Capítulo 3

Servicios Web

3.1 Introducción

Los servicios web permiten funciones software a través de Internet, permitiendo que programas como PHP, ASP, JSP, JavaBeans y otros muchos puedan hacer peticiones a programas (propiamente, los servicios web) que se estén ejecutando en una máquina remota y usar dicha respuesta en una página web, un servicio WAP o cualquier otra aplicación.

El concepto de servicio web comienza a extenderse impulsado por los grandes gigantes de la informática, como Sun, Oracle, HP, Microsoft e IBM. No aporta muchas novedades, pero facilita y simplifica el acceso a software a través de la red.

La estandarización de los servicios web se realiza a través de dos comités: OASIS y W3C. Para mejorar la interoperabilidad entre distintas implementaciones de servicios web surgió la organización WS-I, que ha desarrollado una serie de perfiles para definir mejor los estándares implicados.

3.1.1 Conceptos básicos

Muchas de las ideas detrás de los servicios web son asombrosamente simples y no son una novedad en el mundo de las redes e Internet:

- El proveedor de servicios web define un formato para las peticiones a un servicio y de las respuestas que generará.
- Un ordenador realiza una petición a través de una red
- El servicio web realiza la acción solicitada y devuelve la respuesta

Esta acción puede consistir en buscar un valor actual de la bolsa, encontrar el mejor precio para un producto determinado, guardar una reunión en una agenda, traducir un fragmento de texto a cualquier lenguaje o validar el número de una tarjeta de crédito.

La razón del repentino crecimiento de los servicios web es la incorporación de protocolos estándar para invocar servicios y transmitir datos, como son SOAP y WSDL, ambos basados en XML.

3.1.2 XML

En el pasado cada proveedor de un servicio tenía sus propios estándares, pero actualmente se impone el lenguaje extensible de marcas (eXtensive Markup Language, XML) sobre HTTP. Así se consigue un acceso simple y compatible con cualquier tecnología.

XML es un lenguaje de marcas que permite la representación de información estructurada. Un lenguaje de marcas permite identificar estructuras dentro de un documento. La especificación XML define un procedimiento estándar para añadir marcas a los documentos; de hecho XML es un meta-lenguaje para definir lenguajes de marcas. En otras palabras, XML ofrece unas reglas para definir etiquetas y relaciones estructurales entre ellas. Al no haber etiquetas predefinidas no existe una semántica a priori; toda la semántica de un documento XML debe ser definida bien por la aplicación que lo procesa o bien por hojas de estilo.

Un documento XML tiene dos estructuras, una lógica y otra física. Físicamente, el documento está compuesto por unidades llamadas entidades. Una entidad puede hacer referencia a otra entidad, causando que esta se incluya en el documento. Cada documento comienza con una entidad documento, también llamada raíz. Lógicamente, el documento está compuesto de declaraciones, elementos, comentarios, referencias a caracteres e instrucciones de procesamiento, todos los cuales están indicados por una marca explícita. Las estructuras lógica y física deben encajar de manera adecuada.

Los elementos de un documento XML constan de una etiqueta de apertura <*etiqueta*> y una de cierre <*/etiqueta*>. Entre ambas se encuentra el contenido del elemento. Si el elemento no tiene contenido también es válida la etiqueta <*etiqueta/*>.

```
<nombre>
      Fernando Damián
</nombre>
```

Un elemento también puede tener atributos, que son una forma de incorporar características o propiedades al elemento en cuestión. Los atributos están incluidos dentro de la etiqueta de apertura.

```
<aviso
 tipo="emergencia"
 gravedad="mortal">
      Que no cunda el pánico
</aviso>
```

Las distintas combinaciones de atributos y contenido del elemento son distintas formas de representar un mismo concepto y quedan a la discreción del programador.

```
<aviso
 tipo="emergencia"
 gravedad="mortal">
      Que no cunda el pánico
</aviso>
```

```
<aviso>
      <tipo>emergencia</tipo>
      <gravedad>mortal</gravedad>
      <texto>Que no cunda el pánico</texto>
</aviso>

<aviso
 tipo="emergencia"
 gravedad="mortal"
 texto="que no cunda el pánico"/>
```

Los documentos XML deben seguir una estructura estrictamente jerárquica en lo que respecta a las etiquetas que delimitan sus elementos. Una etiqueta debe estar correctamente "incluida" en otra. Así mismo, los elementos con contenido, deben estar correctamente "cerrados"

Un aspecto esencial de los servicios web es transformar una representación XML genérica en una representación específica para la aplicación.

3.1.2.1 Analizadores XML

Los analizadores XML son un elemento clave en los servicios web, ya que se encargan de interpretar los datos contenidos en el documento y hacerlos accesibles a la aplicación. En dispositivos con baja capacidad de procesamiento son un factor decisivo, ya que suelen tener un gran tamaño y consumen mucha memoria.

Existen tres tipos principales de analizadores XML, cada uno con sus ventajas e inconvenientes:

- **Analizador de modelo**: lee el documento XML completamente y crea en la memoria una representación estructurada de los datos. Su gran ventaja es que el análisis se realiza al principio y luego se pueden tratar los datos como una estructura cualquiera. Su desventaja principal es el alto consumo de memoria.

- **Analizador push**: Se define un conjunto de eventos que pueden aparecer en un documento XML, como pueden ser el comienzo de una etiqueta o la presencia de datos de texto, y al encontrar uno de estos eventos llama a un método que lo procese adecuadamente. El analizador SAX, usado por JSR-172, es de este tipo.

- **Analizador pull**: El documento XML se va recorriendo poco a poco mediante un método que pide el siguiente elemento. El analizador kXML, usado por kSOAP, es de este tipo.

3.1.3 SOAP

El protocolo de acceso a objetos simples (Simple Object Access Protocol, SOAP) en un estándar de W3C que define el formato de las peticiones para los servicios web.

Los mensajes SOAP son enviados entre los dos extremos en los llamados sobres SOAP para hacer una petición o enviar una respuesta. Estos sobres están formateados en XML y son muy fáciles de decodificar.

Actualmente un sin fin de empresas se han decantado por el desarrollo de aplicaciones que puedan trabajar sobre Internet porque permite la distribución global de la información. Las tecnologías más usadas para el desarrollo de estas aplicaciones, han sido hasta hace poco CORBA, COM y EJB. Cada una proporciona un marco de trabajo basado en la activación de objetos remotos mediante la solicitud de ejecución de servicios de aplicación a un servidor de aplicaciones.

Estas tecnologías han demostrado ser muy efectivas para el establecimiento de sitios Web, sin embargo presentan una serie de desventajas, como son total incompatibilidad e interoperabilidad entre ellas, total dependencia de la máquina servidora sobre la que corren, así como el lenguaje de programación.

Esto ha llevado a la necesidad de considerar un nuevo modelo de computación distribuida de objetos que no sea dependiente de plataformas, modelos de desarrollo ni lenguajes de programación. Por todos estos motivos surge el concepto de SOAP (Simple Object Access Protocol).

3.1.3.1 Concepto de SOAP

La funcionalidad que aporta SOAP es la de proporcionar un mecanismo simple y ligero de intercambio de información entre dos puntos usando el lenguaje XML. SOAP no es más que un mecanismo sencillo de expresar la información mediante un modelo de empaquetado de datos modular y una serie de mecanismos de codificación de datos. Esto permite que SOAP sea utilizado en un amplio rango de servidores de aplicaciones que trabajen mediante el modelo de comunicación RPC (Remote Procedure Call).

SOAP consta de tres partes:

- El ***SOAP envelope*** que define el marco de trabajo que determina qué se puede introducir en un mensaje, quién debería hacerlo y si esa operación es opcional u obligatoria.

- Las **reglas de codificación SOAP** que definen el mecanismo de serialización que será usado para encapsular en los mensajes los distintos tipos de datos.

- La representación **SOAP RPC** que define un modo de funcionamiento a la hora de realizar llamadas a procedimientos remotos y la obtención de sus resultados.

