <m:id>uuid:9hshs89s-34f5-35ga-rffg-j402kf'25652d53</m:id>
      <m:relatesTo>uuid:a47e37d2-32f2-41d1-bd2d-
db3708719146</m:relatesTo>
    </m:path>
  </Env:Header>
  <Env:Body>

```

...

</Env:Body>

</Env:Envelope>

Por último es posible ver que en el mensaje de respuesta del nodo B hacia el A, el elemento *<fwd>* vuelve a estar vacío, mientras que en el elemento *<rev>* se ha generado una nueva ruta:

<Env:Envelope

xmlns:Env="http://schemas.xmlsoap.org/soap/envelope/">

<Env:Header>

<m:path xmlns:m="http://schemas.xmlsoap.org/rp/">

<m:action>http://www.acme.com/Respuesta</m:action>

<m:fwd>

<m:via/>

</m:fwd>

<m:rev>

<m:via/>

<m:via>soap://www.C.com/proc/it2</m:via>

<m:via>soap://www.D.com/proc/pp2</m:via>

</m:rev>

<m:from>mailto:alguien@A.com</m:from>

<m:id>uuid:9hshs89s-34f5-35ga-rffg-j402kf'25652d53</m:id>

<m:relatesTo>uuid:a47e37d2-32f2-41d1-bd2d-db3708719146</m:relatesTo>

</m:path>

</Env:Header>

<Env:Body>

...

</Env:Body>

</Env:Envelope>

7.2.3. Protocolos

Los protocolos utilizados como medio de transporte en esta especificación, son normalmente TCP, UDP y HTTP. Si el lector posee conocimientos sobre redes, se dará cuenta de algo extraño, y es que el protocolo HTTP no es un protocolo de transporte de red como lo son UDP y TCP, pero en un entorno de mensajes HTTP, puede actuar como tal para transportar el mensaje SOAP. No se va a entrar en estos protocolos por ser complicada la explicación de su uso, pero si remarcar que en la utilización de el protocolo HTTP es recomendable el uso del atributo *mustUnderstand* en la cabecera correspondiente al encaminamiento, y que el atributo *actor* sea "<http://schemas.xmlsoap.org/soap/actor/next>".

Por otra parte, los protocolos TCP y UDP necesitan de un nuevo protocolo para encapsular los datos binarios de forma que sea posible transmitirlos usando WS-Routing. Este protocolo se llama *DIME*, (Direct Internet Message Encapsulation, "Encapsulación" directa de mensajes para Internet) se puede encontrar información al respecto en "http://gotdotnet.com/team/xml_wsspecs/dime/default.htm".

7.2.4 Errores

Al igual que en otras secciones, durante el tránsito del mensaje a lo largo de la ruta, es posible obtener errores de distintas naturalezas, pudiendo ser generados por cualquiera de los intermediarios. En este caso, si existiera la ruta de respuesta, se deberá informar del error al emisor inicial.

Para informar del error producido, toma mucha importancia el elemento `<relatesTo>` que se vio anteriormente en este capítulo, con él se deja patente la relación del mensaje de error con el mensaje original que produjo dicho error.

El error es transportado en la cabecera del mensaje, junto con un mensaje de error en el cuerpo, de manera análoga a lo visto en el capítulo sobre SOAP, sólo que en este caso, los errores en lugar de pertenecer a las clases "*Client*" y "*Sender*", pertenecen a las clases "*Sender*" (emisor) y "*Receiver*" (receptor).

Hay que tener en cuenta que los intermediarios actúan como receptores, y más tarde realizan el papel de emisores, por lo que un nodo puede tener errores de los dos tipos.

El elemento incluido en la cabecera, es el elemento `<fault>` y posee dos subelementos.:

- `code`

Es un número de error que define la especificación WS-Routing. Su valor es normalmente utilizado por programas informáticos.

- `reason`

Es la explicación legible por un humano del número contenido en `<code>` y detalla la causa del error producido.

Además pueden contener otros elementos dependiendo del tipo de error producido, como ejemplo de esta posibilidad, a continuación se muestra mensaje producido por un servicio no disponible, en este caso se añade un nuevo elemento llamado `<retryAfter>`, que indica el tiempo en el que se espera que el servicio vuelva a estar disponible de nuevo:

```
<Env:Envelope
```

```
  xmlns:Env="http://schemas.xmlsoap.org/soap/envelope/">
```

```
  <Env:Header>
```

```
    <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
```

```
      <m:action>http://schemas.xmlsoap.org/soap/fault</m:action>
```

```
      <m:fwd>
```

```
        <m:via>soap://www.C.com/mm/</m:via>
```

```
      <m:via/>
```

```
    </m:fwd>
```

```

    <m:rev>

    </m:rev>

    <m:from>mailto:alguien@A.com</m:from>

    <m:id>uuid:84bgfbhf0-3hfg-4hdf-fghb-5b90nzxd1d6</m:id>

    <m:relatesTo> uuid:84546nw04-23fb-4art-4w2b-
4vs24df431c1d</m:relatesTo>

    <m:fault>

    <m:code>811</m:code>

    <m:reason>Service Unavailable</m:reason>

    <m:retryAfter>1100</m:retryAfter>

    </m:fault>

    </m:path>

</Env:Header>

<Env:Body>

    <Env:Fault>

    <Env:faultcode>Env:Server</Env:faultcode>

    <Env:faultstring>Server Error</Env:faultstring>

    </Env:Fault>

</Env:Body>

</Env:Envelope>

```

Pero si en lugar de este error se obtiene un error por no haber encontrado el destino, el elemento `<fault>` ya no tiene el subelemento `<retryAfter>`, sino que en este caso será el elemento `<endpoint>` el que aparezca:

```

<fault>

    <code>710</code>

    <reason>Endpoint Not Found</reason>

    <endpoint>soap://www.D.com/partidas</endpoint>

</fault>

```

Si al devolver un mensaje de error se produce otro error (sea de la naturaleza que sea), este último se deshecha sin avisar ni dar ningún tipo de reporte.

A continuación se expone una lista con los errores de las clases “*Sender*” y “*Receiver*” que se pueden encontrar cuando se usan mensajes rutados.

Tabla 7.1. Errores de clase “*Sender*”.

Código (code)	Descripción	Elementos Adicionales	Función
700	Cabecera inválida. Todos los valores 7XX no conocidos deben tratarse como un 700.		
710	No se encuentra destino	<endpoint>	Describe el destino no encontrado.
711	El destino se ha eliminado. Se sabe que no volverá a existir		
712	Destino no soportado.		
713	Destino no válido		
720	Si falla el destino, pero se conocen otros destinos que pueden dar el mismo servicio que el solicitado.	<found>	Tiene subelementos <at> con los URI que representan a los destinos encontrados.
730	El URI que representa el destino, es excesivamente largo	<maxsize>	Contiene el tamaño máximo expresado en bytes.
731	El mensaje es excesivamente largo. En muchas ocasiones depende directamente del protocolo de transmisión empleado.		
740	Agotado el tiempo de espera para el intercambio de mensajes	<maxtime>	Tiempo máximo de espera para la respuesta.
750	Se detectaron bucles en el encaminamiento		
751	No es posible usar la ruta indicada como camino de respuesta		

Tabla 7.2. Errores de clase “*Receiver*”

Código (code)	Descripción	Elementos Adicionales	Función
800	Error desconocido, se usa siempre que se obtenga un error 8XX desconocido.		
810	Se encontró un elemento en la cabecera, que no es soportado por el receptor.		
811	Servicio no disponible.	<retryAfter>	Tiempo que hay que esperar (en segundos) hasta volver a hacer otra petición. si no se indica nada serán 5 minutos.
812	Servicio ocupado.	<retryAfter>	Tiempo que hay que esperar (en segundos) hasta volver a hacer otra petición. si no se indica nada serán 5 minutos
820	No se ha podido contactar con el destino.		

Seguridad

8.1. Introducción

A lo largo del libro, el lector ha tenido la oportunidad de contemplar cómo se transmiten las “instrucciones” entre cliente y servicio o entre dos servicios distintos. El empleo de mensajes de tipo texto para realizar estos intercambios de datos, deja al descubierto un problema bastante grande. A nadie se le pasa por alto el hecho de que variar un mensaje de texto es bastante factible, y dependiendo de la naturaleza de dicho mensaje, este problema es aún mayor, ya que si se está tratando por ejemplo una orden de compra, en el mensaje pueden ir datos como números de tarjetas de crédito o claves personales. Queda claro, que en este tipo de transacciones se debe poner todo el cuidado posible para que el mensaje no sea atacado, y en caso de serlo poder detectar este ataque, protegiendo así los datos.

En el capítulo anterior se hizo una introducción al encaminamiento de mensajes, pero en ocasiones este tipo de soluciones no son posibles o no son lo suficientemente robustas para ofrecer los niveles de seguridad deseados.

Actualmente existen varios modelos de seguridad, algunos basados en conexiones codificadas, otros en certificados, etc, y como es de esperar, cada empresa tiene un modelo de seguridad completamente diferente, por ejemplo una puede basarse en un modelo Kerberos, mientras que otra puede ofrecer seguridad mediante SSL. Por ello, los web services deberán de proveer distintos mecanismos de validación y protección de datos. Estos mecanismos son tres:

- Chequeos de integridad de mensaje
- Confidencialidad
- Posibilidad de envío de elementos de seguridad como parte del mensaje.

El uso de estos mecanismos puede realizarse individual o colectivamente, es decir, el uso de uno de ellos no priva de la posibilidad de usar cualquier otro, incluso sobre el mismo dato.

El único objetivo perseguido, es poder realizar comunicaciones de forma segura y flexible, pero si se tiene en cuenta la forma en que la transmisión se lleva a cabo, se puede ver que el abanico de distintos tipos de ataque posibles, es bastante amplio, por lo que se debe dotar de medios para evitarlos. Las soluciones adoptadas:

- Uso de dominios de confianza
- Uso de múltiples tipos de encriptación
- Uso de distintos elementos de seguridad
- Protección punto a punto y no sólo en el transporte
- Uso de firmas digitales

Como se verá más adelante en este mismo capítulo, la mayor parte de la información concerniente a seguridad (excepto formas seguridad no estándar) se almacena en la cabecera, lo cual es lógico, ya que sin leer los datos propiamente dichos del cuerpo del documento (parte `<Body>`), se puede llegar a desechar mensajes que no sean aceptables, ganando tiempo de computación (ya que no hay que hacer un análisis del cuerpo del mensaje) y seguridad.

8.2. Protección del mensaje

Viendo la naturaleza de los documentos SOAP y su modo de transmisión, es posible que el lector haya pensado ya en algunas formas en las que puede ser atacado un mensaje de estas características. Seguramente la mayoría hayan coincidido en dos formas muy evidentes (existen casi ilimitados modos de ataque), la sustitución del emisor y la sustitución (o variación) de los datos del cuerpo..

La sustitución del emisor se daría cuando el mensaje enviado, no ha partido de la máquina que figura como emisora. Suponga que hacemos un pedido de precios a la empresa *acme*, y la empresa *anticame* (la competencia) intercepta el mensaje, y responde al mismo con unos precios falsos, el receptor de este

mensaje falseado debe ser capaz de detectarlo. Este caso es más evidente si se habla directamente de dinero, como en transacciones bancarias.

La sustitución del mensaje podría darse de modo total o parcial. Total sería la generación de un nuevo mensaje (pero manteniendo tanto emisor como receptor) y parcial sería la variación de ciertos datos dentro del propio mensaje.

La forma en la que se protegerá el mensaje será introduciendo firmas digitales que le identifiquen como dueño de éste y lo protejan de variaciones por terceras personas, y elementos de seguridad (ASCII o binarios) para ligar el mensaje con su / sus propietarios de manera unívoca, de tal forma que si fallara alguna de las comprobaciones de los elementos insertados, se podrá rechazar el mensaje. Además este tipo de firmas permiten la imposibilidad de repudio del mensaje por parte del emisor, es decir, una vez enviado el mensaje convenientemente firmado, el receptor sabe que es de un emisor concreto, y éste no puede decir que el documento no fue enviado por él, puesto que su firma está estampada en el mensaje.

En ocasiones es muy importante evitar, no solamente la modificación del mensaje sin ser detectado, sino que también es primordial impedir que los datos contenidos puedan ser leídos por terceras personas. Imagine que está enviando una transacción monetaria, en la que se especifican cuentas bancarias o números secretos, en este caso no interesa sólo el que no modifiquen la información, sino que hay que evitar en lo posible la lectura de la misma. Para ello, se puede codificar tanto partes del mensaje como la totalidad del mismo, pudiendo incluso ser codificado bajo distintos algoritmos, para que cada uno de los actores que intervengan en la transmisión, tenga acceso a una parte restringida del mensaje.

La integridad del mensaje puede controlarse mediante firmas digitales, que analizan la información transmitida y generan cadenas específicas a partir de esta información, funcionando de manera semejante a los algoritmos de redundancia cíclica. Por ejemplo si se envía “1234” se marca como que la suma de guarismos es 10, por lo que si alguien cambia un número, se puede detectar por que la nueva suma ya no será 10. Este ejemplo es muy sencillo, pero es una explicación cercana a lo que se realiza realmente.

Aunque el receptor puede optar por la aceptación de cualquier tipo de mensaje, lo lógico es que se deseché cualquier tipo de documento que esté mal construido, que no pertenezca a los dominios autorizados, que la firma no sea correcta, que esté caducado o que los elementos de reivindicación del mensaje sean inaceptables.

8.3. Ejemplo

Como se puede ver en el siguiente ejemplo, el hecho de usar mensajes SOAP seguros, implica una gran cantidad de información adicional en la cabecera, ya que interesa saber cuanto antes si el mensaje debe o no ser aceptado, y dado que la cabecera se analiza antes que el cuerpo del mensaje, es este el lugar idóneo guardar todo tipo de referencias a elementos seguros.

El ejemplo mostrado, utiliza una autenticación mediante nombre de usuario y “clave”. La clave del usuario en este caso, no se transmite como una palabra en texto plano, sino que se realiza una combinación de la clave, con el elemento `<Create>`, que representa el momento de creación de la cabecera (*Timestamp*) y el elemento `<Nonce>` (elemento efímero normalmente creado de manera aleatoria) que aparecen como hijos del elemento de seguridad `<UsernameToken>`. La conjunción de estos tres elementos, sirve para generar un bloque HMAC (Keyed-Hashing for Message Authentication, fragmentos clave para autenticación de mensajes) (RFC2104) que utilizará para firmar el mensaje. El uso del elemento `<Nonce>` y del elemento `<Create>`, se realiza para conseguir variabilidades en las claves, y que sea más difícil para los atacantes reproducir el mensaje.

El receptor debe tener conocimiento de los datos necesarios para reproducir de nuevo este bloque HMAC, que comparará con el enviado por el emisor (que tiene que ser igual) y poder así comprobar si el mensaje está autorizado o no, desechándolo si no lo estuviera.

Tal como dicta la especificación WSSC (Web Services Security Core Language, núcleo del lenguaje de seguridad de servicios web), para añadir elementos seguros a un mensaje SOAP, se debe incorporar en la cabecera un elemento denominado `<Security>` (con *namespace* <http://schemas.xmlsoap.org/ws/2002/xx/secext>), y dentro de él será donde se definirán todos estos elementos de seguridad.

Uno de los primeros elementos que se pueden ver en el ejemplo siguiente es `<UsernameToken>`. En él se informa el nombre del usuario (elemento `<Username>`), el momento de creación (elemento `<Created>`) y el *nonce* (elemento `<Nonce>`), que se usarán en el servidor para crear el bloque HMAC

junto con la clave. Cabe destacar que en este tipo de verificación, la clave NO se transmite, sino que el servidor la debe conocer, para poder generar el bloque HMAC del mismo modo que lo generó el cliente, y poder así comparar ambos bloques.

El elemento `<Signature>` permite firmar el mensaje para asegurar su integridad. En este caso se usa un XML Signature (firma XML)(con *namespace* `http://www.w3.org/2000/09/xmldsig`). En este bloque se especifica que partes del mensaje están signadas, por ejemplo en este caso, la firma se aplica a todo el cuerpo:

```
<ds:Reference URI="#MsgBody">
  <ds:DigestMethod Algorithm=http://www.w3.org/2000/09/xmldsig#sha1"/>
  <ds:DigestValue>FDJHH6S31...</ds:DigestValue>
</ds:Reference>
```

El elemento `<SignatureValue>` mantiene el valor de la firma de los elementos definidos como seguros, que deben ser verificados, antes de darse como válidos.

El elemento `<SecurityTokenReference>`, define una relación con el identificador que identifica (valga la redundancia) el elemento de seguridad que se utilizará para la verificación de las firmas, ya que puede haber mas de uno. En el ejemplo, el elemento de seguridad que se utiliza para la verificación es `<UsernameToken>`, con identificador *SID*.

El mensaje sería:

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
xmlns:ds=http://www.w3.org/2000/09/xmldsig
xmlns:wsu=http://schemas.xmlsoap.org/ws/2002/xx/utility>
<S:Header>
  <wsse:Security
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext">
    <wsse:UsernameToken wsu:Id="SID">
      <wsse:Username>Manuel</wsse:Username>
      <wsse:Nonce>ASLDKAR</wsse:Nonce>
      <wsu:Created>2002-12-11T04:03:23</wsu:Created>
    </wsse:UsernameToken>
    <ds:Signature>
      <ds:SignedInfo>
        <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
```

```

    <ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
        <ds:Reference URI="#MsgBody">
<ds:DigestMethod
    Algorithm= "http://www.w3.org/2000/09/xmldsig#sha1"/>
        <ds:DigestValue>FDJHH6S31</ds:DigestValue>
    </ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>CKLN7W4EJHF8764</ds:SignatureValue>
<ds:KeyInfo>
    <wsse:SecurityTokenReference>
        <wsse:Reference URI="#SID"/>
    </wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</S:Header>
<S:Body wsu:Id="MsgBody">
    <dde:Pedido xmlns:dde="http://acme.com/pedidos">
<!--resto del mensaje -->
    </dde:Pedido>
</S:Body>
</S:Envelope>

```

8.4. Cabecera de seguridad

En el punto anterior se pudo observar un mensaje en el que la información acerca de la seguridad, se almacenaba en la cabecera, esto es por que, a parte de las razones expuestas anteriormente, los elementos siguientes pueden depender de la información que haya escrita en la cabecera. Si por ejemplo se dicta un tipo de encriptación para un elemento concreto, antes de procesar este elemento, se deberá tener conocimiento del algoritmo de encriptación, ya que de otra forma no se podría acceder a su información de manera correcta.

Dentro de la cabecera se encuentra el bloque `<Security>` que será el elemento encargado de albergar los datos referentes a la seguridad.

El elemento `<Security>` posee un atributo llamado *role* (papel desempeñado), que identifica el destinatario de dicho bloque. El atributo es opcional, pero sólo puede existir un bloque sin él, es decir, una vez que se ha añadido un elemento `<Security>` sin este atributo, se hace obligatorio su uso en los restantes. Además no podrá haber dos elementos cuyo valor del atributo *role* coincida, ya que sería indicar que un mismo destinatario tendría dos políticas de seguridad distintas, lo cual no es lógico.

Como ya se ha dicho, el atributo *role*, identifica al destinatario del bloque que lo define. Destinatarios diferentes deben tener roles diferentes, y por lo tanto, deben tener bloques distintos. Si se omite este atributo, entonces es una política genérica, y debe ser tomada en cuenta por todos.

Normalmente cuando un *role* procesa su bloque, este suele ser retirado de la cabecera, ya que no será utilizado de nuevo. El bloque `<Security>` que no tiene definido el atributo *role*, tiene que estar presente hasta el destinatario final, dicho de otro modo, no puede ser eliminado del mensaje por ninguno de los intermediarios.

Por otra parte, a lo largo del trayecto entre el emisor y el destinatario final del mensaje, pueden añadirse nuevas políticas de seguridad; éstas deben colocarse precediendo a las ya existentes. Esta forma de actuar asegura que los subelementos siguientes pueden ser procesados, ya que su política de seguridad ya ha sido analizada, y se evita así tener referencias a elementos futuros, o lo que es lo mismo, elementos aún no procesados.

Las implementaciones que soportan los elementos `<Security>`, deben ser capaces de procesar todos los elementos y subelementos contenidos en la cabecera de seguridad, de modo contrario, el mensaje no debe ser admitido, y en este caso se devolvería al emisor un mensaje de error.

El esquema de la cabecera de seguridad dentro de un mensaje SOAP sería:

```
<?xml version="1.0" encoding="utf-8"?>

<Env:Envelope xmlns:Env="http://www.w3.org/2001/12/soap-envelope">

  <Env:Header>

    <!--Otros elementos-->

    <wsse:Security

      xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext"

      Env:role="http://www.acme.com" Env:mustUnderstand="true">

      <!--Otros elementos-->

    </wsse:Security>

    <!--Otros elementos-->

  </Env:Header>

<Env:Body>

<!--Otros elementos-->

</Env:Body>

</Env:Envelope>
```

En las secciones siguientes se hará una exposición de los elementos que pueden encontrarse dentro del elemento `<Security>`.

8.4.1. Elementos de seguridad

Los elementos de seguridad que se pueden encontrar en una cabecera de seguridad se dividen en tres categorías:

- elementos de nombre usuario
- elementos XML
- elementos binarios

8.4.1.1. Elementos de nombre de usuario

Mediante el elemento `<UsernameToken>` se puede dar información acerca del usuario emisor del mensaje, así como de otros datos como la clave.

Si la clave está presente, puede representarse de dos formas distintas *PasswordText* o como *PasswordDigest*. El atributo *Type* del elemento `<Password>` debe indicar mediante una de las palabras anteriores, la manera que se eligió para su envío. La diferencia entre estas dos métodos de remisión, radica en la forma que se tiene de codificar la clave, ya que en el primer caso (*PasswordText*), la clave se transmite tal cual, como texto plano, sin realizar ninguna operación sobre ella, mientras que en el segundo caso se realiza un algoritmo y una codificación base64.

El método *PasswordDigest* crea una versión codificada bajo UTF-8 de la clave a enviar a la cual se le aplica un algoritmo SHA1 (RFC3174) (Secure Hash Algorithm 1, algoritmo seguro de fragmentos) y finalmente el resultado se codifica bajo base64, que es lo que realmente se envía. Además, en este método se suelen añadir dos elementos que proporcionan mayor seguridad a esta forma de transmisión, que son los elementos `<Nonce>` y `<Created>` vistos anteriormente en el primer ejemplo. `<Nonce>` es un elemento que suele ser generado de forma aleatoria, una cadena que se crea sin ningún patrón preestablecido (en ocasiones se calcula a partir de la cadencia de tecleo, u otros mecanismos aleatorios), y el elemento `<Created>` que tiene como valor el momento de creación del mensaje (*timestamp*). Estos dos nuevos elementos entrarán a formar parte de la cadena a la que se le aplicará el algoritmo SHA1, es decir, se crea una cadena con la clave, el elemento `<nonce>` y el elemento `<created>`, y es a esta cadena a la que se le aplica el algoritmo SHA1. Lo que se consigue de esta forma es que cada vez que se realice esta acción (la de aplicar SHA1), se obtendrán resultados distintos, ya que no todos los elementos que forman la cadena son iguales. Para que el servidor sea capaz de interpretar si es un mensaje válido o no, se tienen que dar dos condiciones adicionales.

1. Si se envía una clave con el tipo *PasswordDigest* que se haya generado utilizando los elementos `<Nonce>` y `<Created>`, estos también se deben enviar, para que el servidor los pueda utilizar en su algoritmo SHA1 y así poder comparar.
2. El servidor tiene que conocer la clave del usuario, ya que la necesita para generar el bloque de comparación.

El esquema del elemento `<UsernameToken>` es el siguiente:

```
<wsse:UsernameToken wsu:Id="...">
  <wsse:Username>...</wsse:Username>
  <wsse:Password Type="...">...</wsse:Password>
  <wsse:Nonce EncodingType="...">...</wsse:Nonce>
  <wsu:Created>...</wsu:Created>
</wsse:UsernameToken>
```

Por defecto, el valor del atributo *Type* del elemento `<Password>` es *PasswordText* y en cuanto al atributo *EncodingType* (tipo de codificación) del elemento `<Nonce>`, su valor por defecto es *Base64*.

Además se pueden extender los elementos, siempre y cuando se informe en la cabecera de la localización de los documentos explicativos de estas extensiones.

Un ejemplo de uso de una clave de tipo *PasswordText* se puede ver en el siguiente fragmento:

```
<wsse:Security>

  <wsse:UsernameToken

    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext"

    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/xx/utility"

    wsu:Id="SID">

      <wsse:Username>Manuel</wsse:Username>

      <wsse:Password Type="wsse:PasswordText">

        vacacionesYA

      </wsse:Password>

    </wsse:UsernameToken>
  </wsse:Security>
```

y un ejemplo de uso del método *PasswordDigest* sería:

```
<wsse:Security>

  <wsse:UsernameToken

    xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/xx/utility"

    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext"

    wsu:Id="SID">

      <wsse:Username>Manuel</wsse:Username>

      <wsse:Password Type="wsse:PasswordDigest">

        KD98353frD8GF4SJ

      </wsse:Password>

      <wsse:Nonce>A234FB9AC32A</wsse:Nonce>

      <wsu:Created>2002-12-11T04:03:23</wsu:Created>

    </wsse:UsernameToken>
  </wsse:Security>
```

8.4.1.2. Elementos XML

En esta guía no se verá el uso de este tipo de elementos, pero si el lector está interesado en leer acerca de ellos, puede encontrar información al respecto en la web de OASIS (<http://www.oasis-open.org/>).

8.4.1.3. Elementos binarios

Para el envío de elementos binarios de seguridad, la especificación proporciona una etiqueta propia para incorporar dentro del elemento `<Security>`. Esta etiqueta es `<BinarySecurityToken>`. El uso de estos elementos es debido a que algunos sistemas de seguridad emplean bloques de datos binarios como parte de su autenticación (por ejemplo certificados X.509).

El esquema del elemento es:

```
<wsse:BinarySecurityToken wsu:Id="..."

    EncodingType="..."

    ValueType="...">

    ...

</wsse:BinarySecurityToken>
```

Al igual que en ocasiones anteriores, el atributo `Id` del elemento, sirve para identificarlo de manera unívoca, el atributo `EncodingType` sirve para especificar el tipo de codificación utilizada en el dato, que normalmente será `Base64Binary` aunque se pueden definir sistemas propios de codificación, tal y como se vio en el capítulo referente al lenguaje SOAP. Por último, el elemento `ValueType` tiene información sobre el tipo de dato que lleva (por ejemplo si es un pase Kerberos), y pueden indicarse tipos nuevos creados por necesidades específicas del usuario, pero en este caso, se le debe dar un nombre cualificado mediante un *namespace*. En el ejemplo siguiente se puede ver este hecho, donde `ValueType` toma un valor definido por el usuario.

```
<wsse:BinarySecurityToken

    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext"

    wsu:Id="BSID"

    ValueType="ntn:NuevoTipo" xmlns:ntn="http://www.acme.com/tipos"

    EncodingType="wsse:Base64Binary">

    DFSgfLKNdf97958u3LFBRDFAiSAREte884234523

</wsse:BinarySecurityToken>
```

8.4.2. Firmas digitales

En ocasiones interesa que el receptor del mensaje sea capaz de determinar si el emisor del mismo es realmente quien lo firma y que su identidad no ha sido suplantada, o ver si en alguno de los pasos de la comunicación entre el emisor y el receptor, alguien ha alterado de alguna forma el contenido del mensaje, bien borrando elementos o modificando de algún modo la información. Estas tareas se pueden desempeñar mediante firmas digitales XML (RFC3075).

Mediante las firmas se pueden autenticar cabeceras y / o cuerpos de mensajes, incluso partes de ellos. La manera de hacerlo es colocar una referencia a un elemento de seguridad (por ejemplo un

certificado Kerberos) dentro de la firma que asegura el contenido de la parte del mensaje. En cada firma se incluye un elemento llamado `<KeyInfo>` que contendrá un subelemento `<SecurityTokenReference>` con la referencia al elemento de seguridad. Aunque parezca un poco enrevesado, cuando se vea el ejemplo se podrá comprender perfectamente.

Además, mediante las distintas referencias, se podrán aplicar firmas a elementos ya firmados, incluso mediante elementos seguros distintos.

Cuando se va a realizar la firma de cualquier elemento del mensaje, se introduce en la cabecera de seguridad el elemento `<Signature>`, y éste se debe anteponer a todos los existentes previamente; esto se hace para asegurar que si se signa un elemento anteriormente firmado, se pueda verificar la segunda firma antes de proceder a la verificación de la primera, en otras palabras, la verificación de firmas debe realizarse en orden inverso a la realización de las mismas.

Cuando se firman mensajes, hay que tener cuidado con las partes sobre las que se realiza esta acción y no bloquear elementos del mensaje que realmente deben ser modificados durante la transmisión. Es preferible crear varias firmas pequeñas que no una grande. Siempre sin descuidar la seguridad.

Todas las referencias incluidas (elementos `<Reference>`) dentro del elemento `<Signature>` tienen que estar contenidas en el elemento `<Envelope>` o bien en un añadido si se usan transmisiones con *attachment* (algunos sistemas de autenticación guardan su elemento de seguridad en un fichero y en este caso sería lógico usar un añadido firmado).

A la hora de crear las referencias, es posible el uso del estándar XPATH para indicar las partes del documento XML, aunque debido a que es posible modificar bloques ya firmados, en ocasiones el resultado obtenido en el cliente no es el mismo que el obtenido en el servidor, por lo que debe usarse con cuidado (la recomendación sobre XPATH puede ser consultada en <http://www.w3.org/TR/1999/REC-xpath-19991116>). A continuación se presenta un ejemplo del uso de XPATH en la referencia:

```
<Reference URI=" ">
  <Transforms>
    <Transform
      Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
      <XPath xmlns:dsig="&dsig;">
        count(ancestor-or-self::dsig:Signature |
             here()/ancestor::dsig:Signature[1]) >
        count(ancestor-or-self::dsig:Signature)
      </XPath>
    </Transform>
  </Transforms>
  <DigestMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue></DigestValue>
</Reference>
```


A la hora de validar el mensaje, se debe analizar cada una de sus partes, tomándolo como mensaje erróneo si alguna de las firmas no supera la verificación. Las causas por las que un elemento *<Signature>* puede darse como erróneo, se dividen en dos tipos:

- errores de XML-SOAP
- errores de validación

Dentro de los errores XML-SOAP se engloban todos aquellos relacionados con la formación incorrecta del mensaje en sí, desde formaciones erróneas de sintaxis hasta falta de referencias. El otro tipo de error está más ligado con la idea de seguridad, es decir, una vez analizada la firma, el programa puede determinar que no pertenece a un usuario validado, que ha sido modificada la información o cualquier otro tipo de error en este campo.

Tanto si se produce un error como el otro, debe notificarse al emisor, para que en lo posible ponga remedio a la situación. La forma en que se transmite esta información acerca del fallo es idéntica a que se vio en capítulo sobre el lenguaje SOAP.

A continuación se muestran los distintos códigos de error que pueden ser devueltos por causas relacionadas con la seguridad:

Tabla 8.1 Errores relacionados con la seguridad

Código de fallo (faultcode)	Descripción del fallo
FailedAuthentication	No se puede autenticar el elemento de seguridad, o bien el resultado ha sido negativo.
FailedCheck	La firma no es válida.
InvalidSecurity	Se produjo un error al verificar la cabecera segura (elemento <i><Security></i>).
InvalidSecurityToken	El elemento de seguridad no es válido.
SecurityTokenUnavailable	El destino de la referencia del elemento de seguridad no se encuentra.
UnsupportedAlgorithm	El algoritmo de creación de firmas no está soportado por el sistema.
UnsupportedSecurityToken	El elemento transmitido no es soportado por el sistema.

