Enfoques Alternativos para el Consumo de Servicios Web Mediante Clientes HTML

Claudio Ochoa

DSIC, Universidad Politécnica de Valencia,
Camino de Vera s/n, E-46022 Valencia, España.
cochoa@dsic.upv.es

Abstract. Un servicio web es un servicio, con un interfaz definido y conocido, al que se puede acceder a través de Internet. Existen varias formas de consumir servicios web, sin embargo aún no es posible hacerlo desde un cliente HTML/Javascript. Esto se debe principalmente a que el código Javascript está limitado a poder cargar datos del *mismo* dominio en el cual reside. En este artículo se exploran dos enfoques alternativos para solucionar este problema, y se analizan las ventajas y desventajas de cada uno.

1 Introducción

Internet ha evolucionado con gran celeridad en los últimos años, ofreciendo en la actualidad páginas interactivas, que son realzadas generalmente mediante scripts residentes en el lado del cliente o del servidor, entre otros componentes tales como animaciones Flash, imágenes animadas, etc. Los servicios Web permitirán extender la performance y la versatilidad de dichas páginas proporcionando nuevas capacidades que los navegadores existentes pueden usar para brindar un nuevo nivel de performance. Esto permitirá la evolución a una futura generación de sitios Web programables que establecerán vínculos directamente con organizaciones, aplicaciones, servicios y dispositivos entre sí.

SOAP [8] es la base sobre la que se construyen los servicios web. SOAP es un estándar que emplea XML [11] para comunicar procesos y dispositivos. Debido a que SOAP es implementado sobre HTTP, se está convirtiendo rápidamente en un mecanismo popular para la comunicación de plataformas cruzadas de aplicaciones basadas en web. Sin embargo, el uso de SOAP es tedioso, requiriendo mucha intervención manual para la construcción de dicho mensaje esperado por el servicio web. La respuesta del servicio web, la cual también utiliza SOAP, debe ser analizada (*parsed*) manualmente para recuperar la información requerida. Esto dificulta la creación de clientes HTML para el consumo de Servicios Web. Sin embargo, el mayor problema que presenta la implementación de este tipo de clientes es el modelo de seguridad de dominios cruzados (*cross-domain security model*), en el cual el código Javascript [3] está limitado a poder cargar datos del *mismo* dominio en el cual reside. Algunas soluciones para este problema han sido propuestas de manera independiente, principalmente por Microsoft y por Netscape. En este artículo exploramos ambas soluciones, y enumeramos sus ventajas y desventajas a través de la implementación de clientes HTML para el consumo de Servicios Web.

En la sección 2 se explora el modelo de seguridad de dominios cruzados, y se explica por que esto dificulta la implementación de clientes HTML para servicios web. En la sección 3 se analizan las soluciones propuestas por Microsoft® para Internet Explorer, y por Netscape para navegadores Gecko-compatibles (Netscape 7.x, Mozilla, CompuServe 7, AOL for Mac OS X). A continuación, la sección 4 introduce un ejemplo de una implementación de un servicio web para monitoreo de estado del servicio de mensajería MSN, y se implementan clientes HTML utilizando los enfoques de Microsoft y Netscape. Finalmente en la sección 5 se analizan las conclusiones de este trabajo.

2 El modelo de seguridad de dominios cruzados

En esta sección exploramos el principal problema que supone la implementación de un cliente HTML para consumir un servicio Web, el cual es el modelo de seguridad de dominios

cruzados (*cross-domain security model*). También se exploran soluciones alternativas a este problema desde los lados del cliente y del servidor.

En el modelo de seguridad de dominios cruzados, el código Javascript [3] está limitado a poder cargar datos del *mismo* dominio en el cual reside. Cuando se carga un script externo detrás de un cortafuegos (*firewall*), este es ejecutado en un modelo de seguridad de caja de arena (*sandbox*, un entorno protegido donde las aplicaciones pueden ser ejecutadas sin riesgo para el resto del sistema). Estos scripts podrían, legítimamente, requerir acceso a recursos internos, pero si se les permitiera dicho acceso se podría comprometer a recursos que normalmente no estarían disponibles a aplicaciones externas al cortafuegos. Dicho de otro modo, el *sandbox* debe distinguir y proteger recursos internos.

A continuación exploramos algunas soluciones alternativas a este problema, las que pueden ser aplicadas desde el lado del cliente o del servidor [5].

Controlando el acceso a recursos en el cliente

Varias soluciones del lado del cliente se han diseñado para resolver este problema.

Restricción del mismo origen *(Same Source Restriction)*: Una solución muy simple aunque restrictiva es permitir que los scripts en la caja de arena tengan acceso solamente a recursos en el dominio del cual fueron cargados. Esta política ha sido exitosamente aplicada en applets Java y código Javascript (JS).
Desafortunadamente, esta técnica no permite que los scripts tengan acceso a recursos externos legítimos no alojados en su mismo dominio. Esto *impide que el código Javascript pueda acceder a servicios web* y datos publicados en otros dominios.