3.1.3.2 Objetivos de SOAP

A la hora de realizar el diseño de SOAP se han tenido en cuenta una serie de consideraciones con el fin de cumplir una serie de objetivos claros, objetivos que le darán el potencial que reside en SOAP y que le harán tan atractivo. Estos objetivos son:

- **Establecer un protocolo estándar** de invocación a servicios remotos que esté basado en protocolos estándares de uso frecuente en Internet, como son HTTP (Hiper Text Transport Protocol) para la transmisión y XML (eXtensible Markup Language) para la codificación de los datos.

- **Independencia** de plataforma hardware, lenguaje de programación e implementación del servicio Web.

El logro de estos objetivos ha hecho de SOAP un protocolo extremadamente útil, ya que el protocolo de comunicación HTTP es el empleado para la conexión sobre Internet, por lo que se garantiza que cualquier cliente con un navegador estándar pueda conectarse con un servidor remoto, además los datos en la transmisión se empaquetan o serializan con el lenguaje XML, que se ha convertido en algo imprescindible en el intercambio de datos ya que es capaz de salvar las incompatibilidades que existían en el resto de protocolos de representación de datos de la red.

Por otra parte, los servidores Web pueden procesar las peticiones de usuario empleando tecnologías tales como Servlets, páginas ASP (Active Server Pages), páginas JSP (Java Server Pages) o sencillamente un servidor de aplicaciones con invocación de objetos mediante CORBA, COM o EJB.

La especificación SOAP (SOAP Specification 1.1) indica que las aplicaciones deben ser independientes del lenguaje de desarrollo, por lo que las aplicaciones cliente y servidor pueden estar escritas con HTML, DHTML, Java, Visual Basic o cualquier otra herramienta o lenguaje disponibles.

3.1.3.3 Un ejemplo sencillo de mensajes SOAP

Supongamos un servicio web que permita comprobar si un código postal es válido y pertenece realmente al país especificado. Este servicio web sería muy útil para asegurar la validez en un formulario de una página web. El código relativo a la petición SOAP sería:

```
<env:Envelope
 xmlns:env="http://www.w3.org/2001/06/soap-envelope">
      <env:Body>
         <m:ValidatePostcode
          env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
          xmlns:m="http://www.somesite.com/Postcode">

                <Postcode>
                     WC1A8GH
                </Postcode>

                <Country>
                     UK
                </Country>

         </m:ValidatePostcode>
      </env:Body>
</env:Envelope>
```

Esta petición tiene dos parámetros *(postcode y country)* contenidos en un elemento llamado *ValidatePostcode*, que sería el nombre del servicio web al que estamos haciendo la petición. El resto de los datos del sobre, como la versión de SOAP y la codificación del texto, ayuda al servicio web a procesar la petición.

La respuesta al mensaje anterior tendría el siguiente formato:

```
<env:Envelope
 xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
      <env:Body>
         <m:ValidatePostcodeResponse
          env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
          xmlns:m="http://www.somesite.com/Postcode">
               <Valid>
                     Yes
               </Valid>
         </m:ValidatePostcodeResponse>
      </env:Body>
</env:Envelope>
```

El elemento *ValidatePostcodeResponse* contesta al elemento *ValidatePostcode* de la petición, conteniendo un único elemento, *Valid*, que indica si el código postal introducido es válido o no.

3.1.3.4 Partes de un mensaje SOAP

Un mensaje SOAP no es más que un documento en formato XML que está constituido por tres partes bien definidas que son: el *SOAP envelope*, el *SOAP header* de carácter opcional y el *SOAP body*. Cada uno de estos elementos contiene lo siguiente:

- El ***envelope*** es el elemento más importante y de mayor jerarquía dentro del documento XML y representa al mensaje que lleva almacenado dicho documento.

- El ***header*** es un mecanismo genérico que se utiliza para añadir características adicionales al mensaje SOAP. El modo en la que se añadan cada uno de los campos dependerá exclusivamente del servicio implementado entre cliente y servidor, de forma que cliente y servidor deberán estar de acuerdo con la jerarquía con la que se hayan añadido los distintos campos. De esta forma será sencillo separar entre sí los distintos datos a transmitir dentro del mensaje.

  Un ejemplo de uso del *header*, donde se indica un cierto parámetro útil para el servicio Web que le indica como debe procesar el mensaje sería:

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

<SOAP-Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/"/>

  <SOAP-ENV:Header>
    <t:Transaction
     xmlns:t="some-URI"
     SOAP-ENV:mustUnderstand="1">
        5
    </t:Transaction>
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    <getQuote
      xmlns="http://namespaces.cafeconleche.org/xmljava/ch2/">
      <symbol>
         RHAT
      </symbol>
    </getQuote>
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

- El ***body*** es un contenedor de información en el cual se almacenarán los datos que se quieran transmitir de lado a lado de la comunicación. Dentro de este campo, SOAP define un elemento de uso opcional denominado *Fault* utilizado en los mensajes de respuesta para indicar al cliente algún error ocurrido en el servidor. Un ejemplo de uso de este nuevo elemento sería el siguiente, en el que se ha detectado un error en la aplicación que corre sobre el servidor que provoca que no opere convenientemente, por lo que se le indica al cliente:

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

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" >
 SOAP-ENV:encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/"/>
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
```

```
      <faultcode>
           SOAP-ENV:Server
      </faultcode>

      <faultstring>
           Server Error
      </faultstring>

      <detail>
           <e:myfaultdetails
            xmlns:e="Some-URI">

             <message>
                  My application didn't work
             </message>

             <errorcode>
                  1001
             </errorcode>

          </e:myfaultdetails>
    </detail>

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

El atributo *encodingStyle* se utiliza para indicar las reglas de serialización utilizadas en el mensaje SOAP. No existe un formato de codificación por defecto, sino que existen una serie de posibles formatos a utilizar. El valor de este atributo es una lista ordenada de una o más URIs que identifican la regla o reglas de serialización que pueden ser utilizadas en el mensaje, en el orden en el que se han de aplicar. De entre todas las posibles, las más utilizadas son: ***http://schemas.xmlsoap.org/soap/encoding/*** y ***http://my.host/encoding/*** **.**

Todo mensaje SOAP debe tener un elemento ***Envelope*** asociado a la dirección de red ***http://schemas.xmlsoap.org/soap/envelope/*****.** Si un mensaje recibido por una aplicación SOAP contiene en este elemento un valor distinto al anterior la aplicación trataría dicho mensaje como erróneo.

3.1.4 WSDL

WSDL (Web Services Definition Language) es un lenguaje basado en XML que usamos para describir un servicio web. Al publicar nuestro servicio es recomendable publicar conjuntamente un archivo WSDL que indique los métodos, dirección y estructura de éste. Cuando un cliente quiera consumir nuestro servicio bastará con que interprete el contenido del archivo WSDL. Esto proporciona una gran flexibilidad, ya que es posible modificar el servicio sin que el cliente se vea afectado.

3.1.4.1 Elementos de un documento WSDL

- ***<definitions>:*** Dentro de éste elemento se definirán los servicios asociados al documento. Generalmente cada documento WSDL describe un sólo servicio, y suele seguir la siguiente estructura:

```
<definitions>
  <types>
     Definición de tipos........
  </types>

  <portType>
    Definición de operaciones...
  </portType>

  <message>
    Definición de mensaje....
  </message>

  <binding>
    Definición de invocación....
  </binding>

  <service>
    Ubicación del servicio...
  </service>
</definitions>
```

- ***<types>:*** Define tipos de datos. Ejemplo:

```
<types>
  <complexType
   name="request">
      <sequence>

        <element
         name="String_1"
         type="string"/>

        <element
         name="String_2"
         type="string"/>

      </sequence>
  </complexType>
</types>
```

Aquí se define un tipo complejo llamado *request* formado por dos cadenas de caracteres, *String_1* y *String_2*.

- ***<portType>:*** Define las operaciones que realiza el servicio y los mensajes correspondientes. Ejemplo:

```
<portType
 name="Interface">
    <operation
     name="request">

       <input
        message="tns:Interface_request"/>

       <output
        message="tns:Interface_requestResponse"/>

  </operation>
</portType>
```

Aquí define la operación *request* dentro del puerto *Interface* cuyo mensaje de entrada es *Interface_request* y el de salida es *Interface_requestResponse*.