En el siguiente ejemplo se realiza la firma de un elemento del cuerpo del mensaje:

```
<?xml version="1.0" encoding="utf-8"?>
<Env:Envelope
  xmlns:Env="http://www.w3.org/2001/12/soap-envelope"
  xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
  xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <Env:Header>
    <wsse:Security>
      <wsse:BinarySecurityToken
        ValueType="wsse:X509v3"
        EncodingType="wsse:Base64Binary"
        wsu:Id="X509Cer">
        KpsoIddhfy54hth6TBEDED566r
```

```

</wsse:BinarySecurityToken>
<ds:Signature>
  <ds:SignedInfo>
    <ds:CanonicalizationMethod Algorithm=
      "http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
      <ds:SignatureMethod Algorithm=
        "http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
      <ds:Reference URI="#ac:pNom">
        <ds:Transforms>
          <ds:Transform Algorithm=
            "http://www.w3.org/TR/2000/CR-xml-c14n-20001026">
            </ds:Transforms>
          <ds:DigestMethod Algorithm=
            "http://www.w3.org/2000/09/xmldsig#sha1"/>
          <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=
            </ds:DigestValue>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>
        vmMQFd%gdJhf56-vNN7z4
      </ds:SignatureValue>
      <ds:KeyInfo>
        <wsse:SecurityTokenReference>
          <wsse:Reference URI="#X509Cer"/>
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
    </ds:Signature>
  </wsse:Security>

```

```
</Env:Header>

<Env:Body>

    <!-- otros elementos -->

    <ac:PagoNomina xmlns:ac="http://www.acme.com/procesos"

        wsu:Id="ac:pNom">

        <!-- otros elementos -->

    </ac:PagoNomina>

    <!-- otros elementos -->

</Env:Body>

</Env:Envelope>
```

8.4.3. Cifrado

La especificación actual de seguridad en los servicios web, permite la encriptación de bloques de la cabecera, del cuerpo, cualquier subelemento de ellos e incluso de los elementos que lleven anexos (*attachments*). Aunque se permitan codificar bloques tanto de la cabecera como del cuerpo, no es posible codificar la cabecera ni el cuerpo directamente.

Existen dos tipos de codificación, los conocidos como clave simétrica y los de clave asimétrica. La diferencia entre ellos es que en la clave simétrica solamente existe una clave, que se usará tanto para cifrar como para descifrar, y en los de clave asimétrica existen dos claves, una para cifrar, y otra distinta para descifrar. Aunque pueda parecer extraño que se usen dos claves distintas para cifrado y descifrado, es un método bastante utilizado; algoritmos como DSA usan este tipo de claves. Se basan principalmente en propiedades matemáticas como la descomposición en factores primos de números grandes (algoritmo RSA). En la figura 1 se puede ver un esquema del funcionamiento de las claves simétricas y asimétricas para el cifrado.

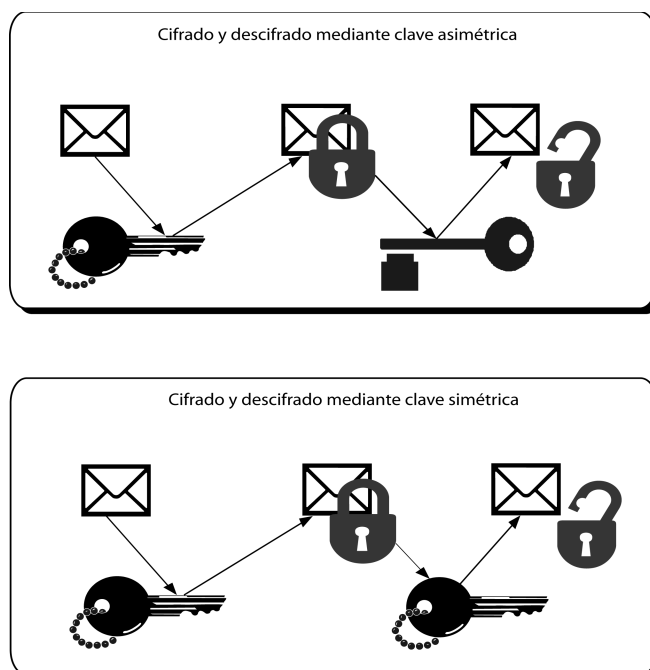


Figura 28: Diferencia entre claves simétricas y asimétricas la encriptación.

Los métodos de cifrado usados para los datos en mensajes SOAP son de clave simétrica como 3DES o AES, aunque podemos encontrar en el XML Encryption Core (núcleo de encriptación XML), que es el estándar en el que se basa la encriptación en los servicios web, que se pueden usar claves asimétricas como RSA para codificar a su vez las claves simétricas en su transporte. Un uso muy normal, es la generación de una clave simétrica de forma aleatoria, que se codifica mediante una clave pública asimétrica y se envía como parte del mensaje, para que el receptor utilice la clave privada correspondiente a la clave pública utilizada, para obtener así la decodificación del mensaje. El uso de claves asimétricas para la codificación de datos no es utilizado debido a su baja eficiencia en comparación con las claves simétricas..

Cuando el emisor o alguno de los intermediarios realiza una acción de encriptación de alguna de las partes del mensaje, se debe de anteponer un elemento `<ReferenceList>` en la cabecera de seguridad correspondiente (`<Security>`), en el que se informará claramente de las partes que se han cifrado, para que el receptor pueda saber cuales de ellas tiene que descodificar.

Cuando nos referimos a un cifrado, siempre nos referiremos al acto de cifrar, e insertar el elemento correspondiente en la cabecera, ya que son acciones totalmente dependientes. En el elemento `<ReferenceList>` se mantienen las referencias a las partes cifradas para poder conocer exactamente que zonas se han de tratar cuando se descifre.

Cuando algún elemento del mensaje se codifica, se almacena su referencia a modo de índice en el elemento `<ReferenceList>`, siendo remplazado por un nuevo elemento `<EncryptedData>`. Todos estos bloques `<EncryptedData>` sin excepción, deben tener una referencia `<DataReference>` dentro del elemento de reseñas `<ReferenceList>`.

En el siguiente ejemplo se incluye un elemento `<KeyInfo>` con la información de la clave usada para la codificación, dentro de cada unidad `<EncryptedData>`, se puede ver que dentro del elemento `<ReferenceList>` se ha añadido una referencia hacia la parte de l mensaje que ha sido cifrada:

```
<Env:Envelope
```

```
  xmlns:Env="http://www.w3.org/2001/12/soap-envelope"
```

```
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
```

```

    xmlns:wss="http://schemas.xmlsoap.org/ws/2002/xx/secext "
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
<Env:Header>
    <wsse:Security>
        <xenc:ReferenceList>
            <xenc:DataReference URI="#bID" />
            <!--aqui se pueden añadir nuevas referencias -->
        </xenc:ReferenceList>
    </wsse:Security>
</S:Header>
<S:Body>
    <xenc:EncryptedData Id="bID">
        <ds:KeyInfo>
            <ds:KeyName>CN=Prueba, C=ES</ds:KeyName>
        </ds:KeyInfo>
        <xenc:CipherData>
            <xenc:CipherValue>
                <!-- Valores -->
            </xenc:CipherValue>
        </xenc:CipherData>
    </xenc:EncryptedData>
</Env:Body>
</S:Envelope>

```

8.4.3.1. Datos cifrados

En ocasiones es posible que alguno de los datos enviados dentro del mensaje SOAP, contenga información sensible, por lo que se debe ocultar su contenido. En este caso, el elemento *<EncryptedData>* entra en juego, proporcionando estos servicios de codificación.

Cuando se codifican datos anexos al mensaje (*attachments*), hay que tener en cuenta varios aspectos:

- El contenido del anexo se sustituye por la cadena codificada.

- El tipo original del anexo se guarda en un atributo del elemento `<EncryptedData>` llamado `MimeType`.
- El tipo del anexo pasa a ser `application/octet-stream`.
- Se introduce un elemento `<EncryptedData>` por cada uno de los anexos codificados.
- El elemento MIME codificado, se referencia desde un elemento `<CipherData>` dentro del `<KeyInfo>` del `<EncryptedData>`.

A continuación se muestra un ejemplo con dos ficheros codificados, un documento de texto PDF, y un archivo gráfico SVG:

```
<Env:Envelope
xmlns:Env="http://www.w3.org/2001/12/soap-envelope"
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/xx/secext"
xmlns:xenc=http://www.w3.org/2001/04/xmlenc#
xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<Env:Header>
<wsse:Security>
<xenc:EncryptedData MimeType="image/svg+xml">
  <ds:KeyInfo>
    <wsse:SecurityTokenReference>
      <xenc:EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#3des-cbc"/>
      <wsse:KeyIdentifier EncodingType="wsse:Base64Binary"
        ValueType="mt:nTipo" xmlns:mt=http://www.acme.com/tipos>
        f$MsaYvs4GTWh8
      </wsse:KeyIdentifier>
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>
  <xenc:CipherData>
    <xenc:CipherReference URI="#cid:imagen"/>
  </xenc:CipherData>
</xenc:EncryptedData>
<!-- segundo elemento -->
```

```

<xenc:EncryptedData MimeType = "application/pdf">
  <ds:KeyInfo>
    <wsse:SecurityTokenReference>
      <xenc:EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#3des-cbc"/>
      <wsse:KeyIdentifier EncodingType="wsse:Base64Binary"
        ValueType="wsse:X509v3">
MOpad25.ER...
      </wsse:KeyIdentifier>
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>
  <xenc:CipherData>
    <xenc:CipherReference URI="#cid:texto"/>
  </xenc:CipherData>
</xenc:EncryptedData>
</wsse:Security>
</Env:Header>
<Env:Body> <!-- Otros datos --> </Env:Body>
</Env:Envelope>

```

El proceso de codificación de un mensaje SOAP, tiene que dar como resultado SIEMPRE otro mensaje SOAP válido, es decir, no se puede codificar directamente los elementos `<Envelope>`, `<Header>` o `<Body>` pero si cualquiera de sus elementos componentes.

8.4.3.2. Claves cifradas

Si en el proceso de cifrado se utiliza una clave simétrica, que será enviada como parte del propio mensaje codificado, entonces se debe incluir el elemento `<EncryptedKey>` con la codificación de dicha clave simétrica.

El elemento `<EncryptedKey>` debe mantener una lista con las referencias de los elementos que se han codificado usando esta clave, esto se hace utilizando elementos `<ReferenceList>`, que ya se han visto anteriormente. Además los elementos que se codifican con esta clave, son sustituidos por el elemento `<EncryptedData>`, y son referenciados como hijos del elemento `<ReferenceList>`.

La especificación sobre seguridad en web services aconseja que los elementos `<EncryptedKey>` se definan como hijos directos de la cabecera de seguridad, es decir, como hijos del elemento `<Security>`, mientras que la especificación XML Encryption, obliga a que esté dentro de los elementos

`<EncryptedData>`, por lo que si se tiene que cumplir las dos especificaciones, se debe seguir la de XML Encryption, ya que es más restrictiva que la de web services.

8.5. Otras consideraciones

En ocasiones, en temas de seguridad es muy importante el tiempo en el que se ha realizado el envío del mensaje, por ejemplo, puede que se den permisos a un usuario en unas horas determinadas o un tiempo limitado, por lo que interesa saber cuando se ha creado el mensaje o los elementos seguros, porque pueden ser válidos pero estar caducados, por lo que no sería un mensaje aceptable.

Para controlar el tiempo de creación, el WSSC (Web Services Security Core Language, núcleo del lenguaje de seguridad de servicios web), proporciona el elemento `<Timestamp>`, que permite definir dos nuevos elementos, que son:

- `<Created>` para indicar el momento de creación de la cabecera, más concretamente se usa el momento en el que se produce la serialización del mensaje, su carácter es opcional, y solamente puede crearse uno dentro de cada elemento `<Timestamp>`.
- `<Expires>` para indicar el momento a partir del cual el mensaje no debe ser válido. El receptor que obtenga un mensaje, cuyo tiempo de expiración se haya superado, debe rechazarlo, pudiendo o no informar al emisor a través de un mensaje de fallo (*Fault*). El elemento es de carácter opcional y al igual que `<Created>`, sólo puede haber, a lo sumo, uno.

Los elementos `<Created>` y `<Expires>` están creados para su uso como hijos de `<Timestamp>`, pero pueden ser utilizados en cualquier parte del mensaje donde puedan ser necesarios o útiles, tanto en la cabecera como en el cuerpo.

Un ejemplo de una cabecera con caducidad, lo puede verse en este listado:

```
<Env:Header>
```

```

  <wsu:Timestamp>
    <wsu:Created>2002-05-04T09:12:10Z</wsu:Created>
    <wsu:Expires>2002-05-20T22:00:00-06:00</wsu:Expires>
  </wsu:Timestamp>

```

```
<Env:Header>
```

La representación de fechas se realiza según el XML Schema, y se puede encontrar información sobre la sintaxis de los tipos tiempo en la norma ISO 8601. La recomendación es usar en la medida de lo posible tiempos universales, es decir, medidos desde la coordenada cero de tiempos (meridiano con longitud 0°), es lo que se conoce como UTC (*Coordinated Universal Time*, Tiempo Universal Coordinado), aunque nada impide utilizar tiempos desplazados como en el elemento `<Expires>` del ejemplo anterior.

Si en algún momento interesa conocer los retrasos producidos entre los distintos pasos de la comunicación del mensaje SOAP, es posible utilizar el elemento `<TimestampTrace>`, para ir almacenando un registro de ellos.

El elemento `<TimestampTrace>` está compuesto de elementos `<Received>` con la información respecto a los retrasos, tiempo de recepción, y rol que lo introdujo.

Un ejemplo del uso de este elemento se muestra en el siguiente fragmento de mensaje:

```
<S:Header>
```



```

<wsu:Timestamp>
  <wsu:Created>2002-05-04T09:12:10Z</wsu:Created>
  <wsu:Expires>2002-05-20T22:00:00-06:00</wsu:Expires>
</wsu:Timestamp>
<wsu:TimespampTrace>
  <wsu:Received Role="http://www.acme.com/"
    Delay="10000" wsu:Id="TTRacme">
    2002-05-04T09:18:10Z
  </wsu:Received>
</wsu:TimespampTrace>
<!--Otros elementos -->
</S:Header>

```

Los atributos utilizados son:

- **Delay:** Representa el tiempo en milisegundos que ha pasado desde que el receptor toma el mensaje hasta que lo vuelve a transmitir; es el tiempo de proceso de este receptor. Es un atributo optativo.
- **Id:** Es el identificador XML definido por el XML Schema. Servirá para referirse a este elemento desde cualquier otra parte del mensaje. Su carácter es opcional.
- **Role:** Atributo de carácter obligatorio, que sirve para conocer el receptor.

En este caso, como en el anterior, los tiempos es recomendable que sean UTC (Ver nota anterior).

Cooperación entre Servicios Web

9.1. Introducción

Se ha visto que los servicios web además de ser accedidos por programas, pueden comunicarse entre ellos, incluso buscar y descubrir el servicio al que tienen que llamar de forma automática. El problema es que esto puede resultar complicado.

Para cubrir estas carencias surgen nuevas especificaciones. BEA, IBM y Microsoft publicaron la especificación para transacciones de negocio y automatización de procesos, donde se explica la forma de definir, crear y conectar procesos mediante servicios web. Ésta especificación ayuda a la conexión de procesos entre aplicaciones internas y de otro tipo, como pueden ser las aplicaciones de los clientes o de los proveedores.

Esta nueva especificación se basa a su vez en otras tres: WS-Coordination (coordinación entre servicios web), WS-Transaction (transacciones en servicios web) y BPEL4WS.

