# 6 DEBATE REST-SOAP

En este capítulo se pretende dar una visión general del debate que ha existido en la Web entorno a la comparación entre REST y SOAP. Se exponen los artículos más importantes que se han escrito hasta el momento.

Desde que REST salió a la luz, siempre ha habido un debate en torno a su comparación con SOAP. REST fue rápidamente catalogado como alternativa a SOAP. Aún así, actualmente SOAP todavía posee el monopolio en el mundo de los Servicios Web. Ambos difieren en muchos aspectos comenzando por que **REST** fue concebido en el ámbito **académico** y **SOAP** es un estándar de la **industria**, creado por un consorcio del cual **Microsoft** formaba parte.

Las principales diferencias en el funcionamiento de ambos son:

- SOAP es un estilo de arquitectura orientado a RPC, mientras que para REST solamente existen los métodos de HTTP y está orientado a recursos.

- REST no permite el uso estricto de "sesión" puesto que por definición es sin estado, mientras que para SOAP, al ser orientado a RPC, los servicios Web crean sesiones para invocar a los métodos remotos.

- SOAP utiliza HTTP como un túnel por el que pasa sus mensajes, se vale de XML para encapsular datos y funciones en los mensajes. Si dibujásemos una pila de protocolos, SOAP iría justo encima de HTTP, mientras que REST propone HTTP como nivel de aplicación.

En el debate de comparación entre REST y SOAP, la mayoría de los desarrolladores de aplicaciones Web toman posiciones muy extremas a favor de uno u otro. Los afines a SOAP, suelen pronunciarse diciendo que SOAP es más flexible que REST a la hora de implementar servicios Web y muestran como un defecto de REST la restricción "sin estado", mientras que los adeptos de REST (también llamados Restafarians), critican la escasa transparencia de SOAP y opinan que hace las cosas más difíciles de lo que de verdad son, dicen que SOAP da "un paso hacia delante y dos hacia atrás". Además opinan que SOAP puede suponer un problema porque puede dar lugar a la creación de agujeros de seguridad en las implementaciones HTTP. .

Este tema ha sido ampliamente discutido en la red durante los últimos años. Ahora expondremos las opiniones de los autores más importantes que se han pronunciado al respecto.

## 6.1 PAUL PRESCOD

Paul Prescod es un programador independiente de las tecnologías basadas en el lenguaje de marcas. Trabajó en un grupo del W3C para escribir la familia de estándares de XML y escribió el libro más importante sobre la familia de estándares: *The XML Handbook.*

Fue uno de los primeros autores en estudiar la disertación de Roy Fielding y emitir su juicio respecto al debate REST-SOAP. A continuación expondremos dos artículos especialmente importantes.

### 6.1.1 Raíces del debate REST-SOAP [14]

Este artículo hace un resumen de lo que es REST, y su comparación con SOAP. Explica que WSDL puede convertirse en un cuello de botella. Además, incide en los beneficios de usar URI para tener un sistema de direccionamiento global.

Fielding desarrolló las ideas de REST pero no argumentó si REST o HTTP constituían unas bases suficientes para una arquitectura de servicios Web. Él nunca ha dicho si creía que era necesaria una arquitectura de servicios Web distinta de la arquitectura Web. Algunos han llegado a la conclusión de que no es necesaria. Nosotros sentimos que la arquitectura Web y REST son ya unas bases viables para la arquitectura de servicios Web.

REST enfatiza:

- Escalabilidad de los componentes de interacción.
- Generalidad de las interfaces.
- Independencia en el desarrollo de componentes.
- Sistemas intermedios para reducir la el tiempo de interacción, mejorar la seguridad, y encapsular los sistemas de herencia.

El primero, “escalabilidad de los componentes de interacción”, ha sido incuestionablemente logrado. La Web ha crecido exponencialmente sin degradar su funcionamiento. Una faceta de la escalabilidad se muestra en la variedad de clientes software que pueden acceder y ser accedidos a través de la Web.

El segundo objetivo, “generalidad de las interfaces”, es la clave que hace que REST plantee unas bases mejores que SOAP para el marco de trabajo de los servicios Web. Cualquier cliente HTTP puede hablar con cualquier servidor HTTP sin necesidad de configuración. Esto no le ocurre a SOAP. Con SOAP, deben conocerse los nombres de los métodos, el modelo de direccionamiento y los modelos de procesamiento de cada servicio en particular. Esto ocurre porque HTTP es un protocolo de aplicación mientras que SOAP es un marco de trabajo para el protocolo.