- **<*message*>:** Define los tipos de datos de cada mensaje. Ejemplo:

```
<message
 name="Interface_request">
   <part
    name="parameters"
    element="ns2:request"/>
</message>
```

Aquí define el mensaje *Interface_request*, cuyo parámetro es un elemento del tipo *request*, anteriormente definido en <*types*>.

- **<*binding*>:** Define el formato y detalles del protocolo para cada operación. Ejemplo:

```
<binding
 name="InterfaceBinding"
 type="tns:Interface">
    <soap:binding
     transport="http://schemas.xmlsoap.org/soap/http"
     style="document"/>

  <operation
   name="request">

    <soap:operation
     soapAction=""/>

      <input>
        <soap:body
         use="literal"/>
      </input>

      <output>
        <soap:body
         use="literal"/>
      </output>

  </operation>
</binding>
```

Aquí se especifica que los mensajes van a ser *document/literal*, que es el único soportado por JSR-172. El significado de estos parámetros se aclara en el siguiente apartado.

- ***<service>:*** Define la ubicación del servicio. Ejemplo:

```
<service
 name="Serverscript">

    <port
     name="InterfacePort"
     binding="tns:InterfaceBinding">
        <soap:address
         location="http://localhost:8080/server/>
    </port>
</service>
```

Se indica que el servicio estará situado en la dirección *http://localhost:8080/server*.

3.1.4.2 Estilo y uso de un documento WSDL

Un archivo WSDL puede tener un estilo *RPC* o un estilo documento (*document*), que afectará al formato de los datos en dicho archivo y en los mensajes SOAP que se intercambien.

Un archivo WSDL con estilo *RPC* tendría la siguiente forma:

```
<message
 name="myMethodRequest">

    <part
     name="x"
     type="xsd:int"/>

    <part
     name="y"
     type="xsd:float"/>

</message>

<message
 name="empty"/>

<portType
 name="PT">
    <operation
     name="myMethod">

        <input
         message="myMethodRequest"/>

        <output
         message="empty"/>

    </operation>
</portType>
```

Define dos mensajes: El primero de petición, llamado *myMethodRequest* que recibe dos parámetros númericos, uno entero y otro flotante. El segundo mensaje está vacío. Por último especifica la operación *myMethod* que recibe el mensaje *myMethodRequest* y devuelve el mensaje vacío *empty*. Este mismo ejemplo en estilo documento sería así:

```
<types>
    <schema>
        <element
         name="xElement"
         type="xsd:int"/>

        <element
         name="yElement"
         type="xsd:float"/>
    </schema>
</types>

<message
 name="myMethodRequest">
    <part
     name="x"
     element="xElement"/>

    <part
     name="y"
     element="yElement"/>
</message>

<message
 name="empty"/>

<portType
 name="PT">
    <operation
     name="myMethod">
        <input
         message="myMethodRequest"/>

        <output
         message="empty"/>
    </operation>
</portType>
```

En este caso se crea un esquema con los elementos *xElement* e *yElement* de tipos entero y flotante respectivamente, y en el mensaje se especifican estos elementos en lugar de los tipos. RPC tiene la ventaja de ser más simple, pero el estilo documento es más correcto, ya que el archivo WSDL podría validarse con un analizador XML y además cumple con las especificaciones de WS-I.

Otro factor que afecta a los mensajes SOAP del servicio es el uso, que puede ser codificado (*encoded*) o literal. Un mensaje SOAP *RPC/encoded* tendría la siguiente forma:

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x
             xsi:type="xsd:int">
                  5
            </x>

            <y
             xsi:type="xsd:float">
                  5.0
            </y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

Como se puede apreciar en el propio mensaje se identifica el tipo del parámetro. Esto hace al mensaje más legible, aunque en realidad no aporta información ya que el que recibe el mensaje ya sabe el tipo de los parámetros, por lo que es poco eficiente. Además no es compatible con WS-I.

Un mensaje *RPC/literal* sería así:

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x>
                  5
            </x>

            <y>
                  5.0
            </y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

En este caso no se envía el tipo en el mensaje lo que mejora la eficiencia, además este modo sí es compatible con WS-I. Sin embargo sigue sin poder validarse con facilidad.

Un mensaje *document/literal* tendría este aspecto

```
<soap:envelope>
    <soap:body>
        <xElement>
             5
        </xElement>

        <yElement>
             5.0
        </yElement>
    </soap:body>
</soap:envelope>
```

Este modo sí es validado fácilmente, y es compatible con WS-I siempre que el elemento *soap:body* tenga un sólo hijo. Eso provoca un problema, ya que el mensaje anterior tiene dos hijos, y además al perder el nombre de la operación, en ocasiones es imposible distinguir a qué método remoto está dirigido el mensaje.

Para evitar los problemas de cada tipo Microsoft creó el modo *document/literal wrapped*, que no está definido en ningún sitio. Este modo complica mucho el archivo WSDL, que tendría el siguiente aspecto:

```
<types>
    <schema>
        <element
         name="myMethod">
            <complexType>
                <sequence>
                    <element
                     name="x"
                     type="xsd:int"/>
                    <element
                     name="y"
                     type="xsd:float"/>
                </sequence>
            </complexType>
        </element>
        <element
         name="myMethodResponse">
            <complexType/>
        </element>
    </schema>
</types>

<message
 name="myMethodRequest">
    <part
     name="parameters"
     element="myMethod"/>
</message>

<message
 name="empty">
    <part
     name="parameters"
     element="myMethodResponse"/>
</message>

<portType
 name="PT">
    <operation
     name="myMethod">

        <input
         message="myMethodRequest"/>

        <output
         message="empty"/>
    </operation>
</portType>
```

En este caso los elementos que se definen en el esquema no son los parámetros por separado, sino el conjunto de parámetros para cada mensaje. Ahora, el mensaje SOAP sí contendrá un elemento con el nombre de la operación

```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x>
                5
            </x>

            <y>
                5.0
            </y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

La diferencia entre *document* y *wrapped* podría verse de la siguiente forma: el mensaje *document* se mapearía en un método así:

```
public void method (myMethod m);
```

Es decir, *myMethod* se considera como un objeto que contiene dos elementos. En cambio, el mensaje *wrapped* sería:

```
public void myMethod (int x, int y);
```

3.1.4.3 Generación del documento WSDL

La mayoría de aplicaciones de desarrollo de servicios web cuentan con una herramienta de generación automática del archivo WSDL, por ejemplo, *Java2WSDL*.

Si ejecutamos ésta aplicación desde la línea de comandos basta pasar como parámetro la clase principal de nuestro servicio y obtenemos un documento WSDL perfectamente viable, pudiendo especificar nombres, métodos a incluir, etc.

Para el desarrollo de éste proyecto se ha usado el contenedor de servicios web XML Axis para la creación del documento WSDL.

3.1.4.4 Interpretación del documento WSDL

El esquema general de un servicio web se muestra en la siguiente figura:

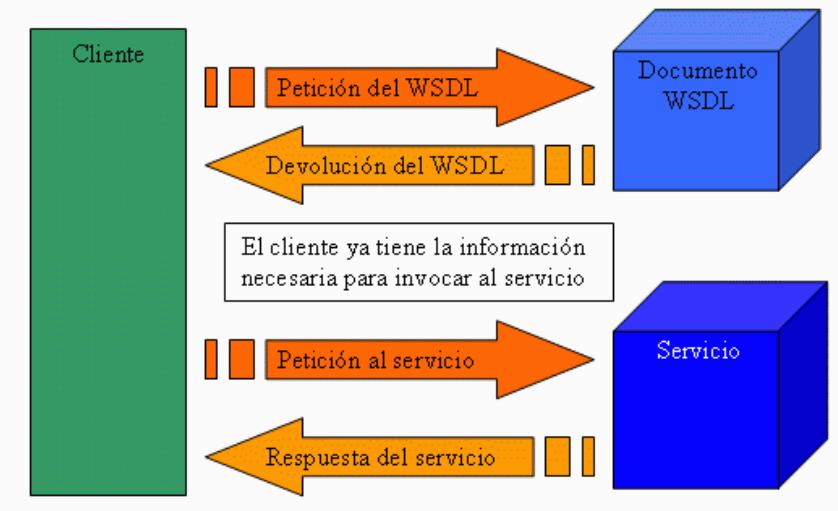

Figura 3.1: Esquema de un servicio web

El cliente obtendría el documento WSDL, generalmente de la dirección del servicio o por medio del descubrimiento UDDI, y a partir de ahí se comunicaría con el servicio mediante mensajes SOAP. Sin embargo los dispositivos móviles no tienen la capacidad suficiente como para interpretar el documento WSDL, por lo que el esquema de servicio web para móviles es ligeramente distinto. En el caso de kSOAP se debe tener en cuenta el formato de la comunicación a nivel de código, ya que la petición SOAP se crea añadiendo los parámetros necesarios a un objeto SOAP y enviándolo luego al servidor.

La aproximación kSOAP pierde mucha flexibilidad, ya que sería necesario retocar el código en caso de pequeños cambios en el servicio. El caso de JSR 172 el diferente y sigue el siguiente esquema:

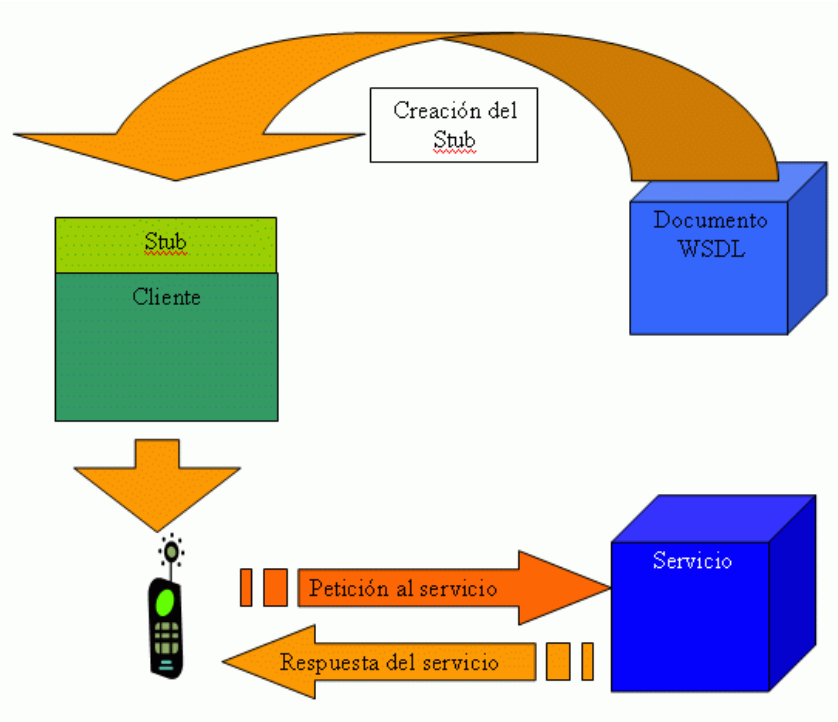

Figura 3.2: Esquema de un servicio web JSR 172

En este caso antes de incorporar la aplicación al dispositivo móvil es necesario obtener el stub a partir del documento WSDL. Una vez creado, el cliente puede llamar al servicio web a través del stub, así el formato de la comunicación es totalmente transparente, ya que basta con invocar un método del stub para que se realice la petición y se reciba la respuesta.

La creación del stub a partir del documento WSDL es muy sencilla usando el Wireless ToolKit. Una vez abierto nuestro proyecto seleccionamos *Proyectos>Stub generator*:

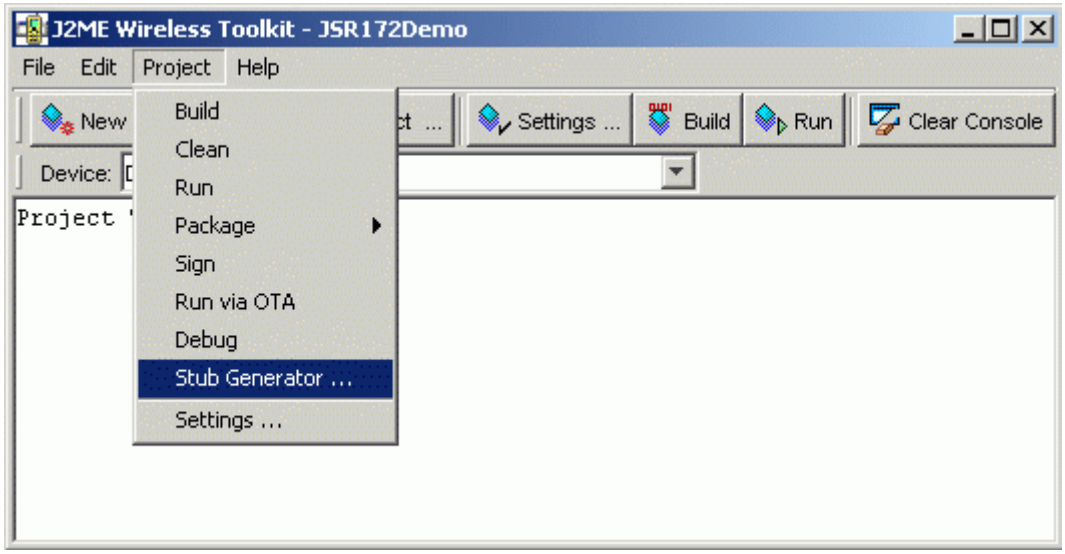

Figura 3.3: Generación del stub en WTK

Se nos pedirá la ruta al archivo WSDL y el paquete donde queremos poner las clases. Una vez terminado podemos compilar el stub y usarlo junto con el cliente.

3.2 kSOAP

El uso de SOAP para transferir datos en lugar de otras tecnologías tiene sus ventajas. En primer lugar SOAP define más de 40 tipos de datos estándar a transmitir mediante lenguaje XML, en segundo lugar permite distintos esquemas de comunicación como pueden ser llamadas a procedimientos remotos (RPC), mensajes asíncronos, multicast, etc, y en último lugar debido a que SOAP ha ganado tanta popularidad en los servicios Web otros muchos protocolos han enfocado sus esfuerzos a poder interactuar con éste, es el caso de WSDL, UDDI, etc.

3.2.1 kSOAP parsers

Una vez que se tenga un mensaje SOAP se pueden extraer de él los distintos datos que tenga almacenados mediante un parser XML, pero será más cómodo hacerlo mediante un parser SOAP ya que, con el primero habría que extraer la información en forma de texto para después pasarla a un objeto Java que la contenga, mientras que con el parser SOAP la extracción de datos es directa.

Un parser SOAP se construye sobre un parser genérico XML al cual se le añade una serie de mecanismos específicos con los cuales realizar el "mapeo" de datos. Dicho parser SOAP es capaz de comprender los tipos de datos de la información almacenada en el mensaje XML y de forma automática convertir texto en objetos Java. La ventaja de esto es que para el programador existe una total transparencia a la hora de programar entre el programa Java que envía y recibe datos y el propio mensaje XML. El programador solo le pasa los datos a un *SOAP writer* y se queda a la espera de la respuesta, que se la devuelve un *SOAP parser*.

Un ejemplo de todo esto puede verse en el siguiente mensaje SOAP en el que se ha almacenado un String que contenga el típico ejemplo *"Hola Mundo"*.

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>

        <message
         xsi:type="xsd:string">
            Hello World
        </message>

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

Si a este mensaje SOAP le pasamos un parser SOAP podremos obtener fácilmente el String almacenado, supongamos que el mensaje está en el string *"mesg"*:

```
ByteArrayInputStream bis = new ByteArrayInputStream (mesg.getBytes ());
InputStreamReader reader = new InputStreamReader (bis);
XmlParser xp = new XmlParser (reader);