Un ejemplo típico del uso de un proceso que involucra varios servicios web de distintos propietarios es el de la reserva de un crucero. Cuando se reserva el crucero, también se reservan otras cosas, como pueden ser hoteles, coches de alquiler, excursiones, etc. Todo esto se puede hacer mediante servicios web coordinados. Por ejemplo, al enviar la petición, el web service principal (coordinador) detecta que se quiere reservar, para unas fechas, un hotel en una ciudad determinada, entonces contacta con un servicio web que se encargue de ello y realiza la reserva; por otro lado se tiene la reserva del coche, para la cual el sistema actuará de forma similar e incluso si por cualquier causa una de las reservas no se ha podido realizar, se pueden deshacer todos los procesos (o alguno) realizados anteriormente. Así se tiene que mediante una sola petición a un servicio web, el sistema se ha encargado de todo, repartiendo mensajes y tareas a otros servicios, basándose en las especificaciones con las que fue concebido.

De forma abreviada, se puede decir que WS-Coordination y WS-Transaction proveen de mecanismos para trabajar con distintos servicios web que interactúen, sin importar infraestructuras y el estándar BPEL4WS permite describir los procesos que se dan lugar entre distintos web services, y así estandarizar el intercambio de mensajes.

Visto desde otro punto, primero se definen los procesos que se deben realizar, esto se hace mediante BPEL4WS. Una vez creados, se coordinan mediante WS-Coordination y WS-Transaction.

A continuación se tratará de explicar un poco más la utilidad y funcionamiento de cada uno de ellos sin entrar en detalles, puesto que especificaciones muy extensas.

9.2 WS-Coordination

Bajo este estándar se describe un entorno en el que ofrecen unos protocolos para el control de las aplicaciones. Estos protocolos pueden coordinar de forma segura las transacciones entre distintas aplicaciones distribuidas.

Esta especificación se ha de tomar como un bloque de construcción que ofrece ciertos servicios, pero normalmente no es por si sola capaz de satisfacer las necesidades de los clientes, es por lo que se suele utilizar en conjunción con otras especificaciones (otros bloques) como WS-Security o WS-Transaction para conseguir un entorno que se ajuste a las necesidades.

El entorno creado por esta especificación, permite la creación por parte del servicio de un contexto sobre el cual propagar sus procesos hacia otros servicios y registrar los protocolos de coordinación. El entorno permite a los flujos de trabajo y a otros tipos de sistemas para coordinar integrarse mediante la ocultación de sus protocolos no estándares de coordinación (protocolos propios) y trabajar con los estándar para inter operar en un entorno heterogéneo.

Los mensajes que trabajan en este tipo de entornos, mantienen la información sobre él en la cabecera mediante un elemento llamado *<CoordinationContext>*.

Información sobre el XML Schema y WSDL de la última versión de este estándar puede obtenerse en:

<http://schemas.xmlsoap.org/ws/2002/08/wscoor/wscoor.xsd>

<http://schemas.xmlsoap.org/ws/2002/08/wscoor/wscoor.wsdl>

9.3 *WS-Transaction*

Esta especificación define los tipos de coordinación que son utilizados en el entorno concretado en el punto anterior.

En la especificación se definen dos tipos de transacciones:

1. Transacciones atómicas (denominadas AT, Atomic Transaction)
2. Actividades empresariales (denominadas BA, Business Activity)

Como programador se tiene la opción de elegir cualquiera de ellas (o las dos) a la hora de diseñar las aplicaciones y al igual que WS-Coordination, WS-Transaction sirve como bloque de construcción de aplicaciones, que se puede combinar con otros, por ejemplo con el propio WS-Coordination.

Puede encontrarse la especificación completa en la dirección:

<http://dev2dev.bea.com/techtrack/ws-transaction.jsp>

9.3.1 Transacciones atómicas

Una transacción atómica es usada para coordinar actividades de corta duración y que se ejecutan en dominios de confianza. En su especificación se definen mecanismos para crear enlaces entre sistemas transaccionales existentes y estos nuevos mecanismos, permitiendo así su interoperabilidad.

Se las denomina atómicas, porque los procesos que se dan en ellas se deben cumplir sin problemas. Si en alguno de ellos surgiera algún problema, de cualquier tipo, se procedería a deshacer los procesos que estando dentro de esta transacción, se hubieran ejecutado correctamente. Al terminar los procesos, el coordinador pregunta por el resultado de cada uno de ellos. Si alguno de los procesos indicados le responde que no ha tenido un final satisfactorio, o bien no le responde, el coordinador da la orden de deshacer todas las acciones que se habían realizado durante la transacción atómica. Si todos los procesos responden que sus acciones han finalizado correctamente, procede a la aceptación de la transacción y a la aplicación de las acciones realizadas.

Las transacciones atómicas bloquean recursos tanto a nivel de datos (registros de bases de datos, ficheros) como físico (memoria) para realizar sus procesos. Así, los procesos se aseguran poder deshacer la acción en caso de recibir esta orden por parte del coordinador.

Todas estas acciones se deben realizar de manera transparente al resto de aplicaciones o de servicios que usen los sistemas que estén involucrados en la transacción.

9.3.2 Actividades empresariales

Las actividades empresariales se emplean para coordinar procesos de larga duración y en las que se desea aplicar una lógica de negocio a los fallos, es decir mecanismos para tratarlos. A diferencia de las transacciones atómicas, las actividades empresariales no bloquean datos mientras realizan sus procesos (pueden ser largos), y además, sus consecuencias son irremediables; una vez realizado el proceso, no hay manera de deshacerlo (no hay forma de deshacerlo de modo automático, está claro que en ocasiones es posible realizar la operación inversa).

Las actividades empresariales comparten los recursos, permitiendo el acceso a estos por cualquier otra aplicación en cualquier momento, no como en las transacciones atómicas.

Al igual que se vio en el punto anterior, la especificación de las actividades empresariales, define protocolos para facilitar la integración con los procesos de negocio y flujos de trabajo ya existentes y poder inter operar.

9.4 *BPEL4WS*

BPEL4WS (también conocido simplemente como BPEL) es la abreviatura de Business Process Execution Language for Web Services (Lenguaje de ejecución de procesos empresariales para servicios

web). Pese a ser el más reciente de los tres, realmente su vida ha sido la más larga de todos ellos. BPEL4WS surge de la conjunción de dos lenguajes ya existentes, el WSFL de IBM y el XLANG de Microsoft, uniendo en un solo lenguaje lo mejor de las dos partes. La primera implementación de BPEL4WS fue desarrollada por IBM y se publicó mediante alphaWorks bajo el nombre de BPEL4J (<http://www.alphaWorks.ibm.com/tech/bpws4j>).

Con BPEL es posible definir procesos empresariales basados en servicios web, permitiendo crear procesos complejos mediante la colaboración de distintos servicios, como paso de datos o errores, finalización de procesos, etc. Estos servicios web pueden estar a su vez dentro de procesos más complejos. Mediante esta especificación es posible hacer que los procesos se ejecuten en paralelo, en serie, que varíen respecto a parámetros o que se sincronicen entre ellos.

En BPEL se distinguen dos escenarios de uso, que son la implementación de procesos empresariales ejecutables y los no ejecutables abstractos. El segundo caso no es muy utilizado y viene a ser una representación de la interfaz del proceso.

Los documentos BPEL son realmente una representación algorítmica de los diagramas de flujo del proceso. A cada uno de los pasos que se dan en el diagrama se denomina actividad, por lo que existirán unas actividades básicas para describir el proceso. Cada una de ellas se corresponde con un elemento:

- Llamadas a servicios web (elementos <invoke>)
- Esperas de llamadas (elementos <receive>)
- generación de respuestas (elementos <reply>)
- Copias de datos (elementos <assign>)
- Esperas (elementos <wait>)
- No hacer nada (elementos <empty>)
- Terminar (elementos <terminate>)
- Lanzar errores (elementos <throw>)
- Atrapar errores (elementos <catch>)

Además estas actividades pueden combinarse de innumerables formas hasta conseguir completar el gráfico de proceso, sea cual sea su dificultad. Las relaciones entre ellos se realizan igualmente mediante elementos XML:

- Secuencia de actividades (elemento <sequence>)
- Secuencia de actividades en paralelo (elemento <flow>)
- Selección tipo “if” o “case” (elemento <switch>)
- Bucles (elemento <while>)
- Ejecución de una ramificación del gráfico (elementos <pick>)

No se incluye un ejemplo debido a la extensión que ocuparía, pero puede encontrar ejemplos y la especificación completa en <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>

Apéndice A

A.0. Introducción

En el capítulo cinco se realizó una introducción a la especificación del lenguaje UDDI y entre otras cosas, se vieron los distintos mensajes que se podían dar lugar en un registro, bien provenientes de un cliente o de otro registro. En este apéndice se verán unos casos prácticos de utilización de estos mensajes, y como instalar y configurar el servidor de registro que está disponible con el jwsdp (Java Web Service Developer Pack) de Sun.

A.1. Instalación y configuración

Para la instalación en sistemas Windows simplemente es necesario hacer doble clic sobre el fichero *jwsdp-1_0_01-windows-i586.exe*, tras lo cual se abrirá una ventana de diálogo con un asistente que guiará a lo largo de la instalación.

En la instalación sobre Unix pueden plantearse pequeños problemas sobre todo de permisos; para ejecutar el programa de instalación puede hacerse, desde algunos entornos gráficos, simplemente pulsando sobre el archivo, sino, puede procederse desde la línea de comando, que funcionará en todos los sistemas. Para ambos métodos hay que tener en cuenta que posean los permisos apropiados, ya que dependiendo de cómo se copie desde el CDROM, se pueden perder los permisos de ejecución.

Una vez copiado el archivo, para ejecutarlo desde línea de comando la sentencia sería:

```
chmod 700 jwsdp-1_0_01-unix.sh; ./jwsdp-1_0_01-unix.sh
```

Tras lo cual se abrirá una ventana con el asistente de instalación. En ambas instalaciones, se pide un usuario y una contraseña. Este usuario actuará como administrador del servidor Tomcat, que no tiene que ver directamente con el registro UDDI.

Una vez instalado, se pueden repasar los ficheros de configuración para asegurarse que todo es correcto. Por defecto, Tomcat viene preparado para utilizar los puertos 8080 y 8081 para sus conexiones, si esta configuración interfiere con algún servicio ya instalado en su máquina, deberá variar el archivo *server.xml* que está situado en el directorio *conf* dentro del directorio raíz de la instalación. En este archivo es también posible (entre otras cosas) activar las conexiones SSL (Secure Socket Layer, capa de conexión segura) o configurar los archivos de registro de sucesos. Otro documento que puede interesar variar es el archivo *tomcat-users.xml*, donde se almacenan los usuarios privilegiados del servidor Tomcat, así como sus roles y contraseñas. Este archivo puede ser muy útil si uno olvida la contraseña del administrador.

Para que los scripts de ejecución de los servidores funcionen correctamente hay que tener cuidado con tener bien configurada la variable de sistema `JAVA_HOME`, que debe contener el directorio donde se encuentre instalado el JRE de Java. Además en sistemas Unix esta variable debe estar exportada. Por ejemplo en Windows 2000, se puede definir esta variable desde el escritorio pulsando con el botón derecho del ratón sobre “Mi PC” y escogiendo el menú “propiedades...”. Se abre entonces una nueva ventana con varias pestañas; en la pestaña “Avanzado” se encuentra un botón **Variables de entorno**, si lo pulsa se abrirá una nueva ventana en la que es posible añadir, modificar y eliminar las variables a través de los botones **Nueva...**, **Editar...** y **Eliminar...** respectivamente. Esta variable debe ser global a todas las shell que se abran, por lo que dependiendo de la versión de Windows en la que se vaya a trabajar, no es válido definir esta variable en la línea de comando, puesto que no se exporta a las nuevas shell que se deriven de ésta.

En sistemas Unix (shell Bourne) se realizaría mediante:

```
CLASSPATH = $CLASSPATH:$HOME_SOAPL/soap.jar
```

```
export CLASSPATH
```

En estos momentos el sistema está preparado para iniciar el registro UDDI. El registro se compone de dos partes. Por un lado se tiene un servicio que se encargará de gestionar los mensajes del cliente

(programa u otro registro) y por otro una base de datos que mantendrá la información de este registro. Inicie primero el servidor Tomcat, que contiene el servicio que será el encargado de recibir las peticiones y procesarlas, esto se realiza mediante el archivo *startup.bat* en Windows y mediante *startup.sh* en Unix, situados en el directorio *bin* dentro del raíz de la instalación. Una vez que esté en ejecución, se debe poner en marcha la base de datos Xindice. Para ello ejecute el archivo por lotes *xindice-start.bat* si está en Windows o el archivo *xindice-start.sh* si está en Unix, que también están ubicados en el directorio *bin*. En ambos sistemas es probable (en Windows es seguro) que se hayan creado unos iconos de acceso directo a estos scripts. El lugar de localización de éstos depende de la plataforma y del gestor de ventanas, por ejemplo en Windows se puede acceder desde el botón **Inicio**.

Para detener el servidor de Xindice y Tomcat también existen unos scripts, llamados *xindice-stop.sh* (*xindice-stop.bat* en Windows) y *shutdown.sh* (*shutdown.bat* en Windows) respectivamente, también en el directorio *bin*.

Una vez que se han ejecutado los scripts para la ejecución de los servidores, se puede hacer una primera prueba para ver si todo ha ido correctamente. Si en un navegador web introduce la dirección (suponiendo que el Tomcat se está ejecutando en la misma máquina que el navegador, y que está escuchando las peticiones en el puerto 8080):

```
http://localhost:8080/registry-server/RegistryServerServlet
```

El resultado debería ser:

```
RegistryServerServlet.mapUrlToUddiRequest(...)soap-env:ServerUnable to
map URL to UDDI Request: RegistryServerServlet: null
```

Si la salida es un error de tipo 404 o 500, deberá repasar los archivos de configuración y ver que el puerto sobre el que se ejecutará el servidor Tomcat está realmente disponible.

A.2. Registry Browser

Aunque el registro UDDI está pensado sobre todo para sean los programas los que interactúen con él, la personas humanas también tienen acceso a su información, y es posible (necesario en ocasiones) utilizar navegadores o *browsers* para indagar en esta información del registro y obtener los datos necesarios sobre una empresa o un servicio. Con el paquete jwsdp se incluye una utilidad en modo gráfico, para generar mensajes de forma cómoda y poder acceder así a la información necesaria del registro, esta herramienta es el *Registry Browser*.

Para ejecutar el Registry Browser que acompaña a jwsdp, se puede pulsar sobre el icono correspondiente en caso de tenerlo (tiene como leyenda *JAXR Registry Browser*), o si se opta por la línea de comando para ponerlo en marcha, se debe ejecutar el fichero *jaxr-browser.bat* en Windows y *jaxr-browser.sh* en Unix. Si está en Unix, debe tener en cuenta que al ser en modo gráfico tiene que tener acceso a las X Window.

Tras la ejecución se mostrará la pantalla de la figura 1, en la que se pueden distinguir dos pestañas distintas. La primera de ellas (*Browse*) sirve para realizar los mensajes de consulta, y la segunda (*Submissions*) para realizar los mensajes de publicación.

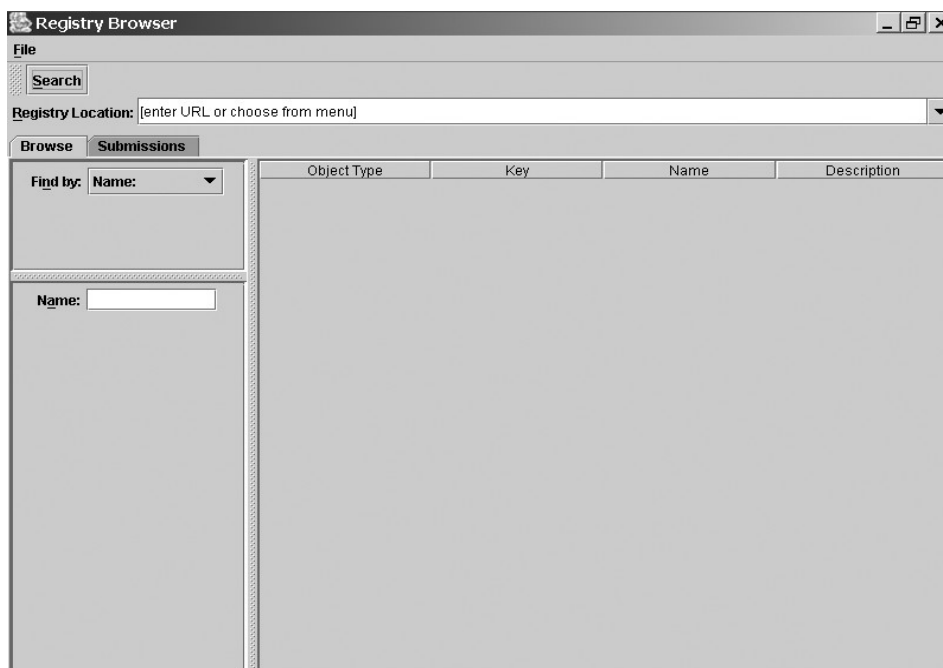


Figura 29: JAXR Registry Browser en ejecución.