Fielding pide “independencia en el desarrollo de los componentes” porque es necesario a la hora de tratar con Internet. Las implementaciones del cliente y servidor pueden tardar en desarrollarse años e incluso décadas. Los antiguos servidores HTTP podrían no actualizarse y los nuevos clientes deben poder hablar con ellos. Los antiguos clientes

podrían necesitar hablar con los nuevos servidores. Diseñar un protocolo que permita esto es bastante difícil. HTTP se hace extensible por medio de las cabeceras, a través de las nuevas técnicas de generación de URI, a través de la capacidad para crear nuevos métodos y a través de la capacidad de crear nuevos tipos de contenidos.

El último objetivo de REST es la "compatibilidad con los sistemas intermedios". Los sistemas intermedios más famosos son los Web proxies. Algunos proxies hacen el papel de caché para mejorar el funcionamiento. Otros mejoran las políticas de seguridad. Otro tipo de sistemas intermedios muy importantes, son los gateways, que encapsulan sistemas que originalmente no son Web.

REST consigue esos objetivos aplicando cuatro constantes:

- Identificación de recursos
- Manipulación de recursos a través de sus representaciones
- Mensajes auto-descriptivos
- Hipermedios como el motor del estado de la aplicación

La identificación de recursos siempre se hace por medio de URI. HTTP es un protocolo centrado en URI. Los recursos son los objetos lógicos a los que les mandamos mensajes. Los recursos no pueden ser directamente accedidos ni modificados. En vez de eso, trabajamos con representaciones de ellos. Cuando usamos el método HTTP PUT para enviar información, se toma como una representación del estado en el que nos gustaría que estuviese el recurso. Internamente el estado del recurso puede ser cualquier cosa desde una base de datos relacional hasta un simple texto plano.

REST obliga a que los mensajes HTTP sean tan auto-descriptivos como sea posible. Esto permite que los sistemas intermedios puedan interpretar los mensajes y mejoren el comportamiento con los usuarios. Uno de los modos con los que HTTP realiza esto es estandarizar los métodos, las cabeceras y los mecanismos de direccionamiento. Por ejemplo los servidores caché conocen que por defecto GET es "cacheable" (porque no puede tener efectos colaterales al solicitar una representación de un recurso). Por el contrario POST es no-cacheable por defecto. HTTP es un protocolo sin estado, y si se usa correctamente, es posible interpretar cada mensaje sin conocer los mensajes que existieron anteriormente, ni los que le seguirán. Por ejemplo en vez de usar "loggin in" de la manera que lo hace FTP, HTTP envía la información nombre/contraseña en cada mensaje.

Finalmente. REST describe un mundo en el que cada hipermedio es el motor del estado de la aplicación. Esto significa que el estado actual de una aplicación Web particular debe estar capturado en uno o más documentos de hipertexto, residiendo en el cliente o en el servidor. El servidor conoce el estado de sus recursos pero no va almacenando los pasos de cada "sesión" de un cliente. Es el cliente quien conoce la misión que se necesita completar. Es responsabilidad del cliente ir navegando de recurso en recurso, recopilando la información que necesita y cambiando el estado cuando lo necesita.

Hay millones de aplicaciones Web que implícitamente hacen uso de las restricciones de HTTP. Hay una disciplina detrás de cada sitio Web escalable y que puede ser aprendida de varios estándares y documentos de arquitectura Web. También es cierto que muchos sitios Web comprometen uno o más de esos principios para hacer cosas que van en

contra de la arquitectura como puede ser almacenar los movimientos que hace un usuario en un sitio Web. Este almacenamiento es posible dentro de la infraestructura Web, pero influye negativamente en la escalabilidad porque transforma lo que es esencialmente sin conexión en algo basado en conexión.

Aunque REST y HTTP hace uso de muchas reglas y principios, algunas de ellas pueden colapsar. Por ejemplo si queremos hacer un sitio HTML de gran calidad, ¿Cómo debemos hacerlo? La respuesta, es que podemos cambiar y usar representaciones XML.

Los defensores de REST han llegado a creer que esas ideas son tan aplicables para los problemas de integración de las aplicaciones como para los problemas de integración del hipertexto. Fielding es bastante claro cuando dice que REST no es la solución absoluta para todo. Señaló que su diseño de las características no sería apropiado para otro tipo de aplicaciones de red. Nada más lejos, los que han elegido adoptar REST como modelo de servicios Web, sienten que al menos articula una filosofía de diseño con sus puntos fuertes, sus puntos débiles y áreas donde se puede aplicar.