Lista de sitios confiables *(white-list)*: Otra solución es la creación de una lista de sitios confiables, la cual es menos restrictiva que la solución descripta anteriormente, al permitir que código JS de dominios externos pueda acceder a recursos almacenados en otros dominios. Estas listas pueden construirse de manera de que permitan definir, con una mayor granularidad, qué dominios pueden acceder a otro dominio dado, pero esto requiere un manejo extensivo y costoso, y el cual puede ser muy propenso a errores, abriendo agujeros en cortafuegos, y pudiendo afectar a otros servicios o recursos del mismo dominio.

Scripts Firmados: Cuando el autor de un script lo ha firmado digitalmente, se le puede otorgar cierto grado de confianza adicional. Desafortunadamente, el problema básico persiste, ya que pudiera ocurrir que un usuario malicioso escriba un script firmado que pudiera acceder a recursos internos.

Preguntándole al usuario: Cuando el *sandbox* no puede determinar si debe permitir acceso a un recurso al script que está ejecutándose en ese momento, se puede preguntar al usuario si desea concederle privilegios especiales. Esto se hace actualmente para scripts almacenados localmente y para scripts firmados. Esto puede combinarse con las otras soluciones mencionadas anteriormente, como por ejemplo listado de sitios confiables, etc. Sin embargo, el mayor defecto que esta solución plantea es que el usuario típico usualmente no es un experto en infórmatica, y puede abrir inadvertidamente un agujero en el cortafuego de su compañía. Esta solución es la implementada por los *behaviors* de Microsoft, como veremos en la sección 3.

Controlando el acceso a recursos en el servidor

El acceso a recursos internos por parte de scripts desconocidos debiera estar bajo el control del propietario de dicho recurso, en lugar de delegar esta responsabilidad en el usuario final.

Usando un encabezado SOAP para la verificación: Los mensajes SOAP agregan un encabezado que debe ser verificado y aceptado por el receptor de dicho mensaje, y el cual puede usualmente distinguir un script confiable de uno desconocido.
Sin embargo, la verificación no puede por sí sola garantizar que peticiones del servicio de fuentes desconocidas sean rechazadas correctamente por los servicios que deben ser protegidos por el cortafuego
Por otra parte, puede ser peligroso modificar un servicio SOAP para que ignore el encabezado de verificación específico.

Fichero de declaraciones: Una solución más robusta es confiar en encontrar un fichero llamado "web-scripts-access.xml" en el directorio raź del servidor con el cual el script en la caja de arena desea comunicarse. Este fichero puede ser fácilmente construido por los proveedores de recursos públicos. Esta es la solución propuesta por Netscape, como veremos en la siguiente sección.

3 Soluciones propuestas: Behaviours vs. Proxys WSDL

Para afrontar el problema del modelo de seguridad de dominios cruzados, descrito en la sección anterior, se han propuesto algunas soluciones, donde las más destacadas son la de Microsoft® para su navegador Internet Explorer, y la de Netscape para los navegadores compatibles con Mozilla. Microsoft propone el uso de *behaviours* [4], los cuales permiten que un cliente HTML pueda invocar métodos remotos expuestos por servicios web u otros servidores Web, a través de un simple código JS. Por otra parte, Netscape ha propuesto un modelo de seguridad al W3C en el que el proveedor de servicios web determina, para cada uno de sus servicios, si estos son públicamente accesibles, son privados, o son accesibles solo a ciertos dominios.

Ambas propuestas son estudiadas en más detalle a continuación.

3.1 Behaviours

Un *WebService Behaviour* [4] permite que scripts ejecutándose en un cliente puedan invocar métodos remotos expuestos por servicios web, u otros servidores Web, que soporten SOAP y WSDL 1.1. Este *behaviour* permite que los desarrolladores de páginas web utilizen SOAP sin que tengan un conocimiento profundo de dicho protocolo. El *behaviour* soporta el uso de una amplia variedad de tipos de datos, incluyendo tipos de datos SOAP intrínsecos, vectores, objetos, y datos XML. El *behaviour* se implementa a través de un fichero HTC (*HTML component*), y es soportado por Microsoft® Internet Explorer 5 y versiones posteriores.

Aún cuando los *behaviours* utilizan el protocolo SOAP para comunicarse con los servicios Web, su propósito es proporcionar una manera simple de aprovechar este protocolo sin requerir un vasto conocimiento del mismo.

Una de las ventajas de los *behaviours* radica en que no es necesario modificar el código de los servicios web existentes, solo se debe agregar unas líneas de código JS a la página web desde donde se pretende acceder a dicho servicio. El servicio Web debe ser accesible desde el servidor que almacena la página web. Para iniciar una llamada remota a través del *behaviour*, sólo es necesario conocer los métodos — y los parámetros requeridos — del servicio que se desea invocar, los que se obtienen del fichero WSDL de dicho servicio.