Puede ver que en la caja de selección (*Registry Location*, localización del registro), vienen configurados varios registros, dispuestos para ser utilizados. Dependiendo del registro que elija, deberá darse de alta para publicar información. Además puede ver que alguna de las direcciones proporcionadas por defecto pertenecen al UBR (por ejemplo <http://uddi.ibm.com/ubr/inquiryapi>), y otras en cambio son para realizar pruebas (<http://uddi.ibm.com/testregistry/inquiryapi>). Cabe recalcar que muchas de las direcciones de publicación utilizan *https://* en lugar de *http://*, esto es por que se realizan sobre una conexión segura. Aunque esto no es obligatorio da mayor seguridad a la transmisión de los elementos hacia el registro.

La última de las direcciones disponibles, es referente al servidor Tomcat local, que viene en el mismo paquete y será la que se utilizará en ejemplos posteriores. Si la configuración de su red es distinta, deberá ajustar esta dirección para poder realizar las peticiones.

Si recuerda en el punto 5.2.2.2 se vinieron distintos *tModelName* para las clasificaciones de la categoría del negocio. Para hacerse una idea del nivel de clasificación que se puede alcanzar con este tipo de registros, elija *Classifications* (Clasificaciones) en la caja de selección *Find by* (hallar por) del Registry Browser. En este momento podrá ver tres clasificaciones:

1. unspsc-org:unspsc:3-1
2. iso-ch:3166:1999
3. ntis-gov:naics:1997

Si pulsa sobre una de estas clasificaciones verá que aparece un nuevo símbolo a su izquierda, que permite desplegar esta categoría. Si se entretiene “navegando” por las distintas clasificaciones, se dará cuenta la gran cantidad de distintas clasificaciones que hay disponibles y lo difícil que es no poder encasillar un negocio; quizá tenga problemas para indicar uno sólo de los modelos disponibles.

Pruebe a realizar una consulta. Si dispone de conexión a Internet, podrá buscar directamente sobre cualquiera de los registros disponibles, si no posee conexión, podrá consultar al registro local, que se configuró anteriormente. Lamentablemente, el registro no trae información por defecto para realizar pruebas, por lo que se deberá introducir primeramente. No obstante pruebe a introducir un nombre en la casilla *Name*, por ejemplo “PetaSoftware” y elija modo de búsqueda mediante nombre (combo *Find by* valor *name*), por último asegúrese que está seleccionada la dirección del registro local

(<http://localhost:8080/registry-server/RegistryServerServlet>). Pulse sobre el botón *Search* (Buscar) y lógicamente obtendrá un aviso alertándole sobre la no disponibilidad de esta empresa. Más adelante se introducirá información capaz de cumplir las expectativas de esta última búsqueda.

B.2. Ejemplos de mensajes

En el capítulo referente a UDDI, se vieron los distintos tipos de mensajes que podía interpretar el registro UDDI. Al igual que con los mensajes SOAP no es normal trabajar directamente con los mensajes, sino que se emplean utilidades para enmascarar la creación de estos XML, haciendo más rápida la programación y permitiendo al programador centrarse en otros aspectos, como la lógica de la aplicación. No obstante, por motivos didácticos se verán ejemplos en los que se genera directamente el mensaje a transmitir.

Recordará el lector que una de las acciones que había que realizar para poder publicar información en un registro UDDI, es obtener de éste un elemento de autorización. El mensaje que entraba en juego en este caso era `<get_authToken>`. Como atributos de este mensaje se tienen que informar el nombre del usuario y la clave. Para poder verlo en funcionamiento, cree un fichero llamado `getAuthToken.vbs`, cuyo contenido sea:

```
Dim xmlObj, httpObj

Set xmlObj = CreateObject("MSXML.DOMDocument")

xmlObj.loadXML "<s:Envelope
xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/1999/XMLSchema'><s:Body><get_authToken
xmlns='urn:uddi-org:api_v2' cred='testuser' generic='2.0'
userID='testuser' /></s:Body></s:Envelope>"

MsgBox xmlObj.xml, , "Mensaje SOAP de petición"

Set httpObj = CreateObject("Microsoft.XMLHTTP")

httpObj.open "POST", "http://localhost:8080/registry-
server/RegistryServerServlet"

httpObj.send (xmlObj)

While httpObj.readyState <> 4

Wend

MsgBox httpObj.responseText, , "Mensaje SOAP respuesta"
```

Verá que el nombre del usuario es `testuser` al igual que su clave. Este usuario viene configurado por defecto en el registro y servirá perfectamente para éstos ejemplos. Antes de ejecutarlo asegúrese de que los servidores están en funcionamiento (tanto Tomcat como Xindice), y que ha instalado el paquete de Microsoft de análisis de XML. Si ejecuta este script, verá una salida semejante a:



Figura 30: Mensaje respuesta a petición de autorización.

El número que representa el elemento de autorización por parte del registro será distinto en cada ocasión que lo demande mediante `<get_authToken>` (puede probar a lanzarlo de nuevo). Este elemento será válido mientras no se anule usando un mensaje `<discard_authToken>`, caduque su validez o se apague el servidor.

Anteriormente se vio una consulta mediante el Registry Browser, si quisiera hacer esta misma petición utilizando directamente el mensaje SOAP, simplemente debe generar un documento XML con el elemento `<find_business>` en su `<Body>`. Como ejemplo se expone un programa que consulta sobre el nombre de un negocio a partir del mensaje de petición SOAP. Para crear este programa genere un nuevo fichero llamado `consultaUDDI.vbs`, y en el introduzca el siguiente código:

```
Dim xmlObj, httpObj, strBuss

Set xmlObj = CreateObject("MSXML.DOMDocument")

strBuss = InputBox("Introduzca el nombre de un negocio:")

xmlObj.loadXML "<s:Envelope
xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/1999/XMLSchema'><s:Body><find_business
generic='2.0' xmlns='urn:uddi-org:api_v2'><name>" & strBuss&
"</name></find_business></s:Body></s:Envelope>"

MsgBox xmlObj.xml, , "Mensaje SOAP de petición"

Set httpObj = CreateObject("Microsoft.XMLHTTP")

httpObj.open "POST", "http://localhost:8080/registry-
server/RegistryServerServlet"

httpObj.send (xmlObj)

While httpObj.readyState <> 4

Wend

MsgBox httpObj.responseText, , "Mensaje SOAP respuesta"
```

Actualmente el registro local aún no contiene datos, por lo que las respuestas a todas las peticiones de información acerca de un negocio, tendrán como resultado un mensaje vacío (el mensaje no será vacío, pero sí lo será el elemento `<businessList>`). Si quisiera realizar consultas a otros registros debería cambiar la línea

```
httpObj.open "POST", "http://localhost:8080/registry-
server/RegistryServerServlet"
```

Por ejemplo para consultar al UBR de IBM, cámbiela por:

```
httpObj.open "POST", "http://uddi.ibm.com/ubr/inquiryapi"
```

Para continuar haciendo pruebas y jugando con el registro local, introduzca la información de nuevo negocio que se llamará “PetaSoftware”.

Para poder introducir nueva información en el registro hay que conseguir un elemento de autenticación, y una vez conseguido, usarlo como parte de los datos del negocio a introducir. En este ejemplo se hará manualmente. Supondrá el lector que esta no es la forma normal de realizar esta acción, pero así podrá darse cuenta que realmente el mensaje para la concesión del elemento de autenticación y el mensaje referente a la introducción de datos son totalmente independientes, si exceptuamos que uno de ellos lleva como parte de su información la respuesta del otro.

Obtenga el elemento de autenticación mediante el primer programa realizado en este capítulo (fichero *getAuthToken.vbs*), y anótelos en un papel, puesto que lo utilizará como parte del código del programa que introducirá los datos. En el ejemplo se utilizará el elemento de autenticación “2384939d2384939”, aunque debe tener en cuenta que este elemento no le funcionará a usted.

Cree un nuevo fichero llamado *introduceDatos.vbs* e introduzca el siguiente código. Como la información a introducir es elevada, se ha dividido la cadena del mensaje en varias líneas, tenga cuidado de escribir correctamente las cadenas individuales y los elementos de unión (“+ _”):

```
Dim xmlObj, httpObj

Set xmlObj = CreateObject("MSXML.DOMDocument")

xmlObj.loadXML "<s:Envelope
xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/1999/XMLSchema'><s:Body>" + _

"<save_business generic='2.0' xmlns='urn:uddi-org:api_v2'>" + _

"<authInfo>2384939d2384939</authInfo>" + _

"<businessEntity businessKey=' '>" + _

"<discoveryURLs>" + _

"  <discoveryURL
useType='businessEntity'>http://acme.com/show.jsp</discoveryURL>" + _

"</discoveryURLs>" + _

"<name>PetaSoftware</name>" + _

"<description xml:lang='es'>El mejor software</description>" + _

"<description xml:lang='en'>The best software</description>" + _

"<contacts>" + _

"  <contact useType='reception'>" + _

"    <description xml:lang='es'>Contact description</description>"
+ _

"  <personName>J. Torres</personName>" + _
```

```

    "<email useType='main'>jtorres@acme.com</email>" + _
    "<phone useType='main'>+34803724532</phone>" + _
    "<phone useType='secondary'>+34803724532</phone>" + _
    "<address useType='use' sortCode='a'>" + _
        "<addressLine>J.L. Arrese</addressLine>" + _
        "<addressLine>Valladolid</addressLine>" + _
    "</address>" + _
    "</contact>" + _
    "</contacts>" + _
    "</businessEntity>" + _
    "</save_business></s:Body></s:Envelope>"

```

```
MsgBox xmlObj.xml, , "Mensaje SOAP de petición"
```

```
Set httpObj = CreateObject("Microsoft.XMLHTTP")
```

```
httpObj.open "POST", "http://localhost:8080/registry-
server/RegistryServerServlet"
```

```
httpObj.send (xmlObj)
```

```
While httpObj.readyState <> 4
```

```
Wend
```

```
MsgBox httpObj.responseText, , "Mensaje SOAP respuesta"
```

Al ejecutar este nuevo programa, se introducirá la información correspondiente en el registro. Si obtuvo problemas de autenticación, es posible que sea a causa de que se le haya caducado el elemento de seguridad o bien lo haya escrito mal, proceda entonces a obtener una nueva validación con un nuevo mensaje `<get_authToken>`.

La información recibida por el programa cliente tras la inserción de los datos enviados es mostrada en la figura 3.

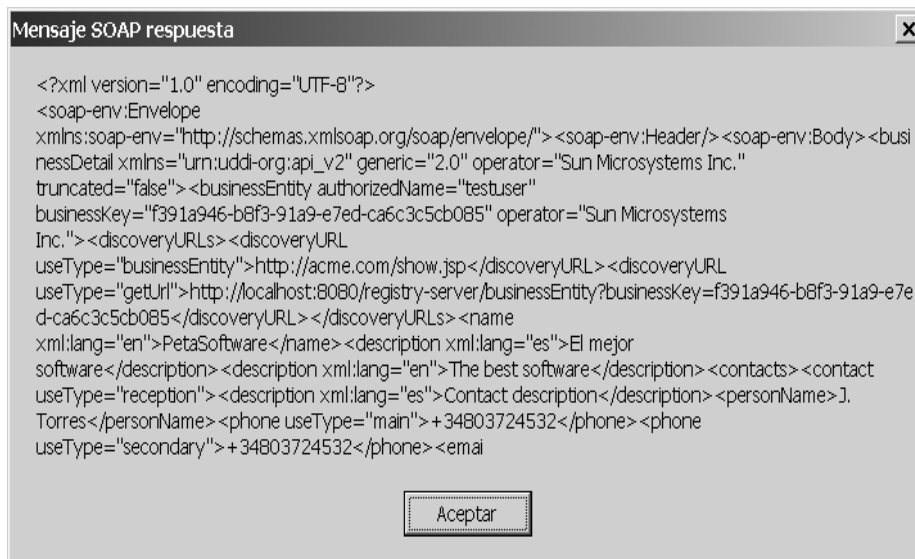


Figura 31: Mensaje respuesta a petición de inserción de datos.

En ésta figura se puede ver que el registro ha asignado un *businessKey* a la empresa, en este caso su valor es: f391a946-b8f3-91a9-e7ed-ca6c3c5cb085. Este valor es único y sirve para identificar unívocamente a este negocio. A partir de este momento este negocio tiene que ser accedido usando este UUID para hacerle referencia.

Si ahora ejecuta el fichero creado en el segundo ejemplo, y proporciona como nombre de empresa a consultar “PetaSoftware”, verá que la respuesta obtenida en este caso ya no está vacía, sino que contiene datos del nuevo negocio introducido. La información que se recibe en este caso es el *businessKey* y las descripciones del negocio. Para obtener más información se deben utilizar otro tipo de mensajes de consulta.



Figura 32: Mensaje respuesta a consulta UDDI.vbs.

Con el tipo de consulta que se ha realizado, el cliente obtiene una lista de todos aquellos negocios que concuerdan con la cadena de texto especificada. Por ejemplo la consulta se podría haber realizado usando como nombre de negocio “Peta” y se hubieran obtenido idénticos resultados. Por otra parte, hay que tener en cuenta que lo que se recibe es una lista de elementos *<businessInfo>* contenidos en otro elemento del tipo *<businessInfos>*.

Si quisiera obtener más información acerca de este negocio, podría realizarlo mediante un mensaje como el siguiente:

```
<get_businessDetailExt generic="2.0" xmlns="urn:uddi-org:api_v2">
  <businessKey>f391a946-b8f3-91a9-e7ed-ca6c3c5cb085</businessKey>
</get_businessDetailExt>
```

Se deja como simple ejercicio al lector su creación.

Ahora que el registro local ya tiene información puede consultarlo nuevamente mediante la utilidad Registry Browser. Para ello escriba en la casilla hallar por (*Find by*) el nombre de la empresa a buscar (Peta por ejemplo) y pulse el botón **Search**. Aparecerá una lista con los servicios que cumplan las especificaciones indicadas. En el caso de la figura se muestran dos compañías, una de software (PetaSoftware) y otra de petardos (Petardos) ; la búsqueda se realizó mediante la palabra clave "peta".