Por el contrario, la arquitectura de servicios Web convencional ha sido conducida sin rumbo. Sus desarrolladores han hecho hincapié en algunas ideas pero han ignorado otras. Lo peor es que han olvidado adoptar las ideas más importantes que hacen que la Web funcione bien.

SOAP ahora no es muy útil. Después de su primera versión, hubo un periodo enfocado al crecimiento en la extensibilidad. Las cabeceras SOAP cada vez se hicieron más prominentes y sofisticadas. SOAP desarrolló un mecanismo para poder correr bajo múltiples protocolos de transporte. SOAP creció de ser un protocolo a ser un marco de trabajo para protocolos. En otras palabras, SOAP creció y cambió tan drásticamente que ahora los defensores de SOAP pueden despreciar una característica (el tipo de sistema de SOAP) que una vez fue considerada la principal razón de la existencia de SOAP. Sus objetivos han cambiado tan drásticamente que la palabra SOAP pasó de ser un acrónimo de “Simple Objet Access Protocol” a ser simplemente un nombre sin significado.

Los cambios más recientes se han hecho para responder a las críticas de los defensores de REST. Se señaló que SOAP abusa de HTTP y falla al integrarse correctamente en la arquitectura Web. Esos cambios todavía no han sido desarrollados en ninguna herramienta y puede que nunca lo hagan. Pero los cambios son por lo menos un signo de que hay esperanza.

Una de las mayores diferencias entre el punto de vista de REST y de SOAP es que REST ve la Web como un sistema de información por sí misma y dice que los otros sistemas deberían estar integrados en el sistema de información por medio de gateways.

Lo más importante de un espacio de información Web compartida es la idea de URI. Los servicios Web basados en SOAP nunca han adoptado la noción de la Web como un espacio de información compartida y por ello nunca han adoptado completamente el modelo Web basado en URI. Ellos, por el contrario, siempre han asumido que cada aplicación debe definir su propio espacio de nombres.

Los estándares de servicios Web nunca fueron diseñados para usar URI como identificadores de recurso y de hecho WSDL hace que esto sea esencialmente imposible. No hay ninguna manera de decir en WSDL que un elemento particular de un mensaje SOAP representa una URI y que la URI apunta a un servicio Web. Por lo tanto cada WSDL describe solo un recurso Web y no proporciona ningún método para describir enlaces a otros recursos. SOAP y WSDL usan URI solo para direccionar un punto final de gran importancia que gestiona todos los objetos dentro de el. Por ejemplo, para enviar un mensaje a una cuenta en particular, debe hacerse algo como esto:

```
bank = new SOAPProxy("http://.....")
bank.addMoneyToAccount(account 23423532, 50 dollars)
```

Para conseguir el balance asociado a una cuenta, debe hacerse algo como esto:

```
bank = new SOAPProxy("http://.....")
bank.getAccountBalance(account 23212343)
```

Hay que hacer notar que la cuenta por si sola no es directamente direccionable. No tiene URI, solo un número de cuenta. Sólo se puede acceder a la cuenta a través del servicio. También debemos darnos cuenta de que los números de cuenta del banco constituyen un espacio de nombres, distinto del espacio de nombres de las URI de la Web. En la versión del servicio centrada en la Web, las cuentas deberían ser individualmente direccionables y cada una debería tener una URI.

Ahora vamos a hacer una analogía. Supongamos que estamos viviendo temporalmente en un hotel. El hotel puede no tener conexiones telefónicas directas con el exterior. Para llamar a una habitación debemos contactar con el operador primero (esto es como contactar con el punto final de SOAP) y después pedir que nos conecten con la habitación. Ahora imaginemos que hay un servicio externo al que queremos acceder. Es un servicio automatizado de Horóscopo. Intentamos acceder al servicio pero cuando nos preguntan el número de teléfono, nos damos cuenta de que no solo hay un número que podamos proporcionar. El servicio debe contactar primero con el operador y después el operador pasará la llamada hasta nosotros. Obviamente, un ordenador en el otro extremo de la comunicación no es capaz de entender esto, y el operador no sabrá a quien pasar la llamada.

Si todo el mundo viviese en un hotel como este, el servicio de horóscopo sería prácticamente imposible. Una aplicación particular del teléfono dejaría de existir.