Beneficios

La principal ventaja de los *behaviours* es que proporcionan una manera simple para acceder a métodos expuestos por servicios web, a través del protocolo SOAP. El *behaviour* permite invocar un método remoto usando algunos métodos simples expuestos por el *behaviour* —`useService` y `callService`, los cuales son explicados mas adelante—, los cuales son invocados desde un código JS. Por medio de DHTML se puede actualizar el contenido de la página.

Un *behaviour* es un componente reutilizable, por lo que sus capacidades encapsuladas ayudan a reducir la necesidad de duplicar código, mejorando de este modo la organización del script de la página cliente. Todo el código específico del protocolo, así como la mayoría del código requerido para manejar las transacciones de datos, se encapsula en el código JS, lo cual es una ventaja general de usar *behaviours* DHTML. Una vez que el *behaviour* es inicializado, se pueden invocar métodos de uno o varios servicios Web.

Los servicios web generalmente residen en repositorios remotos, y encapsulan bloques de funcionalidad que pueden ser luego ensamblados, empaquetados o presentados de manera muy variada en una página web. El uso de una arquitectura tan distribuida provee una gran escalabilidad ya que tareas que hacen un uso intensivo de CPU o de datos pueden

ser organizadas en servicios web, liberando al cliente de carga innecesaria. Los *behaviours* pueden ayudar de este modo a mejorar la organización general de aplicaciones web.

El uso de *behaviours* para el acceso de servicios Web simplifica el código cliente. Los *behaviours* pueden ser actualizados a medida que el standard SOAP evoluciona, sin requerir cambios significativos en el código existente en los clientes.

Los *behaviours* simplifican el uso de servicios web, ocupándose de la comunicación de los paquetes SOAP entre el navegador y el servicio web. El código que realiza esta tarea está encapsulado en el *behaviour*, por lo que el código JS necesario para el acceso al servicio web es muy simple.

Los *behaviours* analizan (*parse*) el XML retornado por el método invocado y expone objetos y propiedades al código JS. Los objetos expuestos por el *behaviour* permiten el manejo de errores y un fácil acceso a los datos retornados. El desarrollador sólo necesita inicializar el *behaviour* una sola vez (por ejemplo, en el evento *onLoad* de la página), aun cuando el script referencie a métodos de servicios web diferentes.

Para comprender porque los *behaviours* son tan útiles, podemos comparar esta técnica con un enfoque que utiliza formularios, como se muestra en la Figura 1.

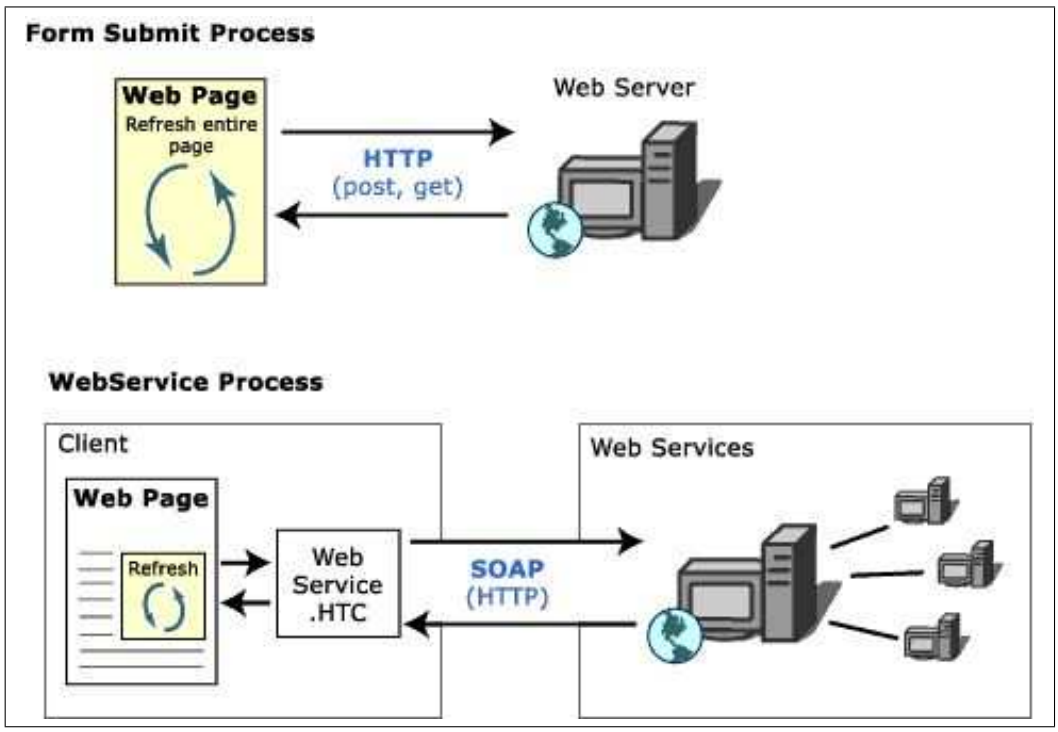

Fig. 1. Comparación entre un formulario y un *behaviour*

La primera parte de la ilustración muestra cómo una página web que contiene un formulario emplea comúnmente los método *get* o *post* a través de HTTP para actualizar una página web. Cada vez que se envía un formulario, el cliente es redireccionado a un nuevo URL, desde donde el navegador descarga la página entera. Este método se utiliza extensivamente en la actualidad, pero es ineficaz porque la página Web debe actualizar la totalidad del contenido, aun cuando solamente un porcentaje pequeño de la página haya cambiado realmente. Este comportamiento puede causar grandes demoras.