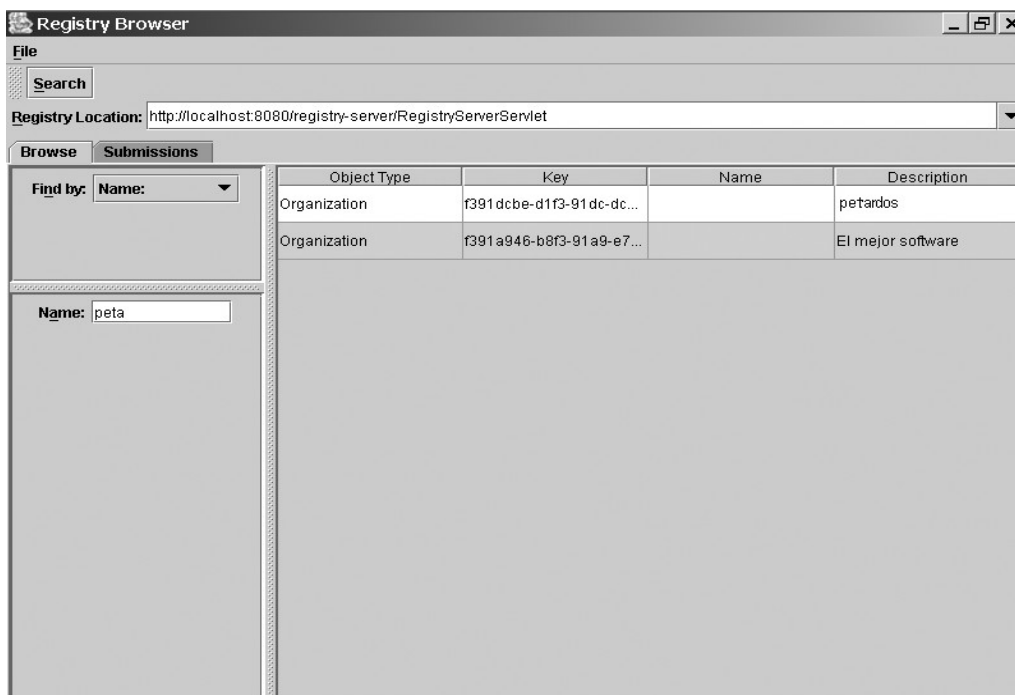


Figura 33: Registry Browser mostrando una consulta.

Se puede obtener más información sobre el negocio haciendo doble clic sobre él. La nueva información se muestra en una ventana aparte y expone datos como el contacto primario, además da la posibilidad de revisar los servicios que ofrece esta compañía (vacíos actualmente, aunque se podrían haber proporcionado en el momento de la creación de la empresa en el registro).

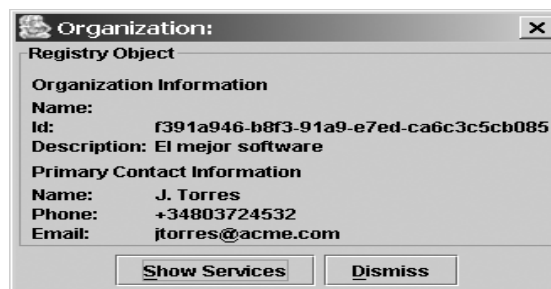


Figura 34: Detalle de la consulta.

Ya se comentó que esta no es la forma normal de trabajar, puesto que si así se hiciera piense que para hacer una simple inserción de datos en el registro habría que hacer varios mensajes y anotarse algunos datos como el elemento de autenticación. Para hacer los mensajes de publicación y consulta, al igual que se hizo con SOAP, se usan librerías que proporcionan las distintas casas de software, para alejar al programador de esta tediosa y poco gratificante tarea.

Por ejemplo si se trabaja con Java para realizar los mensajes SOAP para el registro UDDI, se pueden utilizar clases proporcionadas por otros fabricantes, como UDDI4J de IBM o trabajar con el API proporcionado por el propio Sun. Este API es denominado JAXR (Java API for XML Registries).

Como introducción al uso de JAXR, a continuación se presenta un ejemplo de listados de empresas de un registro a partir de un nombre y un sencillo programa de borrado de negocios del registro.

Para el primer ejemplo se usará una clase que aceptará uno o dos parámetros, el primer parámetro es el nombre de la empresa (o semejante) y el segundo será optativo y representará la dirección del registro UDDI a consultar. Si no se proporcionase el segundo parámetro, se usaría el suministrado por defecto en el programa.

```
static String registro =
    "http://localhost:8080/registry-server/RegistryServerServlet";
```

Las respuestas se pueden ordenar teniendo en cuenta muchas opciones, por ejemplo alfabéticamente, por fecha, etc. En el ejemplo se ordenarán por fecha mediante :

```
fQ.add(FindQualifier.SORT_BY_DATE_ASC); //orden por fecha
```

La consulta al registro se realiza mediante:

```
bqm.findOrganizations(fQ, names, null, null, null, null)
```

en la que el primer parámetro es una colección con los modificadores de la consulta y el segundo es otra colección con los patrones de los nombres a buscar. El resto de los parámetros sirven para filtrar la consulta por otro tipo de áreas como números DUNS u otro tipo de clasificaciones, pero en este caso no se utilizarán.

La búsqueda da lugar a un objeto de tipo *BulkResponse* del que se puede obtener una colección de organizaciones.

El código completo del programa que se detalla a continuación es el contenido del fichero FindBusiness.java:

```
import java.util.ArrayList;

import java.util.Properties;

import java.util.Collection;

import java.util.Iterator;

import javax.xml.registry.*;

import javax.xml.registry.infomodel.RegistryObject;

import javax.xml.registry.infomodel.Organization;

import javax.xml.registry.infomodel.Service;
```

```
public class FindBusiness {

    static String registro =

        "http://localhost:8080/registry-server/RegistryServerServlet";

    public FindBusiness() {

    }

    public static void main(String[] args) throws JAXRException {

        FindBusiness fb = new FindBusiness();

        //si se introduce la dirección del registro, se toma ésta como
        destino de los mensajes

        if (args.length == 2) {

            registro = args[1];

        }

        fb.executeQueryTest(registro, args[0]);

    }

    public void executeQueryTest(String reg, String nombre) throws
    JAXRException {

        try {

            Properties props = new Properties();

            props.setProperty("javax.xml.registry.queryManagerURL", reg);

            props.setProperty("javax.xml.registry.factoryClass",

                "com.sun.xml.registry.uddi.ConnectionFactoryIm
pl");

            ConnectionFactory factory = ConnectionFactory.newInstance();

            factory.setProperties(props);

            Connection conn = factory.createConnection();

            RegistryService rs = conn.getRegistryService();
```

```
BusinessQueryManager bqm = rs.getBusinessQueryManager();

//modificadores de la búsqueda

Collection fQ = new ArrayList();

fQ.add(FindQualifier.SORT_BY_DATE_ASC); //orden por fecha

// nombres a buscar

ArrayList names = new ArrayList();

names.add(nombre);

// se realiza la consulta

BulkResponse br = bqm.findOrganizations(fQ, names, null, null,
null, null);

//test del resultado de la consulta

if (br.getStatus() == 0) {

    Collection comp = br.getCollection();

    System.out.println("Fin de consulta.");

    System.out.println("Resultado: ");

    //obtener resultados

    Iterator iter = comp.iterator();

    while (iter.hasNext()) {

        Organization org = (Organization) iter.next();

        System.out.println("Nombre: " + org.getName().getValue());

        System.out.println("Descripción: " +
org.getDescription().getValue());

        System.out.println("UUID: " + org.getKey().getId());

        System.out.println("Contacto principal: " +
org.getPrimaryContact().getPersonName().getFullName());

        System.out.println("-----");
    }
}
```



```

    }
}
else {
    //errores

    System.out.println("Error durante la ejecución:");

    Collection ex = br.getExceptions();

    Iterator iter = ex.iterator();

    while (iter.hasNext()) {

        Exception e = (Exception) iter.next();

        System.out.println(e.toString());

    }

}

}

}

catch (JAXRException e) {

    e.printStackTrace();

}

}

}

```

Para ejecutarlo le tiene que dar como primer parámetro el nombre de la empresa y si desea usar otro registro como destino de la consulta, también debe proporcionar la dirección de éste como segundo parámetro del programa.

Como se puede apreciar, una vez que se obtiene el objeto *Organization* es muy fácil acceder a cada una de sus partes, como por ejemplo el UUID o el nombre del contacto. En este ejemplo no se han obtenido otras informaciones importantes como podrían ser los servicios de las empresas consultadas o sus direcciones. Para hacerlo sería de modo semejante a como se han consultado anteriormente otros aspectos de la organización, teniendo en cuenta el tipo de dato que devuelve y procesándolo debidamente. Por ejemplo las funciones *getUsers()* o *getServices()* de *Organization* devolverán colecciones de datos, por lo que habrá que tratarlos individualmente. Como ejemplo, para explorar los usuarios se realizaría del siguiente modo:

```

Collection users = org.getUsers();

Iterator iter = users.iterator();

while (iter.hasNext()) {

```

```
        User user = (User) iter.next();

        System.out.println("Nombre de usuario: " +
            user.getPersonName().getFullName() );
    }
}
```

Tenga en cuenta que incluso alguna de las llamadas de objetos de colecciones pueden contener a su vez más colecciones, como es el caso de la función *getTelephoneNumbers()* del objeto *User*, por lo que es muy aconsejable consultar los documentos del API JAXR que proporciona Sun.

Por último, para acabar con este apéndice, se muestra un programa en java que permite borrar entradas del registro. En él se puede observar el funcionamiento del elemento de autorización. El código del fichero *DelBusiness.java* es el siguiente:

```
import javax.xml.registry.*;

import java.net.PasswordAuthentication;

import java.util.Iterator;

import java.util.Properties;

import java.util.Collection;

import java.util.ArrayList;

import java.util.HashSet;

import java.util.Set;

public class DelBusiness {

    static String registro =

        "http://localhost:8080/registry-server/RegistryServerServlet";

    public DelBusiness() {

    }

    public static void main(String[] args) throws JAXRException {

        DelBusiness db = new DelBusiness();

        db.deleteBusiness(args[0], args[1], args[2]);

    }
}
```

```
private void deleteBusiness(String login, String pass, String uuid)
throws
    JAXRException {
    Properties props = new Properties();
    props.setProperty("javax.xml.registry.queryManagerURL", registro);
    props.setProperty("javax.xml.registry.factoryClass",
        "com.sun.xml.registry.uddi.ConnectionFactoryImpl
");

    ConnectionFactory factory = ConnectionFactory.newInstance();
    factory.setProperties(props);
    Connection conn = factory.createConnection();
    RegistryService rs = conn.getRegistryService();
    BusinessLifecycleManager blcm = rs.getBusinessLifecycleManager();
    // clave de entrada a borrar
    javax.xml.registry.infomodel.Key key = blcm.createKey(uuid);
    // creación de credenciales
    PasswordAuthentication passwdAuth =
        new PasswordAuthentication(login,
            pass.toCharArray());

    Set creds = new HashSet();
    creds.add(passwdAuth);
    conn.setCredentials(creds);
    Collection keys = new ArrayList();
    keys.add(key);
    // borrado
    BulkResponse response = blcm.deleteOrganizations(keys);
    Collection exceptions = response.getExceptions();
    if (exceptions != null) {
```

```
System.out.println("Se han producido errores al borrar");

Iterator excIter = exceptions.iterator();

Exception exception = null;

while (excIter.hasNext()) {

    exception = (Exception) excIter.next();

    System.out.println("Excepción: " +

                        exception.toString());

}

}

else{

    System.out.println("Se ha borrado correctamente.");

}

}

}
```

A este programa se le ha de dar como parámetros en nombre de usuario, la clave, y el UUID de la empresa a eliminar. Si interesara cambiar la dirección del registro UDDI, habría que añadir un nuevo parámetro a la entrada del programa o cambiarlo directamente en el código de la aplicación.

Apéndice B

B.0 Introducción

Cuando se realizan programas informáticos es muy normal que las cosas no salgan como se desea a la primera, que los programas no funcionen de la manera esperada o que se produzcan errores durante la ejecución. Para poder detectar y arreglar dichos fallos lo normal es proceder a la depuración del código, seguir su comportamiento línea a línea y poder detectar cualquier irregularidad que se de lugar.

Pero esta tarea es más fácil en unas ocasiones que en otras, por ejemplo, si se realizan desarrollos de servicios web en un entorno en el que se permita la integración del servidor y el cliente, será posible depurar el código de ambos lados (servidor como del cliente), pero si no es posible usar este tipo de entornos o bien no se tiene acceso directo al código del servidor, la tarea puede hacerse realmente difícil, sobre todo si el problema de comunicación entre servicio y cliente está en la interpretación o construcción de los mensajes SOAP; en este caso se convierte casi en un problema de prueba y error en el que el programador hace un cambio en el cliente y prueba a ver si el servicio responde de forma correcta, en caso de que no lo haga piensa en cual ha podido ser la causa y vuelve a realizar una nueva codificación y una nueva prueba.

Podrá comprender que es un sistema de depurar la aplicación muy lento, y es posible que jamás se consiga encontrar la causa del error. Un visor de los mensajes SOAP que se den entre el cliente y servicio es una herramienta que puede ayudar en gran medida a la localización de los problemas. Unas herramientas de este tipo son `TcpTunnelGui` y `tcpTrace`. Existen otras muchas herramientas de funcionamiento parecido y con mayores prestaciones, tanto libres como de pago, incluso en algunos entornos de desarrollo, se incluyen como herramienta complementaria.

B.1 `TcpTunnelGui`

Como se comentó en el capítulo seis, dentro del paquete de clases que ofrece Apache SOAP, se encuentra una utilidad llamada `TcpTunnelGui`, que permite registrar los mensajes que se dan lugar entre el servicio web y el cliente que lo accede.

`TcpTunnelGui` hace de pasarela entre el cliente y el servidor, es decir, este programa se instala entre las dos partes y se encarga de recibir el mensaje del cliente, mostrar dicho mensaje por pantalla, y traducir la dirección destino para poder redirigirlo hacia el servicio web. Una vez procesada la petición por parte del web service, será este mismo programa el encargado de recibir el mensaje de respuesta, que volverá a mostrar en pantalla y por último se lo devolverá al cliente que originó la petición.

`TcpTunnelGui` viene incluido con las librerías de Apache SOAP. Para hacerlo accesible hay que incluirlo en la variable `CLASSPATH` mediante :

```
set CLASSPATH = %CLASSPATH%;%HOME_SOAPL%\soap.jar
```

en Windows o mediante

```
CLASSPATH = $CLASSPATH:$HOME_SOAPL/soap.jar
```

en Bourne shell. Donde `SOAPL` es el lugar de instalación de las librerías.

El código utilizado para ver cómo funciona este programa se aprovecharán los ejemplos realizados en el capítulo seis. Para poder utilizar `TcpTunnelGui` y poder observar los mensajes SOAP que se dan, se debe preparar el cliente para que realice las peticiones al puerto donde escuchará este programa, en lugar de realizarlas sobre el propio servicio.

Se muestra a continuación los cambios a realizar sobre el ejemplo “hola mundo” realizado en Java para poder utilizar `TcpTunnelGui`.

El fragmento de listado siguiente es el correspondiente al cliente del servicio realizado en Java, más concretamente a la función `getResponse()`. La línea a sustituir para cambiar la dirección de envío de la petición es la correspondiente a la definición del URL:

```
public static String getResponse() throws Exception{

    String aQuien = "MiNombre";
```

```
// línea a sustituir.

URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");

Call call = new Call ();

call.setTargetObjectURI("urn:holawebsevice");

//...

debe ser sustituida por:

URL url = new URL ("http://127.0.0.1:80/soap/servlet/rpcrouter");
```

Con esto se consigue que el mensaje enviado por el cliente no sea dirigido hacia el puerto 8080 del servidor con dirección 127.0.0.1 sino hacia el 80, que será donde estará escuchando TcpTunnelGui. Es posible que el lector deba variar también la dirección; tenga en cuenta que para poder usar esta herramienta, el mensaje debe dirigirse al ordenador donde se ejecutará la misma, por lo que el código debe reflejar esta dirección, en otras palabras, la dirección de envío del mensaje del cliente es la dirección en la que se encuentra ejecutándose el monitor.