Hay que darse cuenta de que el problema no es tan obvio en el diseño de cada sistema por separado. Sólo cuando tratamos de unir ambos sistemas es cuando desearíamos haber usado la sintaxis del estándar internacional de direccionamiento telefónico. Los

puntos finales de SOAP son como el operador. Los objetos con los que trabajan (recibos, cuentas de banco) son las habitaciones del hotel. Los datos están a la merced de los puntos finales de SOAP. Si las interfaces de los servidores y clientes no están perfectamente alineadas, no se pueden comunicar. Una tercera parte que sea una aplicación que intenta integrarlas, tendría que extraer explícitamente la información de un servicio y ponerla en el otro. Esto no es una buena solución porque la tercera parte tendría que ser la responsable del intercambio de información. Idealmente sería preferible introducir los dos elementos de datos de tal manera que permitan a cada uno ser informado de los cambios de estado del otro.

En contraste con esto, HTTP recomienda fuertemente el uso de URI como identificadores. De hecho, la arquitectura Web anima a usar URI que puedan ser resueltas en documentos. Comúnmente esto significa URIS “http:”. URI es el elemento que unifica el concepto de HTTP. Podemos inventarnos nuestras propias cabeceras e incluso en algunos casos nuestros propios métodos. Podemos usar cualquier tipo de contenido en el cuerpo (o ninguno) pero *siempre* debemos enviar una URI como el recurso objetivo de cada petición HTTP.

Esta elección tiene consecuencias reales. Si dos servicios HTTP se pueden poner de acuerdo en la representación de una información (por ejemplo RSS o SAML) entonces esos dos servicios pueden comunicarse sobre esos recursos de información simplemente pasando URI a los recursos. Por ejemplo, si tenemos un documento RSS representando la información de un blog, podemos introducirlo en un sistema de visionado de blogs como Meerkat simplemente manejándolo como una URI. Meerkat no necesita saber cual es el mecanismo de direccionamiento porque está estandarizado. Meerkat tampoco necesita saber los métodos que usamos para conseguir la información porque el método HTTP “GET” está implícito. También podemos imaginar el servicio contrario. Podemos pedirle a Meerkat que use un RSS haciendo un PUT a una dirección que especifiquemos.

En muchos aspectos, el estandarizado modelo SOAP difiere de las ideas básicas de lo que llamamos la filosofía XML. La familia de estándares XML presume de que toda la información útil está disponible en un formato XML en una dirección estandarizada. Las tecnologías de servicios Web se usan casi siempre para configurar diferentes espacios de direcciones que casi nunca son direccionables mediante URI. Como ejemplos de esos distintos espacios de direcciones están UDDI y Microsoft's .NET My Services. Esos espacios de direcciones son fundamentalmente incompatibles con las tecnologías XML centradas en URI.

Hay una razón para el fallo de SOAP en el direccionamiento. Si SOAP admitiese que la sintaxis de direccionamiento para SOAP fuese URI, eso sería equivalente a decir que SOAP está diseñado sólo para ser usado en la Web. Sin embargo, los defensores de SOAP son muy claros en el hecho de que los protocolos Web y los modelos de direccionamiento Web, solamente sirven como transporte para SOAP. La Web es un conductor de taxi y su única función es llevar SOAP de un lado a otro. El mensaje SOAP puede salir del taxi y entrar en otro vehículo para proseguir otra etapa de su viaje. En términos técnicos, SOAP hace un túnel sobre la Web. Esto no es para lo que se diseñó la Web.

El punto de vista de REST es que la Web por si sola es un sistema de información increíblemente escalable y bien diseñada. Fue explícitamente diseñada para integrar distintos sistemas de información. Pero la Web lo hace a su manera, uniéndolo todo en un único sistema de nombres y recomendando el uso de un único protocolo.

La Web es un tipo de mundo unificado, global y donde todo el mundo mapea sus sistemas con el mismo modelo, usando el mismo esquema de direccionamiento, un reducido número de protocolos globales que todo el mundo entiende, formatos para la información compartida y vocabularios XML. No tiene mucho sentido adoptar la parte menos potente de la infraestructura Web (la sintaxis del método HTTP POST) e ignorar la parte más potente: El espacio de nombres URI y el modelo de manipulación de recursos estandarizados.