La segunda parte del diagrama ilustra cómo una página web puede utilizar un *behaviour* para evitar las desventajas asociadas al enfoque tradicional. El *behaviour* recibe llamadas a métodos del servicio web a través del código JS, envía un requerimiento al servicio web y

una vez que los resultados son retornados, estos son pasados al código JS, de manera que estos puedan ser utilizados para actualizar la página a través de DHTML.

Una característica clave de los *behaviours* es que permiten el acceso a un servicio web por parte de código JS sin necesidad de navegar a otro URL.

Los *behaviours* son muy simples de utilizar, para invocar un método de un servicio web, primero se debe asociar un archivo llamado `WebService.HTC` a algun elemento de la página, por ejemplo, a un elemento `<DIV>` como se muestra a continuación:

```
<div id="miServicio" style="behavior:url(webservice.htc)" onresult="Resultado()">
```

Esta asociación nombra al behavior como `miServicio` y declara que cuando el servicio web retorne un resultado la función JS `Resultado()` será invocada. Esto posibilita que se puedan hacer invocaciones síncronas o asíncronas al servicio web desde el código cliente. La naturaleza asíncrona de una invocación a un método remoto significa que hay un retardo entre la ejecución del método y la llegada del resultado. Por otra parte, cuando se realiza una llamada síncrona el procesamiento en el código JS se detiene hasta que el método `callService` haya terminado. Por defecto el modo de los *behaviours* es asíncrono.

Como se podrá observar, las invocaciones síncronas a servicios remotos congelan la interfaz de usuario hasta el retorno de la invocación, por lo que no son prácticas para aplicaciones basadas en web.

A continuación, se invocan sobre el `behaviour` los métodos `useService` y `callService`, el primero establece la dirección web donde se encuentra el servicio web a ser llamado, y el nombre del servicio, mientras que el segundo efectúa la llamada al método correspondiente con los parámetros adecuados.

En el ejemplo que sigue a continuación, se invoca al método `add` del servicio web `AdderWS`:

```
miServicio.useService("http://direccionRepositorio.com/servicio.wsdl","AdderWS")
iCallID = miServicio.AdderWS.callService("add",int1,int2);
```

Este código debe ir embebido en una función JS, la cual puede ser invocada cuando se carga la página, o luego de interactuar con el usuario.

Una vez que el servicio web es invocado, y este retorna un resultado, entonces la función que se especificó en la declaración del fichero webservice.htc es invocada. En nuestro ejemplo, esta función es llamada `Resultado()`, y un ejemplo del uso del valor retornado por el servicio web es el siguiente:

```
function Resultado(){
if (event.result.error)
     alert("ERROR: "+event.result.errorDetail.string);
   else
     res = event.result.value;
   ...
}
```

En este caso, luego de chequear que no hubo un error, el valor retornado se almacena en la variable `res` para su uso posterior.

Los *behaviours* ocultan al desarrollador el proceso de invocar a los métodos, y el procesamiento y análisis de los paquetes XML retornados por el servicio web. El usuario tiene la opción de usar un manejador de eventos o una función `callback` para procesar el resultado. Si se utiliza el manejador de eventos, el *behaviour* dispara un evento `onresult`, el que ocurre al recibir la respuesta del método llamado. Por otra parte, si se utiliza una función *callback*, un objeto resultado es pasado directamente como un parámetro de entrada a dicha función

El código JS que emplea *behaviours* debiera "testear" la propiedad `error` para determinar si la invocación al método fué exitosa. En caso de que un error haya ocurrido, el objeto `errorDetail` contiene propiedades que pueden ser evaluadas para ayudar a identificar la causa del error.

Puede observarse claramente que los *behaviours* permiten que los desarrolladores aprovechen la versatilidad de XML y SOAP sin necesidad de desarrollar código SOAP.

3.2 WSDL Proxying

Netscape ha propuesto un modelo de seguridad al W3C en el que el proveedor de servicios web determina, para cada uno de sus servicios, si estos son públicamente accesibles, son privados, o son accesibles solo para ciertos dominios. Este modelo se destaca por su simplicidad, ya que el proveedor del servicio web solo tiene que poner un fichero XML en el directorio raíz donde se encuentra el servicio web. Por ejemplo, el fichero utilizado por XMethods es http://services.xmethods.net/web-scripts-access.xml.

Este modelo se denomina *WSDL proxying*, y está soportado en los navegadores Netscape 7.1 y Mozilla 1.4, y versiones superiores.