Para poner en funcionamiento la herramienta TcpTunnelGui se utiliza (desde línea de comando) la siguiente sintaxis.

```
java org.apache.soap.util.net.TcpTunnelGui puerto_escucha
dirección_destino puerto_destino
```

El lector puede suponer que el *puerto_escucha* es el puerto al que se dirigirán los mensajes del cliente, la *dirección_destino* es la dirección (sólo la dirección IP o nombre, sin protocolos ni directorios) del servidor que alberga el servicio y *puerto_destino* es el puerto que el servidor utiliza para escuchar las peticiones de los servicios. Para visualizar los mensajes del servicio del cliente retocado anteriormente y suponiendo que tanto cliente como servidor como monitor están en la misma máquina, el comando a ejecutar sería:

```
java org.apache.soap.util.net.TcpTunnelGui 80 localhost 8080
```

Una vez que la herramienta está en funcionamiento, se puede ejecutar el nuevo cliente, y la salida de TcpTunnelGui tras los mensajes entre cliente y servidor será semejante a:

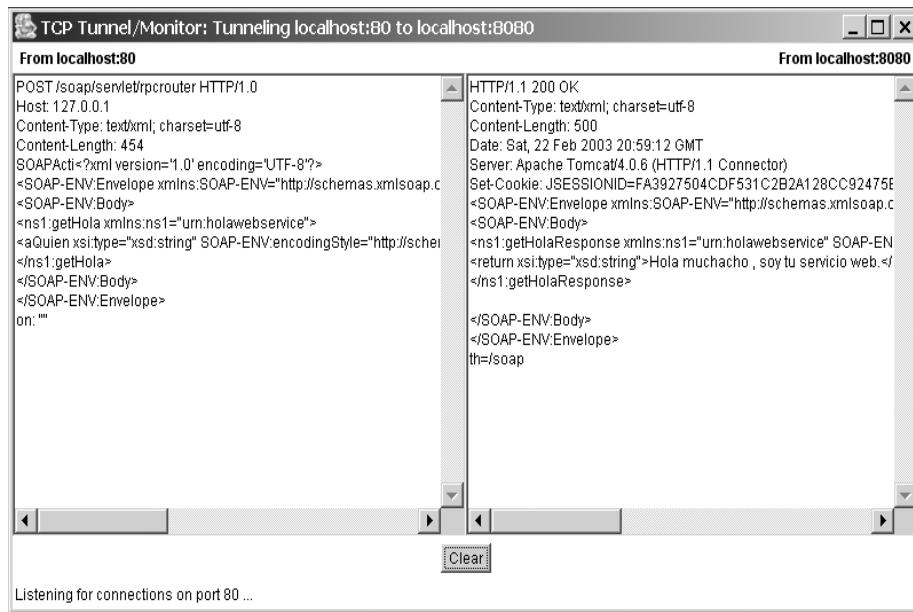


Figura 35: TcpTunnelGui en funcionamiento.

Para ver como se daría un error por un URI incorrecto y poder observar la forma de su mensaje SOAP, se puede variar éste en el código (comprobar que no existe ningún URI que sea igual al nuevo), por ejemplo, se puede sustituir la línea:

```
call.setTargetObjectURI("urn:holawebservice");
```

Por

```
call.setTargetObjectURI("urn:holaweb");
```

Al ejecutar este nuevo cliente, el programa TcpTunnelGui reflejará el problema.

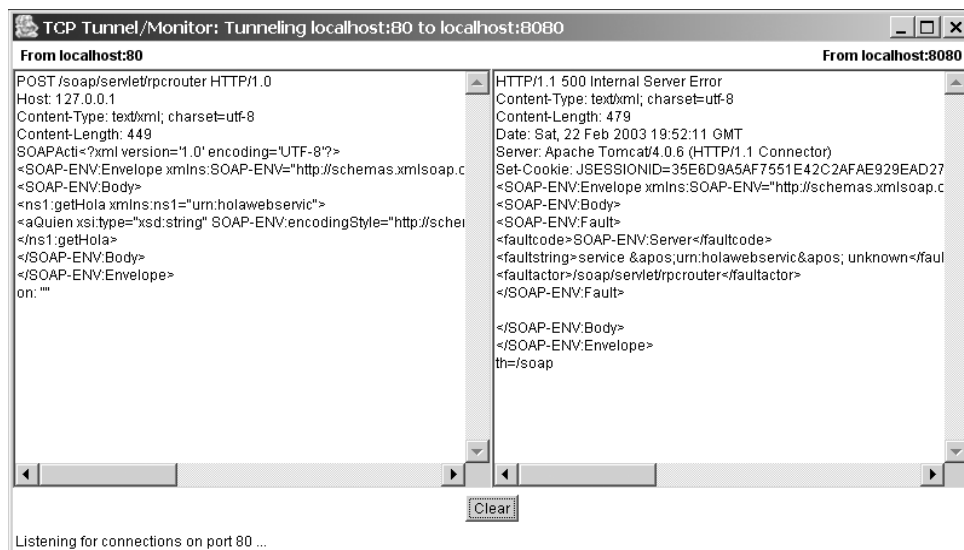


Figura 36: TcpTunnelGui mostrando un error.

Si en lugar del ejemplo de hola mundo se varía el código del servicio que se realizó para transmitir ficheros usando SOAP con *attachments* (ejemplo del punto 6.5.2), se pueden apreciar las distintas partes del mensaje y cómo la información es añadida al final del mismo, tras la etiqueta `</Envelope>`. Para mostrar los mensajes de este ejemplo, se debe variar también el código fuente del cliente para que los mensajes se realicen a través de este monitor. Para ello se sustituye la línea

```
URL url = new URL("http://localhost:8080/soap/servlet/rpcrouter");
```

Por

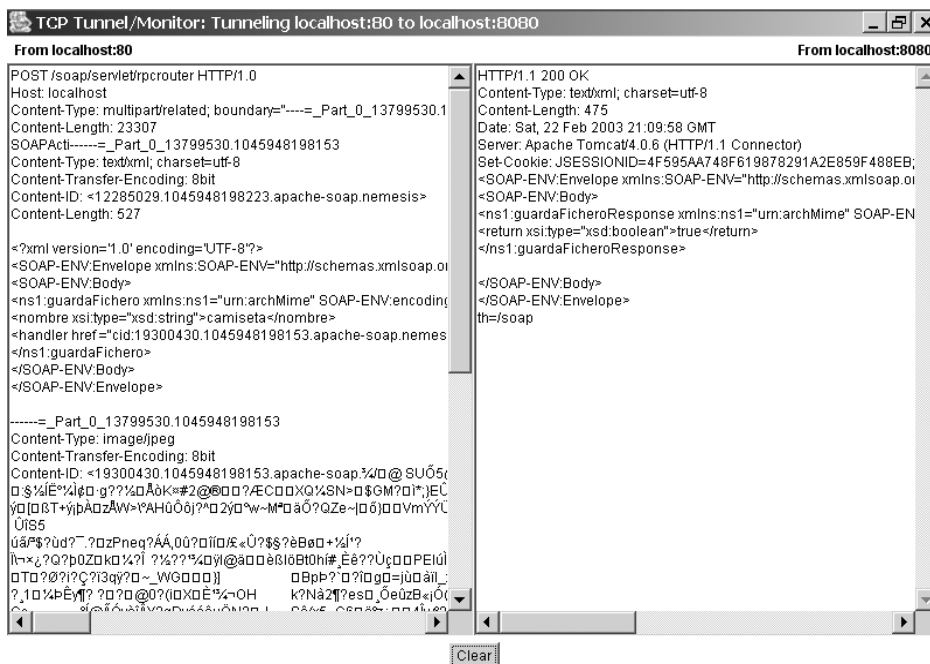
```
URL url = new URL("http://localhost:80/soap/servlet/rpcrouter");
```

Además se ha variado la línea en la que se indica la localización del fichero a enviar, ya que en este caso se enviará un fichero gráfico situado en el directorio raíz del disco duro:

```
DataSource ds = new ByteArrayDataSource(new
File("c:\\camiseta.jpg"), null);
```

El lector se debe asegurar que el fichero a transmitir existe realmente, ya que en caso contrario, el programa lanzaría una excepción. Tras poner en marcha el monitor y ejecutar el nuevo cliente la salida obtenida es la siguiente:

TcpTunnelGui se puede utilizar tanto en sistemas Unix como en Windows, más concretamente se puede usar en todos aquellos sistemas que soporten Java y tengan entorno gráfico.



Listening for connections on port 80 ...

Figura 37: TcpTunnelGui mostrando un mensaje SOAP con attachment.

B.1 tcpTrace

En el CD que acompaña a la guía se incluye un programa llamado tcpTrace, que puede que al lector le resulte más fácil de utilizar. La pega de este programa es que sólo funciona bajo Windows. Para ejecutarlo vale con hacer doble clic sobre el icono del ejecutable, tras lo cual aparece una pantalla

pidiendo la información necesaria para realizar la escucha entre cliente y servicio. Al igual que con TcpTunnelGui, los parámetros a completar son

- puerto de escucha.
- dirección del host que hospeda el servicio.
- puerto donde redireccionar la petición.

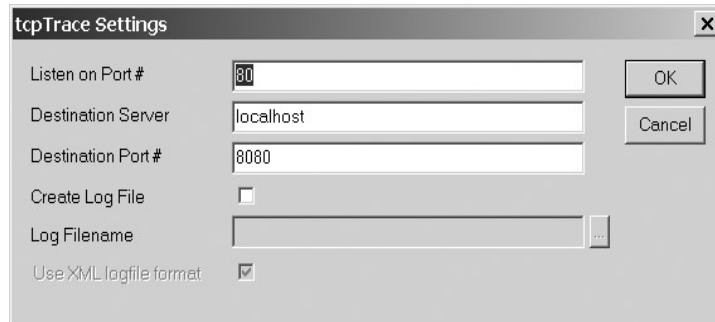
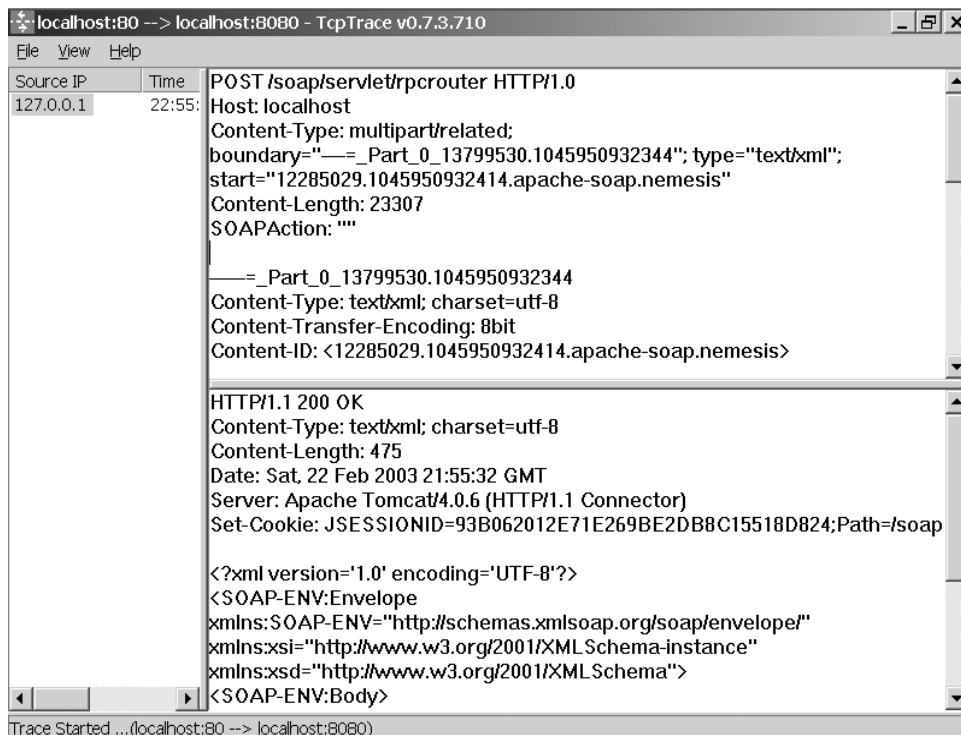


Figura 38: Pantalla de configuración de la conexión de tcpTrace.

Una vez completada esta información, el funcionamiento es idéntico al de la herramienta descrita anteriormente. Si mientras está en funcionamiento tcpTrace se ejecuta el cliente modificado, se obtienen los mensajes entre cliente y servidor tal y como muestra la figura siguiente, en la que se ha utilizado el ejemplo de paso de fichero anexo.



Hay que tener cuidado con la elección de los puertos de escucha cuando se usan este tipo de herramientas, ya que tanto la dirección como el puerto donde se hospeda el servicio vienen fijados por el propio servicio, pero el puerto de escucha de la herramienta no y es posible que se tengan otras aplicaciones en marcha ocupando algunos puertos, y si se tiene la mala suerte de escoger uno que ya está siendo utilizado, estas herramientas no funcionarán correctamente. Por ejemplo, en los casos anteriores se ha estado usando el puerto 80 para registrar los mensajes, aunque no es una buena opción, porque mucha gente tiene este puerto usado para servidores de páginas web. Una opción más sensata podría ser por ejemplo el puerto 8085 que no es usado por ninguna aplicación convencional.

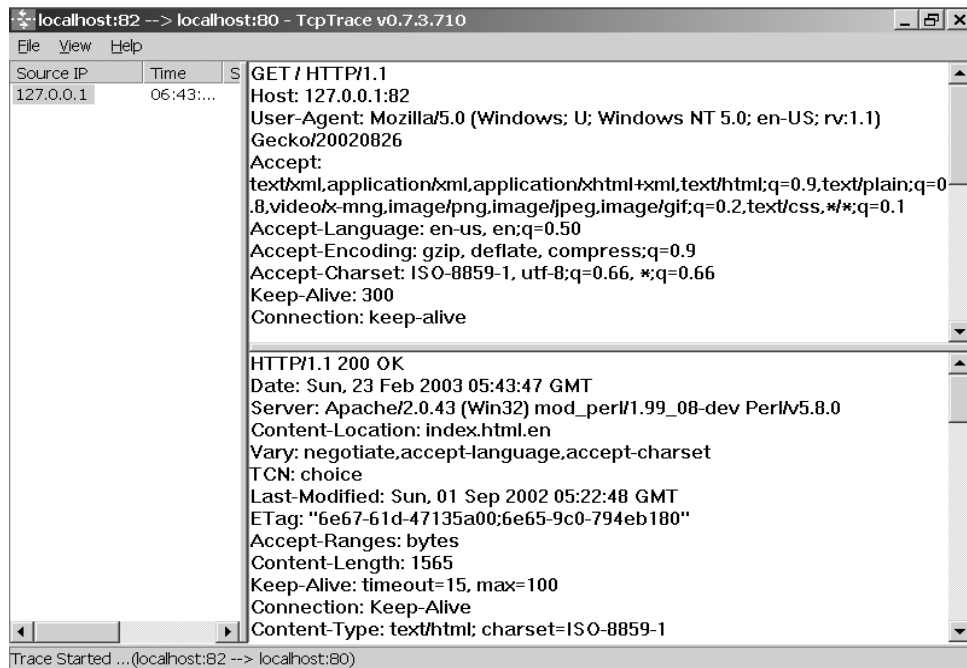
Si varía el puerto de escucha acuérdesese de variar también el código del cliente que enviará los mensajes.

El hecho de que los ejemplos de uso de estas herramientas se hayan expuesto con los servicios realizados en Java no implica que no se pueda utilizar el monitor para comprobar los mensajes que se dan entre servicios y clientes realizados en otros lenguajes. Si se varían los clientes creados por ejemplo en Visual Basic Script, la salida debe ser casi idéntica (variarán las cabeceras y algún *namespace*).

Estas herramientas se pueden utilizar para escuchar otro tipo de mensajes, no hace falta que sean SOAP, aunque sí que funcionen de forma semejante, es decir mediante mensajes, como SMTP, POP3, http, etc. Por ejemplo se puede configurar el monitor para interponerse entre navegador web y la página accedida. Para poder desviar la petición hacia el puerto 80 de la máquina local y escuchar las peticiones a la página web Google se realizaría mediante:

```
java org.apache.soap.util.net.TcpTunnelGui 80 www.google.com 80
```

Ahora solamente hay que hacer la petición desde el navegador a la dirección `http://127.0.0.1:80` y el monitor comenzará a reflejar los datos que circulan entre el navegador y el servidor de páginas web.