// Se realiza un mapeo directo entre objetos Java y elementos Soap
SoapEnvelope envelope = new SoapEnvelope (new ClassMap (Soap.VER12));
envelope.parse (xp);

// Obtenemos el texto almacenado
String result = (String) envelope.getBody();
```

En un mensaje SOAP el atributo *xsi:type* especifica el tipo de dato de un elemento del mensaje, por ejemplo *<midato xsi:type="xsd:int">123</ midato >* especifica un entero de valor 123, en cambio *< midato xsi:type="xsd:string">123 </ midato >* especifica un string de valor "123". Es decir kSOAP es capaz de mapear de forma directa ciertos tipos de datos Java, concretamente:

Tipo SOAP	*Tipo Java*
xsd:int	java.lang.Integer
xsd:long	java.lang.Long
xsd:string	java.lang.String
xsd:boolean	java.lang.Boolean

A la hora de obtener los distintos datos almacenados en un mensaje SOAP el *kSOAP parser* lee elemento a elemento cado uno de los elementos XML que hay en el mensaje y que contienen un dato y realiza el mapeo a un objeto Java de acuerdo con las siguientes reglas:

1. Si el elemento SOAP es uno de los elementos primitivos indicados en la tabla anterior el mapeo se hace directamente.

2. Si el elemento es un tipo primitivo pero no es uno de los de la tabla anterior, lo convierte a un objeto *SoapPrimitive* del cual se puede obtener información mediante los métodos *SoapPrimitive.getNamespace()* y *SoapPrimitive.getName()* y del cual se puede obtener su valor mediante el método *SoapPrimitive.toString().*

3. Si el elemento SOAP es un tipo complejo con varios subcampos, se convierte a un objeto de tipo *KvmSerializable*, concretamente al objeto *SoapObject* que pertenece a la interfaz *KvmSerializable*, y de este *SoapObject* se obtienen la información con los métodos *SoapObject.getNamespace()* y *SoapObject.getName().*

3.2.2 Composición/Descomposición de un mensaje SOAP

Supongamos que queremos montar un mensaje XML mediante los métodos que facilita SOAP, para ello utilizaremos una serie de métodos que montarán un mensaje que después almacenaremos en un *String* y que posteriormente podrá ser transmitido por la red. Supongamos que el mensaje que queremos montar es el siguiente:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENC="http://www.w3.org/2001/12/soap-encoding"
 xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body
     SOAP-ENV:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
        <StockOrderParameters
         id="o0"
         SOAP-ENC:root="1">

            <Symbol
             xsi:type="xsd:string">
                 XYZ
            </Symbol>

            <From
             xsi:type="xsd:string">
                 Michael Yuan
            </From>

            <Shares
             xsi:type="xsd:int">
                 1000
            </Shares>

            <Buy
             xsi:type="xsd:boolean">
                 true
            </Buy>

            <LimitPrice
             xsi:type="xsd:float">
                 123.45
```

```
            </LimitPrice>

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

Para montar todo este mensaje sencillamente tenemos que usar una serie de métodos Java proporcionados por la librería de *kSOAP* de forma que el código obtenido sería:

```
// Creamos el objetos SOAP que almacene el mensaje
SoapObject method = new SoapObject("", "StockOrderParameters");

//Añadímos cada uno de los parámetros
method.addProperty("Symbol", "XYZ");
method.addProperty("From", "Michael Yuan");
method.addProperty("Shares", new Integer (1000));
method.addProperty("Buy", new Boolean (true));
method.addProperty("LimitPrice", new SoapPrimitive
      ("http://www.w3.org/2001/XMLSchema", "float", "123.4"));

// Ensamblamos el mensaje pasándolo por un SoapEnvelope
// Después lo almacenamos en un String
ByteArrayOutputStream bos = new ByteArrayOutputStream ();
XmlWriter xw = new XmlWriter (new OutputStreamWriter (bos));

// En este caso el mapeo es directo
SoapEnvelope envelope = new SoapEnvelope (new ClassMap (Soap.VER12));
envelope.setBody (method);
envelope.write (xw);
xw.flush ();
bos.write ('\r');
bos.write ('\n');
byte [] requestData = bos.toByteArray ();
String requestSOAPmesg = String (requestData);
```

Supongamos ahora el caso contrario, queremos extraer de un mensaje SOAP, que nos ha llegado como respuesta a una petición anterior, una serie de parámetros. Si el mensaje SOAP es el siguiente:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>
        <result>

            <OrderStatus>

                <CustomerName
                 xsi:type="xsd:string">
                     Michael Yuan
                </CustomerName>

                <Symbol
                 xsi:type="xsd:string">
                     XYZ
                </Symbol>

                <Share
                 xsi:type="xsd:int">
                     1000
                </Share>
```

```
                <Buy
                 xsi:type="xsd:boolean">
                     true
                </Buy>

                <Price
                 xsi:type="xsd:float">
                     123.45
                </Price>

                <ExecTime
                 xsi:type="xsd:dateTime">
                     2002-07-18T23:20:50.52Z
                </ExecTime>
            </OrderStatus>

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

Para extraer de este mensaje todos los parámetros, suponiendo que todo el mensaje está en el *String soapRespMesg,* el código que habría que ejecutar es:

```
ByteArrayInputStream bis = new ByteArrayInputStream
      (soapRespMesg.getBytes ());

InputStreamReader reader = new InputStreamReader (bis);
XmlParser xp = new XmlParser (reader);

// El mapeo en este caso es directo
SoapEnvelope envelope = new SoapEnvelope (new ClassMap (Soap.VER12));
envelope.parse (xp);

// Obtenemos la estructura de datos.
SoapObject orderStatus = (SoapObject) envelope.getResult();

// Pasamos los datos a su correspondiente tipo Java
String customerName = (String) orderStatus.getProperty ("CustomerName");
String symbol = (String) orderStatus.getProperty ("Symbol");
Integer share = (Integer) orderStatus.getProperty ("Share");
Boolean buy = (Boolean) orderStatus.getProperty ("Buy");

// Puesto que MIDP no tiene tipo "Float" no existe correspondencia
// entre el tipo objeto Java y "xsd:float" tipo SOAP. Por lo que
// este elemento es mapeado a un objeto "SoapPrimitive".
SoapPrimitive price = (SoapPrimitive) orderStatus.getProperty ("Price");
SoapPrimitive execTime = (SoapPrimitive) orderStatus.getProperty ("ExecTime");
```

3.2.3 Datos complejos en un mensaje SOAP

La importancia de SOAP reside en la facilidad que ofrece de poder representar datos de tipos con estructuras complejas, como pueden ser *arrays*, o incluso estructuras de datos que nosotros mismos definamos.

De esta forma, si queremos incluir en un mensaje SOAP un *array* con una serie de elementos, podemos hacerlo sin más problemas. Supongamos que un servicio Web realiza una serie de operaciones que le hemos solicitado y nos devuelve el siguiente mensaje SOAP en el que ha incluido un *array:*

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"
```

```
xmlns:SOAP-ENC="http://www.w3.org/2001/12/soap-encoding"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:n="http://www.javaworld.com/ksoap/test">
   <SOAP-ENV:Body>
     <result>
          <OrderStatus
           xsi:type="n:orderStatus">

              <CustomerName
               xsi:type="xsd:string">
                   Michael Yuan
              </CustomerName>

              <Transactions
               xsi:type="SOAP-ENC:Array"
               SOAP-ENC:arrayType="n:transaction[2]">

                  <Transaction
                   xsi:type="n:transaction">

                     <Symbol
                      xsi:type="xsd:string">
                            ABC
                     </Symbol>

                     <Share
                      xsi:type="xsd:int">
                            500
                     </Share>

                     <Buy
                      xsi:type="xsd:boolean">
                            true
                     </Buy>

                     <Price
                      xsi:type="xsd:float">
                            43.21
                     </Price>
                  </Transaction>

                  <Transaction
                   xsi:type="n:transaction">

                     <Symbol
                      xsi:type="xsd:string">
                            XYZ
                     </Symbol>

                     <Share
                      xsi:type="xsd:int">
                            1000
                     </Share>

                     <Buy
                      xsi:type="xsd:boolean">
                            true
                     </Buy>

                     <Price
                      xsi:type="xsd:float">
                            123.45
                     </Price>
                  </Transaction>

              </Transactions>
          </OrderStatus>
```