Un fichero WSDL describe las interfases que un servicio web provee. Usando este fichero, Gecko™[7], que es un navegador basado en estándares empotrado en Netscape y Mozilla, entre otros, ofrece a los desarrolladores una manera de incorporar llamadas a los métodos de los web services en el código JS de la página cliente como si fuera un objeto nativo, ocultando de este modo los aspectos SOAP y XML. Por ejemplo, luego de crear una instancia proxy de un servicio web usando WSDL, se pueden invocar métodos sobre el objeto proxy de la misma manera que se haría sobre cualquier objeto JS, por ejemplo

```
proxy.getTranslation("en_fr", "Hello World");
```

Los proxys a los servicios web se crean en JS como una instancia de `WebServiceProxyFactory`. El fichero WSDL se carga invocando al método `createProxyAsync` del objeto proxy.

`createProxyAsync` toma cinco parámetros:

- El primer parámetro indica la ubicación del fichero WSDL. Por ejemplo, para el servicio web BabelFish [1], es http://www.xmethods.net/sd/2001/BabelFishService.wsdl
- El segundo parámetro es el número de puerto, el cual se encuentra en el mismo fichero WSDL, en el elemento *service*, como se muestra en la figura 2.

```
JavaScript:
var factory = new WebServiceProxyFactory();
factory.createProxyAsync("http://www.xmethods.net/sd/2001/BabelFishService.wsdl",
                         "BabelFishPort", "", true, aCreationListener);

WSDL:
<?xml version="1.0"?>
<definitions name="BabelFishService" ...>
  ...
  <service name="BabelFishService">
    <documentation>Translates text between a variety of languages.
</documentation>
    <port name="BabelFishPort" binding="tns:BabelFishBinding">
      <soap:address location="http://services.xmethods.net:80/perl/soaplite.cgi"/>
    <port>
  </service>
</definitions>
```

Fig. 2. Inicialización de un proxy

- El tercer parámetro es un calificador opcional sin importancia.
- El cuarto parámetro indica si el proxy debe ser cargado de manera asíncrona o no. Actualmente, los navegadores mencionados anteriormente no soportan creación de proxy de manera síncrona, por lo que este parámetro es fijado en `True`.
- El quinto parámetro es la función *callback* que es invocada una vez que el proxy es generado, o en caso de que haya ocurrido un error durante la generación.
  Debido a que el proxy se genera de manera asíncrona, la función `onLoad` es invocada luego de que el proxy es inicializado. Siempre que un error se produzca se invocará al método `onError`. Esto se puede observar con detalle en la figura 3.

Por cada método del servicio web que se invoque, deberá existir una función *callback*, la cual tendrá el nombre {`nombreMétodo`}`Callback`. Por ejemplo, el servicio web BabelFish sólo contiene un método, `BabelFish`, por lo que la función *callback* es llamada `BabelFishCallback`. Como se ve en el extracto de WSDL de la figura 4, este método toma como parámetro un `BabelFishRequest`, el cual tiene a su vez dos parámetros, la palabra a ser traducida y el idioma en el que será traducido, y retorna el valor traducido como string.

```
var listener = {
  // invocada cuando se instancia el proxy
  onLoad: function (unProxy)
  {
    gProxy = unProxy;
    gProxy.setListener(listener);
    requestTranslation(unValor);
  },
  // invocada cuando ocurre un error
  onError: function (unError)
  {
    alert("Se ha producido un error al procesar el fichero WSDL: " + unError);
  },
  // función callback
  BabelFishCallback : function (unResultado)
  {
    alert(unResultado);
  }
};

function requestTranslation(unValor){
  if (gProxy) {
    gProxy.BabelFish("en_fr", unValor);
  } else {
    alert("Error: Proxy no pudo inicializarse!");
  }
}
```

Fig. 3. Código JS para inicializar un proxy

```
WSDL:
<message name="BabelFishRequest">
  <part name="translationmode" type="xsd:string"/>
  <part name="sourcedata" type="xsd:string"/>
</message>
<message name="BabelFishResponse">
  <part name="return" type="xsd:string"/>
</message>
<portType name="BabelFishPortType">
  <operation name="BabelFish">
    <input message="tns:BabelFishRequest"/>
    <output message="tns:BabelFishResponse"/>
  </operation>
</portType>
```

Fig. 4. Extracto de WSDL del servicio web BabelFish

4 Implementación de Clientes HTML de servicios web

En esta sección, se analizan las soluciones propuestas por Microsoft y Netscape, utilizando para tal fin dos servicios web, uno que proveerá un monitor de estado del servicio de mensajería MSN (implementado por este autor), y otro es el servicio de traducción BabelFish, el cual está almacenado en XMethods.net.

Los clientes HTML están disponibles en

`http://www.dsic.upv.es/~cochoa/PhDCourses/WebServices.html`

4.1 Servicio Web de Monitoreo del servicio de mensajería MSN.

Este servicio web permitiría que un usuario pueda poner un icono en su página personal indicando si en ese momento se encuentra disponible para ser contactado a través de MSN, y también permitiría conocer el estado de cualquier usuario MSN utilizando para tal fin un formulario en una página web. En cualquier caso, se accedería desde la página web al servicio web, pasándole como parámetro la dirección de MSN del usuario del que se desea conocer su estado, y este servicio respondería con un `string` indicando el estado actual de dicho usuario.