The screenshot shows the tcpTrace application window. The title bar reads "localhost:82 --> localhost:80 - TcpTrace v0.7.3.710". The main window is divided into a table on the left and a text area on the right. The table has columns for "Source IP", "Time", and "S". The first row shows "127.0.0.1", "06:43:...", and "S". The text area displays the following content:

```
GET / HTTP/1.1
Host: 127.0.0.1:82
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.1)
Gecko/20020826
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0
.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,text/css,*/*;q=0.1
Accept-Language: en-us, en;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive

HTTP/1.1 200 OK
Date: Sun, 23 Feb 2003 05:43:47 GMT
Server: Apache/2.0.43 (Win32) mod_perl/1.99_08-dev Perl/v5.8.0
Content-Location: index.html.en
Vary: negotiate,accept-language,accept-charset
TCN: choice
Last-Modified: Sun, 01 Sep 2002 05:22:48 GMT
ETag: "6e67-61d-47135a00:6e65-9c0-794eb180"
Accept-Ranges: bytes
Content-Length: 1565
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1
```

At the bottom of the window, it says "Trace Started ... (localhost:82 --> localhost:80)".

Figura 39: tcpTrace mostrando una petición HTTP.

Apéndice C

C.0 Glosario

- API

Acrónimo de application program interface, o interfaz de programación de la aplicación, conjunto de normas, funciones y protocolos para la construcción de aplicaciones. Permite describir el enlace de una aplicación con el resto.

- BEEP

Nuevo protocolo de transporte basado en XML que permite conexiones tipo duplex y transporte asíncrono. Puede obtenerse más información en <http://www.bxxp.org/>.

- BPEL4WS

Acrónimo de Business Process Execution Language for Web Services (Lenguaje de ejecución de procesos empresariales para servicios web). Permite definir procesos empresariales basados en servicios web.

- COM

Component Object Model. Modelo de código binario desarrollado por Microsoft. Permite que objetos que cumplan esta especificación sean accedidos por aplicaciones que sean compatibles COM. Tanto las tecnologías OLE como ActiveX están basadas en COM.

- CORBA

Es la abreviatura de Common Object Request Broker Architecture. Esta arquitectura permite a los objetos comunicarse entre ellos sin importar el lenguaje en el que se encuentran escritos o el sistema operativo en el que funcionan. Fue desarrollado por la OMG.

- CPAN

Comprehensive Perl Archive Network o red de archivos de Perl, se trata de una colección de recursos disponibles a través de Internet para los usuarios de Perl. Entre estos recursos se encuentran librerías, documentos, actualizaciones, trucos de programación, etc.

- DAML-S

DARPA Agent Markup Language es otro lenguaje de descripción de servicios web. Su página web es <http://daml.semanticweb.org/>.

- DCOM

Acrónimo de Distributed Component Object Model (modelo de componentes distribuidos), es una implementación del estándar CORBA. Surge como extensión de COM (Component Object Model), para soportar objetos distribuidos a través de la red. Al contrario que CORBA, DCOM sólo funciona sobre Windows

- DES

Algoritmo de cifrado simétrico más extendido del mundo. Diseñado por NSA, se basa en el algoritmo LUCIFER de IBM.

- DIME

Es el acrónimo de Direct Internet Message Encapsulation. Se trata de un formato ligero de empaquetado de datos binarios para su transmisión. Es usado sobre protocolos de transporte como UDP y TCP por ejemplo en WS-Routing.

- DNS

Domain Name System. Base de datos distribuida, que permite relacionar nombres de máquinas con sus correspondientes números IP. Es la parte encargada de realizar la resolución de los nombres.

- DSOM

Distributed System Object Model o modelo de objeto de sistema distribuido. Es la versión SOM para soportar objetos distribuidos a través de la red.

- DTD

Document Type Definition (definición de tipo de documento). Uno de los mecanismos existentes que permite definir la estructura de un documento XML. Es usado también para validar la consistencia de mismo.

- ebXML

Electronic Business XML. Es un proyecto respaldado por las Naciones Unidas y OASIS para crear una infraestructura basada en XML para facilitar los negocios electrónicos. Más información en <http://www.ebxml.org>.

- EJB

Abreviatura de Enterprise JavaBeans. Es un API definida por Sun para crear objetos destinados a arquitecturas cliente-servidor multicapa, y permite abstraer al programador de las conexiones entre objetos distribuidos, bases de datos, etc... permitiéndole centrarse en la programación de la lógica empresarial.

- FTP

Es el acrónimo de File Transfer Protocol, o protocolo de transmisión de ficheros. Como su propio nombre indica es uno de los protocolos utilizados para la transmisión de ficheros entre dos máquinas, aunque ahora está tomando protagonismo HTTP (no se pensó para esto, pero es totalmente válido su uso). Una de las peculiaridades de este protocolo es que para transmitir un fichero abre dos conexiones TCP .

- GPG

GNU Privacy Guard, es el equivalente del PGP en software libre, ya que el algoritmo IDEA está patentado.

- HTTP

Hyper Text Transfer Protocol (RFC2616). Protocolo de transmisión de datos en modo texto. Es el protocolo utilizado en las consultas de páginas web.

- IDEA

Es el acrónimo de International Data Encryption Algorithm, data de 1992. Se trata de uno de los algoritmos simétricos más seguros de la actualidad. Usa mismo algoritmo para cifrar que para descifrar los datos.

- IP

Internet Protocol. Proporciona un protocolo no seguro de entrega de datos. Como no seguro se quiere indicar que no se tiene control sobre si el paquete ha llegado o se ha perdido por el camino, por lo que deberá ser otro protocolo el que se encargue de controlar este aspecto (por ejemplo TCP).

- J2EE

Java 2 Platform Enterprise Edition. Entorno basado en Java desarrollado por Sun, destinado a la programación, desarrollo e implantación de aplicaciones empresariales basadas principalmente en Web. Algunas de las características de este entorno son el uso de EJB, de JDBC y Java servlets.

- J2SE

Java 2 Platform Standar Edition.

- Jabber

Sirve tanto como protocolo de transporte como de protocolo de empaquetado de datos. Permite transmisiones no síncronas. Pese a no ser un estándar es bastante utilizado. Se puede obtener la especificación y el software en <http://www.jabber.org>.

- JAXM

Java API for XML Messaging. Proporciona el API para el envío de mensajes XML usando Java. Más información en <http://java.sun.com/xml/jaxm/index.html>.

- JAXR

Java API for XML Registries. Es el API de Java para el uso de registros como UDDI. Puede encontrar más datos en <http://java.sun.com/xml/jaxr/index.html>.

- JAX-RPC

Java API for XML RPC. Es la estandarización del API de Java para el uso de servicios web. Para más información: <http://java.sun.com/xml/jaxrpc.html>.

- JMS

Es la abreviatura de Java Message Service o servicio de mensajes de Java. Proporciona el API que puede ser utilizado por las aplicaciones Java que usen mensajes en su comunicación.

- JSP

Acronimo de Java Server Page. Tecnología de tipo servidor, son una extensión de la tecnología Java Servlets . Se trata de una tecnología que permite ínter operar código HTML con scripts que incluyen código Java para crear HTML dinámico. Antes de ser operativos, deben transformarse en servlets que puedan ser compilados.

- JXTA

Pronunciado “juxta”, deriva su nombre de la palabra *juxtapose*, que significa junto, lado a lado. JXTA es un conjunto de protocolos peer-to-peer capaz de integrar todo tipo de maquinas, desde un PDA hasta servidores o teléfonos móviles. Los protocolos hacen uso de mensajes XML.

- MD5

Algoritmo de generación de signatura diseñado por Ron Rivest. Genera firmas de 128 bits a partir de bloques de 512 bits.

- NASSL

Network Accessible Services Specification Language. Es una descripción de interfaz, junto con otras especificaciones ha desembocado en WSDL. Actualmente está obsoleto.

- OMG

Abreviatura de Object Management Group. Consorcio industrial de más de 700 miembros responsable de algunos estándares como CORBA. Su objetivo es crear un entorno estándar para la programación orientada a objetos.

- PERL

Practical Extraction and Report Language o lenguaje de extracción práctica y reportes. Desarrollado por Larry Wall fue diseñado para el manejo de cadenas de texto. Debido precisamente a esto y a ser multiplataforma se convirtió rápidamente en uno de los lenguajes más utilizados para el desarrollo de aplicaciones CGI. Actualmente se le podría considerar también la semilla de PARROT.

- PGP

Se trata de la abreviatura de Pretty Good Privacy (Privacidad bastante buena). Mecanismo de criptografía de clave pública originalmente creado por Phil Zimmermann. Como algoritmo de encriptación utiliza IDEA, MD5 y RSA (dependiendo de las versiones). Para más información <http://www.pgpi.org>.

- RDF

Resource Description Framework. Integra un conjunto de bibliotecas, catálogos, etc.. donde se recopilan noticias, eventos, fotos, etc y define un entorno que permite el intercambio de información a través de la red mediante XML.

- Reliable HTTP (HTTPr)

Nueva versión del protocolo HTTP propuesta por IBM en la que se propone añadir soporte de entrega segura al protocolo HTTP ya existente. Más información en:

<http://www-106.ibm.com/developerworks/webservices/library/ws-phtt>.

- RMI

Remote Method Invocation o invocación de métodos remotos. Se basa en CORBA, aunque es mucho más sencillo. Sólo puede ser usado entre objetos Java.

- RSA

Algoritmo de cifrado asimétrico muy fácil de implementar y que debe su nombre a sus tres inventores Ronald Rivest, Adi Shamir y Leonard Adleman.

- SAML

Security Assertions Markup Language (lenguaje de marcas de afirmación de seguridad), Lenguaje desarrollado bajo OASIS <http://www.oasis-open.org/committees/security/>.

- SCL

Service Contract Language es un documento XML para la descripción de servicios. Es la base de WSDL. Actualmente está obsoleto.

- SDL

Es la abreviatura de Service Description Language (lenguaje de descripción de servicio), realmente es idéntico a SCL, ya que éste nace con el cambio de nombre de SDL por SCL (en Julio de 2000). Actualmente está obsoleto.

- SHA-1

Desarrollado por NSA, es un algoritmo de cifrado libre y seguro. Genera firmas de 160 bits a partir de bloques de 512.

- SMIL

Es el acrónimo de Synchronized Multimedia Integration Language (Lenguaje de integración multimedia sincronizada), y permite integrar mediante lenguaje de marcas, distintos medios para formar presentaciones.

- SMTP

Simple Mail Transfer Protocol (RFC 821). Protocolo que es utilizado normalmente para transmisiones de correos electrónicos, aunque puede ser utilizado para otros menesteres como puede ser transportar mensajes SOAP.

- SOAP

Abreviatura de Simple Object Access Protocol. Es el lenguaje en el que se apoya la transmisión de los mensajes en los web services.

- SOAP Security Extensions

Bajo esta especificación se detalla el modo de transportar elementos seguros, tales como elementos cifrados, firmas digitales, certificados, etc..

- SOM

System Object Model o modelo de objeto de sistema, es una arquitectura desarrollada por IBM que permite compartir código binario entre aplicaciones. SOM es la implementación completa de CORBA. Su correspondiente arquitectura distribuida se denomina DSOM.

- SSL

Acrónimo de Secure Socket Layer, capa de conexiones seguras, es un protocolo diseñado por Netscape que se basa en el uso de claves públicas para la encriptación del dato transmitido. Por convenio los URL que usan SSL, se acceden mediante <https://> en lugar de <http://>.

- TCP

Transmission Control Protocol, protocolo de control de transmisión. TCP proporciona conexiones permanentes, orientadas a flujo (y no sólo a paquetes sueltos como UDP) y seguras entre dos aplicaciones. Como seguras se entiende que no se perderán paquetes durante la transmisión por tener un control de éstos.

- UBR

UDDI Business Registry, registro UDDI de negocios. Es un registro UDDI universal, con todos los negocios y servicios dados de alta. No se incluyen los registros UDDI privados o de pruebas.

- UDDI

Universal Description Discovery and Integration (descubrimiento universal de descripción e integración). Mediante este protocolo es posible buscar, encontrar e integrar servicios web. Rige la forma y acciones a realizar en un registro de información sobre servicios web. Está muy ligado a WSDL. Más información <http://www.uddi.org>

- UDP

Abreviatura de User Datagram Protocol. Protocolo de transmisión de datos basado en datagramas, en paquetes de información sin conexión permanente. Se distingue de TCP en que éste mantiene las conexiones de modo permanente (hasta que se corta la conexión).

- URI

Acrónimo de Uniform Resource Identifier. Pequeñas cadenas de texto (normalmente direcciones) para hacer referencias a recursos.

- URL

Es la abreviatura de Uniform Resource Locator. Término usualmente aplicado a los patrones URI más populares, como son [http](http://), [ftp](ftp://), [mailto](mailto://), etc.

- UUID

Universally Unique Identifier, identificador único universal. Como su propio nombre indica, identifica de forma unívoca un elemento. Se usan para distinguir elementos dentro de un registro UDDI.

- WSCL

Web Services Conversation Language. Cuando existen varios mensajes definidos en un proceso, especifica en el orden en el que tienen que ser enviados los mensajes.

- WSDL:

Lenguaje de descripción de servicios web (Web Service Description Language). Es el estándar adoptado por el W3C para la descripción. Reemplaza a antiguos lenguajes como NASSL de IBM y SDL de Microsoft. Su especificación puede encontrarse en: <http://www.w3.org/tr/wsdl>.

- WSFL

Web Services Flow Language. Permite la generación de procesos empresariales basándose en documentos WSDL. Fue creado por IBM. Dio lugar junto a XLANG a la especificación BPEL4WS. Para más información <http://www.ibm.com/developerWorks/webservices>.

- WSIL

Web Service Inspection Language (Lenguaje de inspección de servicios web). Representa un índice de los servicios web disponibles en cierta dirección.

- WS-Routing

Propuesta de Microsoft para definir los caminos que ha de seguir el mensaje en su transmisión, pudiéndose definir cero o varios intermediarios. Puede encontrarse más información en <http://msdn.microsoft.com/library/enus/dnsrvspec/html/ws-routing.asp>.

- WSTX

Es otra forma de denominar a WS-Transaction.

- XACML

Arónimo de eXtensible Access Control Markup Language (lenguaje extensible de etiquetas para el control de accesos). Proporciona la sintaxis en XML para el manejo de accesos a recursos. Mediante pocas líneas de código XML es posible controlar infraestructuras existentes como LDAP. Es un lenguaje estándar OASIS.

- XML Digital Signatures

Representación de las firmas digitales mediante el uso de XML. Es utilizado como elemento de seguridad dentro de la especificación WS-Security.

Puede encontrarse más información en <http://www.w3.org/Signature/>.

- XML Encryption

Define el modelo de encriptación de datos XML y la representación de estos como contenido de mensajes XML. Más información en <http://www.w3.org/Encryption/2001/>.

- XML-RPC

Inventado por Userland software, es un protocolo de comunicación hacia procesos remotos basado en XML, que hace patente la utilidad de SOAP. RPC es la abreviatura de Remote Procedure Call (llamada a procedimiento remoto). Se puede encontrar más información en: <http://www.xmlrpc.org>.

- XKMS

Es el acrónimo de XML Key Management Service (servicio de manejo de claves XML) y es la definición del modo en que se debe implementar una infraestructura basada en servicios web con claves públicas. La especificación se puede encontrar en <http://www.w3.org/tr/xkms>.

- XLANG

Lenguaje de programación de procesos empresariales, salido de la mano de Microsoft. Dio lugar junto a WSFL a la especificación BPEL4WS.

- XPATH

Lenguaje para realizar referencias a partes de un documento XML. Se diseñó para ser utilizado por XSLT y XPointer.

- XPointer

XML Pointer. Lenguaje que se usa para identificación de fragmentos de mensaje cuyo tipo es text/xml, application/xml, text/xml-external-parsed-entity, o application/xml-external-parsed-entity.

Mas datos en : <http://www.w3.org/XML/Linking> y <http://www.w3.org/TR/xptr/>

- XSL

XML Stylesheet Language (lenguaje de hojas de estilo para XML). Es un lenguaje para la representación de hojas de estilo para XML.

- XSLT

XML Stylesheet Language Transformation. Mediante XSLT, XSL define las transformaciones de un XML, en otro que use esta hoja de estilos.