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

Al recibirse este mensaje, kSOAP lee el *array* y lo introduce en un objeto de tipo *java.util.Vector* y mediante el método *Vector.elementAt(i)* extrae de dicho *array* el elemento i-ésimo de entre todos los elementos que lo componen. Dependiendo del tipo del *arrayType* este objeto será almacenado en un SoapObject, en un SoapPrimitive, en un tipo por defecto Java, etc. Si suponemos que todo este mensaje SOAP se encuentra dentro del *String arraySoapRespMesg*, el código que realizaría la extracción del *array* sería:

```
ByteArrayInputStream bis = new ByteArrayInputStream
      (arraySoapRespMesg.getBytes ());
InputStreamReader reader = new InputStreamReader (bis);
XmlParser xp = new XmlParser (reader);
SoapEnvelope envelope = new SoapEnvelope (new ClassMap (Soap.VER12));
envelope.parse (xp);
SoapObject orderStatus = (SoapObject) envelope.getResult();

String customerName = (String) orderStatus.getProperty ("CustomerName");
Vector transactions = (Vector) orderStatus.getProperty ("Transactions");

// Primer elemento del array
SoapObject transaction0 = (SoapObject) transactions.elementAt(0);
String symbol0 = (String) transaction0.getProperty ("Symbol");
Integer share0 = (Integer) transaction0.getProperty ("Share");
Boolean buy0 = (Boolean) transaction0.getProperty ("Buy");
SoapPrimitive price0 = (SoapPrimitive) transaction0.getProperty ("Price");

// Segundo elemento del array
SoapObject transaction1 = (SoapObject) transactions.elementAt(1);
String symbol1 = (String) transaction1.getProperty ("Symbol");
Integer share1 = (Integer) transaction1.getProperty ("Share");
Boolean buy1 = (Boolean) transaction1.getProperty ("Buy");
SoapPrimitive price1 = (SoapPrimitive) transaction1.getProperty ("Price");
```

3.2.4 Protocolo HTTP mediante SOAP

El gran potencial que tiene SOAP a la hora de serializar datos para transmitirlos a través de una red se complementa con una funcionalidades que hacen de SOAP extremadamente útil, que es el uso del protocolo HTTP de forma directa, sin tener que controlarlo explícitamente sino dejando esta tarea en manos del API que implementa SOAP.

kSOAP cuenta con la clase *HttpTransport* que es la que aporta la funcionalidad de envío y recepción de mensajes SOAP vía HTTP. La forma de conexión que ofrece es una *llamada a procedimiento remoto* (RPC), tal que el método *HttpTransport.call()* toma como entrada un objeto *KvmSerializable*, lo serializa introduciéndolo en un mensaje SOAP completo que él mismo monta, envía dicho mensaje al servicio Web cuya dirección se haya indicado y recibe el mensaje respuesta que venga de vuelta. Tras esto pasa el mensaje recibido por un parser SOAP, llama al método *SoapEnvelope.getResult()* para que le de el objeto Java correspondiente y devuelve como resultado este objeto. Y todo esto de forma automática.

La forma de utilizar este método es muy sencilla, tan solo habría que ejecutar el siguiente código:

```
// Resultado de la llamada al servicio Web
String result = null;

// Dato a transmitir al servicio Web
int dato = 100;

// Creamos el objeto SOAP para la llamada al servicio
SoapObject rpc = new SoapObject
      ("urn:nombre del método", "nombre del método");

// Introducimos el parámetro en el mensaje SOAP
rpc.addProperty ("dato", dato);

// Configuramos la llamada al servicio Web
HttpTransport ht = new HttpTransport("URL destino", nombre del método);

// Llamamos al servicio Web y recibimos su respuesta
result = ht.call (rpc);
```

3.3 JSR 172

3.3.1 Introducción

Las APIs de servicios web (WSA) para J2ME se definen en la Java Specification Request 172 (JSR 172). Son dos paquetes independientes y opcionales para invocación de servicios remotos y análisis XML. Están centrados en la CDC y las CLDC 1.0 y CLDC 2.0. La importancia de esto estriba en que la especificación JSR 172 proporciona servicios de invocación remota y análisis XML en nivel de dispositivo, evitando así que los desarrolladores de software tengan que incorporar dichas funcionalidades en sus aplicaciones.

Los servicios web en la J2ME, definidos por la JSR 172, siguen las mismas especificaciones, arquitectura y modelo de invocación que los servicios web estándar, es decir:

- **SOAP 1.1** (Simple Object Access Protocol), que define el transporte y codificación de datos.
- **XML 1.0**, que define el lenguaje XML.
- **El esquema XML.**

JSR 172 no soporta la especificacion UDDI.

Hay una cantidad importante de especificaciones que cubran las distintas tecnologías relacionadas con servicios web, y su número continúa creciendo. Por ello la Organización para la Interoperabilidad de Servicios Web (WS-I) ha definido el WS-I Basic Profile 1.0, que establece las especificaciones mínimas y las reglas que todos los

proveedores de servicios web básicos deben seguir. La especificación JSR 172 se ha desarrollado de acuerdo al WS-I Basic Profile.

3.3.2 Invocación remota

La WSA de JSR 172 afronta los servicios web desde el punto de vista del cliente del servicio: WSA proporciona la API de invocacion remota (JAX-RPC) y el entorno de ejecución que permite a las aplicaciones J2ME consumir servicios en la web, pero no funcionar como productores de servicios. Aparte de esta diferencia, el resto de la arquitectura sigue la arquitectura y la organización estándar de los servicios web, como se puede ver en la figura 1.

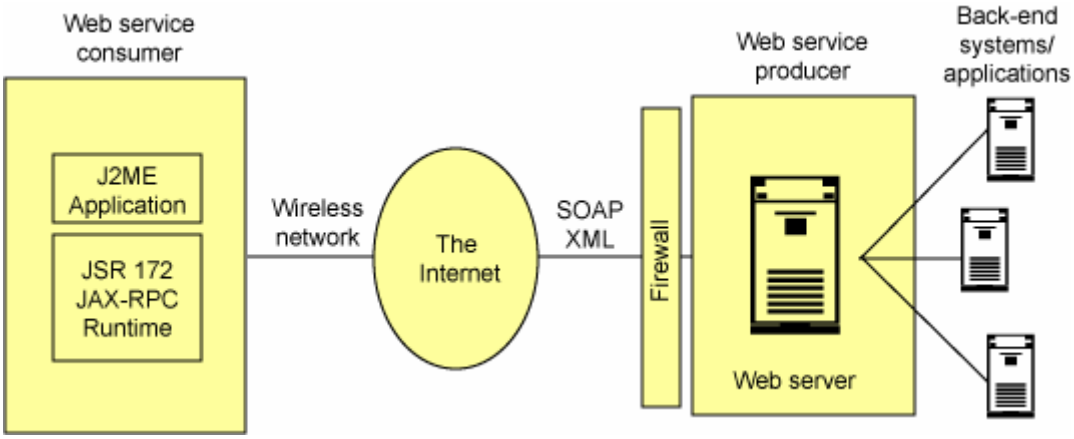

Figura 3.4: Arquitectura de WSA

Esta arquitectura de alto nivel está estructurada en:

- **Un cliente**, el consumidor de los servicios web. Consta de una aplicación J2ME, como MIDP, un stub de JSR 172 con las clases necesarias y la ejecución de JSR 172.

- **La red**: Hace referencia a la red inalámbrica y la red terrestre parte de Internet y los protocolos de comunicación. JSR 172 no impone el uso de XML en el propio dispositivo; permite otras implementaciones (siempre que sean transparentes al cliente y al servidor) para usar más eficientemente los enfoques de codificación, como el uso de protocolos binarios entre el dispositivo y la pasarela inalámbrica.