Con el propósito de entender la implementación, a continuación describimos el protócolo de mensajería MSN.

4.2 El Protócolo de mensajería MSN

El protócolo de mensajería MSN® ha evolucionado bastante a lo largo de los últimos años. Actualmente los servidores de Microsoft permiten clientes de las versiones 8, 9 y 10 del protócolo. Estas versiones son conocidas como MSNP8, MSNP9 y MSNP10 respectivamente.

Básicamente, el protocolo de mensajería MSN consiste de una serie de comandos entre el cliente y el servidor. Por ejemplo, cuando un contacto de un usuario se desconecta el servidor envía un mensaje como el siguiente al cliente:

```
FLN juan@hotmail.com
```

Al recibir un mensaje de este estilo, el cliente debe remover el contacto de la lista de usuarios conectados, y colocarlo en la lista de usuarios desconectados.

Presencia

La información de presencia (*presence*) es la que indica el estado actual del cliente. En MSNP8, esto incluye el estado (ocupado, conectado, ausente, etc.), el nombre del usuario, y un número que representa información específica del cliente (por ejemplo, soporte de cámara web, etc.)

Por ejemplo, el siguiente mensaje enviado por el servidor

```
NLN AWY juan@hotmail.com juancito 268435492
```

notifica al cliente que el contacto `juan` puede recibir mensajes instantáneos pero está en estado `ausente`, y figurará en la lista de contactos como `juancito`. El código 268435492 indica, entre otras cosas, que su cliente usa el protócolo MSNP10.

Estados

Los contactos de un usuario MSN pueden estar en uno de dos estados básicos: conectados o desconectados. El estado conectado tiene los siguientes sub-estados:

```
NLN - Conectado
BSY - Ocupado
IDL - No disponible
BRB - Vuelvo enseguida
AWY - Ausente
PHN - Al teléfono
LUN - Salí a comer
```

4.3 El Servicio Web Monitor MSN

En esta sección describimos detalles específicos de la implementación del servicio web para monitoreo de estado del servicio de mensajería de MSN, el cual fue desarrollado en C#.

dotMSN

Con el propósito de facilitar la comunicación con el servidor MSN, en este trabajo se ha empleado la librería dinámica *dotMSN* [2]. *dotMSN* es una librería dinámica que provee una serie de métodos para emplear el servicio de mensajería MSN. Esta librería está implementada en C#, por lo que puede ser utilizada por cualquier lenguaje soportado por el entorno .NET®. La principal ventaja de *dotMSN* es que provee una abstracción del protócolo de bajo nivel MSN, a través de un enfoque Orientado a Objetos muy simple.

Implementación

El modelo lógico de la aplicación se observa en la figura 5. Las clases principales del