- **El servidor**: Productor de servicios web. Es un servidor web, típicamente detrás de un firewall o una pasarela Proxy. Éste servidor puede tener acceso a recursos de respaldo.

Las aplicaciones J2ME invocan servicios remotos a través de los stubs y de la ejecución de JSR 172, típicamente sobre HTTP y SOAP. Dichos stubs y ejecución esconden toda la complejidad asociada a la invocación del servicio remoto, incluyendo cómo se van a codificar y decodificar los parámetros de la solicitud y la respuesta, y todo lo relacionado con la red de comunicaciones. La invocación de métodos sigue el modelo síncrono de petición-respuesta, como se ve en la figura:

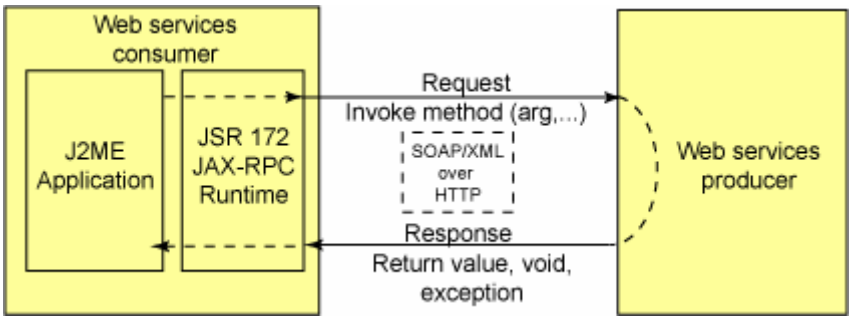

Figura 3.5: Modelo de invocación de JSR 172

Para consumir un servicio web antes hay que crear stubs de invocación de servicio. Estos stubs realizan tareas como codificado y decodificado de los valores enviados y recibidos, y actúan de interfaz de la ejecución de JSR 172 para invocar un punto de servicio remoto. Los stubs interactúan con la ejecución a través del Interfaz de Proveedor de Servicio (SPI), que abstrae los detalles de implementación y permite portabilidad de stubs entre distintas implementaciones.

Los stubs son generados usando una herramienta que lee el documento WSDL, que describe el servicio web a usar. Desde el punto de vista del dispositivo móvil, el documento WSDL que queremos consumir por lo general suele existir a priori, así que lo único que hay que hacer es generar los stubs JSR 172 WSA, usando una herramienta como el generador de stubs incluído en J2ME Wireless Toolkit 2.1.

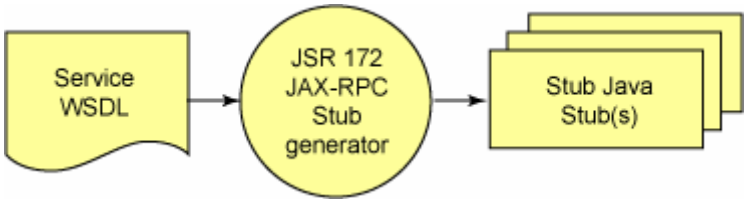

Figura 3.6: Generación del stub de JSR 172

Esto genera los ficheros Java del stub y las clases necesarias. También se ocupa del mapeado del tipo de datos de WSDL a Java, como se describirá con detalle más adelante.

Una vez se han generado los stubs y las clases necesarias, y la aplicación haya sido compilada e instalada en un dispositivo compatible con JSR 172, consumir los servicios web deseados es muy sencillo, y casi transparente. La invocación de métodos remotos se hace tan sencilla como la de métodos locales.

La API de invocación remota de métodos de JSR 172 está basada en un subconjunto de JAX-RPC 1.1 (API Java para RPC basada en XML). También está desarrollada según el WS-I Basic Profile. Soporta lo siguiente:

- **SOAP 1.1**
- **Cualquier transporte** que pueda entregar mensajes SOAP, com HTTP 1.1

- **La representación literal** de un mensaje SOAP que represente una petición o respuesta RPC

- Los siguientes **tipos de datos** y su correspondencia en Java:
  - ***xsd:boolean*** a *boolean* o *Boolean*.
  - ***xsd:byte*** a *byte* o *Byte*.
  - ***xsd:short*** a *short* o *Short*.
  - ***xsd:int*** a *int* o *Integer*.
  - ***xsd:long*** a *long* o *Long*.
  - ***xsd:float*** a *float*, o *Float*. Para plataformas basadas en CLDC 1.0, este tipo de datos se mapea a *String*.
  - ***xsd:double*** a *double*, o *Double*. Para plataformas basadas en CLDC 1.0, este tipo de datos se mapea a *String*.
  - ***xsd:string*** a *String*.
  - ***xsd:base64Binary*** a *byte[]*.
  - ***xsd:hexBinary*** a *byte[]*.
  - ***xsd:complexType*** a *Sequence* de tipos primitivos.
  - ***xsd:QName*** a *javax.xml.namespace.QName*.
  - Vectores de tipos primitivos y complejos (estructuras que contengan tipos primitivos y complejos) basados en el esquema de vector XML.

NO soporta lo siguiente:

- Mensajes SOAP con **adjuntos.**

- **Manejadores** (*handlers*) de mensaje SOAP.

- **Representación codificada** de mensajes SOAP.

- **Puntos de servicio** (productor de servicio web).

- Soporte de **descubrimiento** de servicios (UDDI).

La codificación en XML no es obligatoria en el dispositivo. Así se puede reducir el tráfico de la red permitiendo que las implementaciones usen más eficientemente la codificación de los datos, como un protocolo binario entre el dispositivo y la pasarela inalámbrica, siempre que dicha codificación sea transparente tanto para el productor del servicio web como para el consumidor.

La API de invocación remota de JSR 172 consiste en los siguientes paquetes:

- ***javax.microedition.xml.rpc***

- ***javax.xml.namespace***

- ***javax.xml.rpc***

- ***java.rmi*** (se incluye para satisfacer las dependencias con JAX-RPC)

Las APIs (con algunas excepciones, como *RemoteException*) no son directamente usadas por la aplicación, sino que ésta usa los stubs generados.

Una vez se hayan generado, compilado e instalado en el dispositivo el stub y los ficheros relacionados, consumir servicios remotos es muy simple. Si no contamos la inicialización específica dede JAX-RPC e importar *RemoteException*, el código de una aplicación que se desarrolle para consumir servicios web parece igual que el de una aplicación que no lo use. Ésta simplicidad es posible gracias a que los stubs, como ya hemos mencionado, ocultan los detalles relacionados con la invocación remota.

3.3.3 Análisis XML

La API de JSR 172 para el análisis XML de J2ME se basa en un subconjunto de la API JAXP 1.2 de J2SE y la API simple para análisis XML (SAX) 2.0. El hecho de que sea un subconjunto se explica por la limitada memoria disponible en los dispositivos J2ME. JAXP para J2ME consiste en los siguientes paquetes:

- ***javax.xml.parsers***: Contiene las clases del analizador SAX, incluyendo errores y excepciones.
  - **Interfaces**: Ninguna
  - **Clases**: *SAXParserFactory* (fábrica de analizadores SAX, *SAXParser* (representa un analizador SAX)
  - **Excepciones**: *ParserConfigurationException* (configuración del analizador)
  - **Errores**: *FactoryConfigurationError* (Errores relacionados con las fábricas de analizadores, como *analizador no encontrado)*

- ***org.xml.sax***: La API del subconjunto de SAX 2.0
  - **Interfaces**: *Attributes* (los atributos de un elemento), *Locator* (documento de localización de un evento)
  - **Clases**: *InputSource* (representa la fuente de datos de XML)
  - **Exceptions**: *SAXException* (excepción general se SAX), *SAXNotRecognizedException* (característica no reconocida), *SAXNotSupportedException* (característica reconocida pero no soportada), *SAXParseException* (excepción de análisis SAX)
  - **Errores**: Ninguno

- ***org.xml.sax.helpers***: Define la clase básica para procesar eventos.
  - ***Interfaces***: Ninguna
  - ***Clases***: *DefaultHandler* (manejador del procesado de eventos)
  - ***Excepciones***: Ninguna
  - ***Errores***: Ninguno

JAXP para J2ME proporciona las clases, interfaces y excepciones necesarias para analizar documentos XML. La especificación de JAXP para J2ME proporciona más información sobre dichas clases.