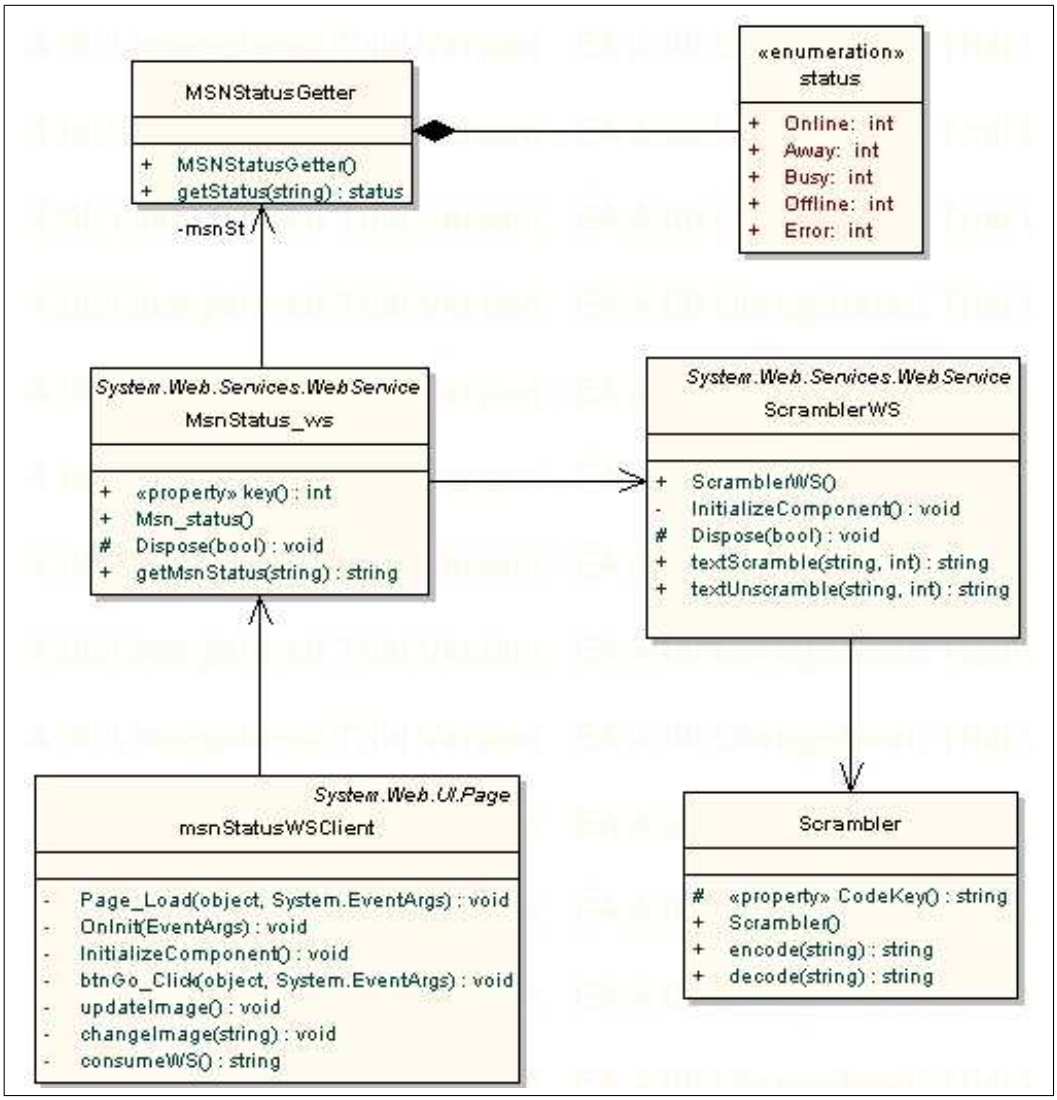

Fig. 5. Modelo Lógico del Servicio Web MSNStatus˙ws

modelo son:

MSNStatusGetter: esta clase es la que usa la dll *dotMSN*, disponiendo sólo de un método público `getStatus(string)`, el cual recibe como parámetro un string conteniendo la dirección msn de interés, y retorna uno de los valores de la enumeración `status`, es decir `Online,Busy, Away, Offline` o `Error`. Cada uno de estos valores (excepto el de error), tiene un icono asociado en los clientes MSN, como se muestra en la figura 6. Estos iconos son también empleados por el cliente HTML para denotar el estado del usuario requerido.

Scrambler: esta clase ofrece facilidades para la encriptación de texto. En particular, el método `textScramble(string, int)` recibe un string conteniendo sólo carácteres ASCII imprimibles por pantalla (en particular nos interesan solo estos carácteres ya que son los que pueden formar parte de una dirección MSN), y retorna un string codificado con carácteres del mismo subconjunto. Además de dicho string, este método recibe como segundo parámetro una clave entera, la cual es necesaria para la decodificación de dicho string.
`textUnscramble(string,int)` realiza la operación inversa, es decir recupera el string original siempre y cuando se conozca la clave utilizada para la codificación.

ScramblerWS: esta clase representa al servicio web que implementa los servicios de codificación/decodificación de strings, utilizando `Scrambler` para tal fin.

MSNStatusWS: esta clase representa al servicio web que implementa el servicio de monitoreo de estado del servicio de mensajería MSN. Utiliza para tal fin la clase MSNStatusGetter, y el servicio web ScramblerWS para decodificación de strings mediante una clave interna. Ofrece dos métodos, `getMsnStatus_plainEmail` y `getMsnStatus_codedEmail`, los cuales reciben como parámetro el email a ser monitoreado, y un email codificado, respectivamente.

msnStatusWSClient: esta clase representa un cliente HTML consumidor del servicio web `MSNStatus_ws`.

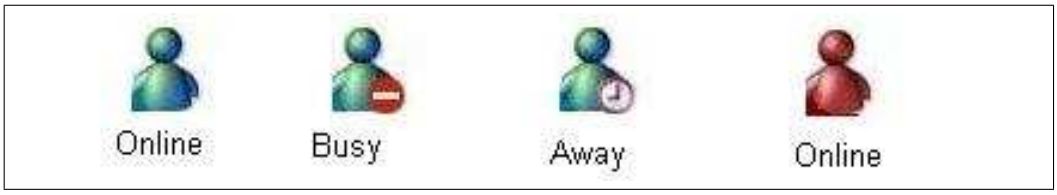

Fig. 6. Iconos para cada estado posible de un usuario de MSN.

Como se observa, el servicio web `MSNStatusWS` hace uso del servicio web `ScramblerWS` cuando se invoca el método `getMsnStatus_codedEmail`, este método es útil cuando un usuario desea hacer uso del servicio web desde su página personal, pero no desea hacer público su email.

4.4 Cliente HTML usando *behaviours*

Esta solución emplea el fichero webservice.htc [9], el cual se debe colocar en el mismo directorio que la página cliente que implementará el consumo del servicio web. Además, en el cuerpo HTML se debe incluir un código como el siguiente

```
<div id="ServicioWeb" style="behavior:url(webservice.htc)"
 onresult="Resultado()">
```

el cual indica el nombre del fichero y el nombre que se le asigna al behavior (`ServicioWeb`). Además, se indica que cuando el servicio web retorne un resultado deberá invocar a la función JS `Resultado()`.

El código JS necesario para implementar un cliente HTML utilizando behaviors es muy simple, como se observa a continuación:

– En primer lugar, en el evento `onLoad` del elemento `<BODY>` de la página cliente se debe invocar a una función que inicializa el servicio web. En nuestro caso, la función List asocia `ServicioWeb` al servicio web MSNStatusWS, cuya descripción (fichero WSDL) está en `http://udine.dsic.upv.es/msn_ws/msn_status.asmx?WSDL`.
– A continuación, se invoca al método `getMsnStatus_codedEmail` pasando como parámetro el email correspondiente, el cual ha sido previamente codificado con el servicio web `ScramblerWS`.

```
function List(){
  ...
  ServicioWeb.useService("http://udine.dsic.upv.es/msn_ws/msn_status.asmx?WSDL",
                         "MSNStatusWS");
  CallID=ServicioWeb.MSNStatusWS.callService("getMsnStatus_codedEmail",
                         "z6TyY2rP+#SM3KA-}7g65V7#8");
}

function Resultado(){
   if (event.result.error)
     alert("ERROR: "+event.result.errorDetail.string);
   else
     updatePhoto(event.result.value);
}
```

– Finalmente la función `Resultado` simplemente chequea si hubo algún error, y actualiza una imagen en la página a través de la función JS `updatePhoto`. Note que `event.result.value` contendrá un `string` conteniendo uno de los valores de la enumeración `status`.

En particular, la implementacin de esta solución resultó muy simple, sin ofrecer mayores problemas. El único inconveniente que presenta es que cuando el navegador ingresa en ésta página por primera vez, muestra el mensaje de la Figura 7 ya que el fichero .htc debe ser descargado en el ordenador donde reside la página cliente.

Fig. 7. Mensaje mostrado por la solución Microsoft.

La implementación del cliente HTML para el servicio web BabelFish se pudo hacer sin problemas, aún cuando este Servicio Web está almacenado en un servidor y dominio diferente.

4.5 Cliente HTML usando *WSDL Proxying*

En el caso de esta alternativa, la implementación del cliente HTML del servicio web BabelFish resultó muy simple, y no presenta la limitación de la solución con *behaviours*, por lo que es mas transparente. Sin embargo, no fue posible la implementación de un cliente para la consumición del servicio web `MSNStatusWS` ya que este debería residir en el repositorio XMethods, el cual es el único repositorio de Servicios Web que implementa el modelo de seguridad de servicios web para navegadores Gecko, propuesto por Netscape. Lamentablemente, no fue posible implementar dicho modelo en el repositorio local, a pesar de seguir las instrucciones de Netscape para este propósito.

5 Conclusiones

Como se explicó en la sección 4, ambas soluciones son muy simples de implementar en cuanto a la cantidad de código JS necesario para consumir los servicios Web, y ambas liberan al desarrollador de tener que conocer acerca de SOAP o XML para este propósito. Sin embargo, la solución propuesta por Netscape es mas clara, ya que centraliza en el repositorio de servicios web el problema de decidir cuales dominios pueden acceder a un servicio web determinado, a través del uso de un fichero xml. Sin embargo, se necesita un poco de trabajo aún en esta solución, ya que no fué posible implementar esta solución para el servicio web de monitoreo de mensajería MSN. En favor de la solución de Microsoft, se puede decir que es posible consumir un servicio web sin necesidad de hacerlo públicamente disponible en un repositorio que implemente el modelo de seguridad propuesto por Netscape, y que la implementación de ambos clientes funcionó sin problemas.

Bibliografía

1. BabelFish web service http://xmethods.net/ve2/ViewListing.po?key=uuid:E00104D5-2AC8-9DEA-EF4C-8BD920E1B4DD
2. dotMSN library http://members.home.nl/b.geertsema/dotMSN/index.html
3. JavaScript Bible D. Goodman and M. Morrison, John Wiley & Sons, Inc.; ISBN: B0001WPRY6; 5th edition (March 2004).
4. Microsoft Behaviours http://msdn.microsoft.com/library/default.asp?url=/workshop/author/webservice/overview.asp
5. Mozilla's Web Services Security Model. http://lxr.mozilla.org/mozilla/source/extensions/webservices/docs/New˙Security˙Model.html
6. MSN Protocol. http://www.hypothetic.org/docs/msn/index.php
7. Netscape Gecko http://devedge.netscape.com/central/gecko/
8. Special Edition Using SOAP J. Mueller, QUE; ISBN: B00012OWLU; (September 2001).
9. Web Service Behavior http://msdn.microsoft.com/library/default.asp?url=/workshop/author/webservice/webservice.asp
10. WSDL Proxying http://devedge.netscape.com/viewsource/2003/wsdl/01/
11. XML in a Nutshell, 2nd Edition E.R. Harold and W.S. Means, O'Reilly & Associates; ISBN: 0596002920; (June 2002) .