JAXP para J2ME debe ser compatible con XML 1.0, y puede validar (usando DTD) o no; el analizador seguirá las reglas de validación indicadas en la especificación de XML 1.0. Esta validación es una operación que consume muchos recursos, por eso la decisión de incluirla dependerá del fabricante del dispositivo y se basará en las limitaciones de almacenamiento y procesado del dispositivo. Para saber si el analizador está validando se puede llamar al método *SAXParser.isValidating()*.

Los analizadores de JAXP para J2ME deben soportar espacios de nombre de XML, tal y como se define en la recomendación *W3C XML Namespaces 1.0*, codificación UTF-8 y UTF-16, y DTDs. No están soportados, debido a las causas ya comentadas, el Modelo de Objetos de Documento (DOM) y el Lenguaje de Transformación basado en Plantillas de Estilo (XSLT).

JAXP para J2ME proporciona todas las funciones necesarias para analizar documentos XML. De hecho, es posible analizar documentos XML simplemente usando un subconjunto de las funciones incluídas en JAXP para J2ME. La figura siguiente muestra los elementos típicos de una aplicación JAXP para J2ME.

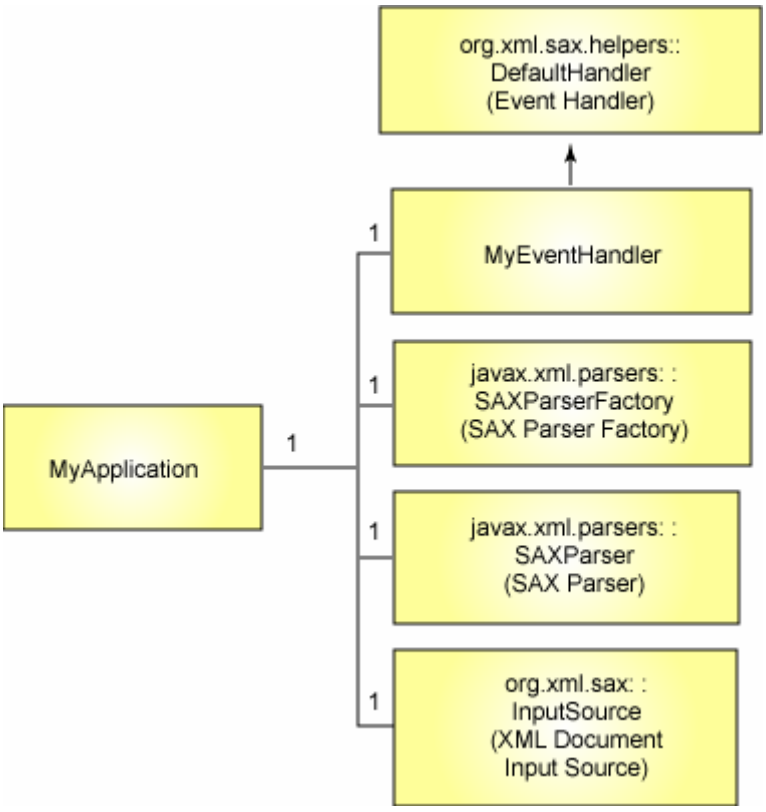

Figura 3.7: Elementos típicos de una aplicación JAXP

Estos elementos son:

- ***MyApplication***: La aplicación J2ME
- ***MyEventHandler***: Una clase que maneja los eventos específicos de la aplicación. Extiende org.xml.sax.DefaultHandler.

- **Default event handler** (*DefaultHandler*): La clase básica de procesado de eventos. Extendida por *MyEventHandler*. Proporciona el procesado de eventos por defecto.

- **SAX parser factory** (*SAXParserFactory*): La fábrica de analizadores SAX usada para obtener una instancia del analizador SAX

- **SAX parser** (*SAXParser*): El analizador SAX usado para analizar los documentos XML de entrada. Este analizador es creado llamando a *SAXParserFactory*.

- **XML input stream**: El documento XML de entrada que queremos analizar, representado como un flujo *java.io.InputStream*, o bien *org.xml.sax.InputSource*. El documento XML puede leerse desde una conexión de red o localmente.

Por supuesto, las aplicaciones típicas tienen más elementos que los mostrados en la figura, como pantallas, modelos de datos, controladores, lectores de entrada de datos, etc. Es importante ver que es una buena idea implementar una clase que encapsule toda la lógica relacionada con XML. Esta lógica consiste principalmente en obtener los objetos JAXP, inicializar, invocar el parser y procesar los eventos de procesado del documento XML de entrada.

El uso de JAXP consiste en tres fases:

- **Definir** (escribir) el manejador específico para la aplicación, una subclase de *DefaultHandler*.

- Usar *SAXParserFactory* para **crear una instancia** del analizador SAX (*SAXParser*).

- **Analizar** el documento XML de entrada invocando el método *SAXParser.parse()*.

El analizador SAX invoca las llamadas del manejador de eventos *startDocument(), endDocument(), startElement(), endElement(),* y *characters()* al procesar las distintas partes del documento XML.

En la figura siguiente podemos ver las distintas interacciones entre los objetos de la aplicación a través del tiempo.

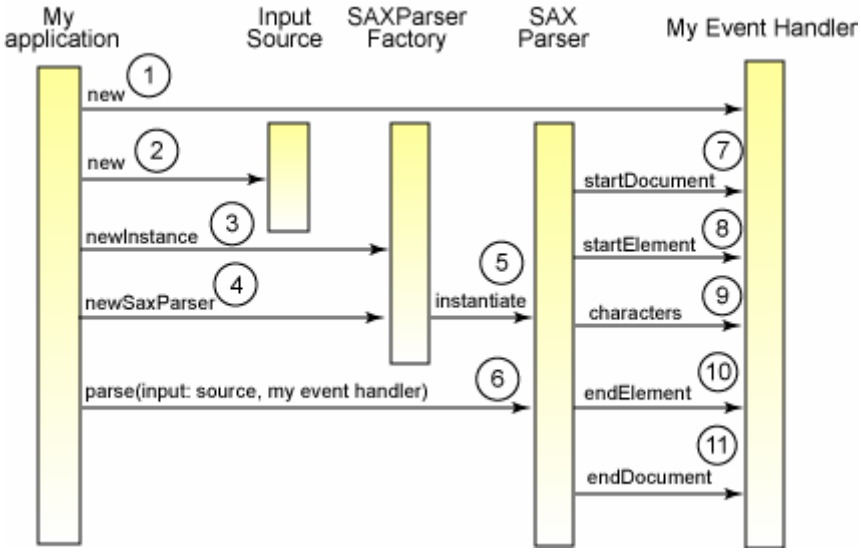

Figura 3.8: Diagrama de la secuencia de la API de JAXP.

Estas interacciones son:

- La aplicación JAXP instancia el manejador de eventos (subclase de *DefaultHandler*).
- La aplicación JAXP instancia la fuente XML de entrada (*InputSource*).
- La aplicación JAXP recibe una instancia de la fábrica de analizadores SAX (*SAXParserFactory*)
- La aplicación JAXP recibe una instancia del analizador SAX (*SAXParser*) llamando a la fábrica de analizadores SAX.
- La fábrica de analizadores SAX instancia y devuelve un *SAXParser*.
- La aplicación JAXP llama al analizador SAX para iniciar el análisis del documento XML.

Una vez se ha iniciado el análisis, *SAXParser* invoca las llamadas al manejador de eventos así:

- ***startDocument***(): Se llama al principio del documento.
- ***startElement***(): Se llama cuando se encuentra un nuevo elemento. Esta llamada también incluye los atributos del elemento, si hay alguno.
- ***characters***(): Se llama cuando se encuentran caracteres de algún elemento.
- ***endElement***(): Se llama cuando se cierra un elemento.
- ***endDocument***(): Se llama al finalizar el análisis del documento.