La relación entre REST y SOAP/WSDL es similar a la que existe entre XML y SGML. XML era prescriptita: "debes usar Unicode". SGML era descriptiva "puedes usar cualquier carácter pero debes declararlo". XML: "Debes usar URI para los identificadores". SGML "Puedes usar cualquier clase de identificador (nombre de fichero, clave de la base de datos…)".

SOAP ignora cosas que hemos aprendido de XML y la experiencia de la Web. Aún así, SOAP continúa ganando adeptos en la industria y el comercio.

Los seguidores de REST deben ver SOAP como una pregunta más que como una respuesta. La pregunta es "¿Cómo hacemos que la Web haga las cosas de los Servicios Web? El hecho de que la gente sienta que necesita SOAP indica que no están satisfechas con HTTP. Por tanto hay que evaluar las cosas que se supone que SOAP añade a HTTP.

Hay algunos temas que surgen cuando los adeptos a REST hablan con los adeptos a SOAP:

*Asincronía:*

Esto significa diferentes cosas para la gente, pero la acepción más común es que dos programas que se comunican (por ejemplo cliente y servidor) no deben estar en constante comunicación mientras que uno de ellos esté realizando cálculos de gran peso. Debe haber alguna manera para que una parte de la comunicación pueda avisar a la otra. SOAP no soporta esto directamente pero puede ser usado de una manera asíncrona si el transporte que utiliza es asíncrono. Esto todavía no está estandarizado pero algún día lo estará. Hay una gran variedad de aproximaciones para hacer HTTP asíncrono. Se ha propuesto la estandarización, una de esas especificaciones es "HTTPEvents".

*Enrutamiento:*

Los mensajes HTTP están enrutados desde los clientes hasta los servidores proxies. Esto es un tipo de enrutamiento controlado por la red. También hay algunos casos en los que tiene sentido que el cliente controle el enrutamiento explícitamente, definiendo el camino que debe seguir un mensaje entre los nodos. Algunas investigaciones de REST están trabajando con variantes del enrutamiento de SOAP para esto.

*XML:*

Actualmente estamos en una situación donde XML es un requisito para la creación de nuevas especificaciones tanto si añade valor como si no. El grupo de trabajo que está estandarizando SOAP dentro del W3C se llama "XML Protocol Working Group". En el mundo actual es probable que las tecnologías pre-XML se reinventen al estilo XML sólo porque está de moda.

*Fiabilidad:*

La fiabilidad es otra palabra que significa diferentes cosas para la gente. Para la mayoría de la gente significa entregar un mensaje una y solo una vez. Esto es relativamente fácil de conseguir con HTTP, pero no es una característica implícita. Lo que podría decirse que es peor es la cuestión de que la fiabilidad garantizada va en contra de las redes a gran escala y tratar de conseguirla puede degradar el funcionamiento.

*Seguridad:*

Aunque HTTP tiene características de seguridad, es útil compararlas con las características que surgen como parte de WS-Security.

*Extensibilidad del modelo:*

SOAP tiene un modelo de extensibilidad que permite al creador de un mensaje SOAP ser muy explícito sobre si comprender una parte del mensaje es opcional o necesario. También permite a las cabeceras que sean interpretadas por sistemas intermedios en particular (como proxies o cachés). Hay una extensión de HTTP con muchas de estas ideas (de hecho la versión de SOAP probablemente esté basada en la versión HTTP) pero no es tan conocida ni está sintácticamente clara.

*Descripción del servicio:*

SOAP tiene WSDL pero HTTP no tiene nada parecido porque REST es un modelo centrado en documento. La descripción de servicios REST puede surgir de los lenguajes esquemáticos y en particular de los lenguajes de esquema semántico como DAML. También sería posible usar la siguiente versión de WSDL de manera adaptada como un tipo de lenguaje de descripción para los recursos HTTP.

*Familiaridad:*

Las tecnologías de servicios Web están diseñadas para ser tan flexibles que podemos usarlas siempre que estemos acostumbrados a programar. Los que no tienen experiencia con la programación en red pueden usar las variantes de RPC y usar las funciones como si fuesen locales. Los que tienen experiencia en el intercambio de mensajes, pueden usar unidireccionalmente SOAP para envolver sus mensajes. Los defensores de REST pueden usar una extensión de HTTP. Ninguno de esos tres grupos de programadores podrá interoperar con los demás.

**Mirando hacia el futuro**, hay tres posibles futuros. En uno, las críticas de REST sobre la filosofía de SOAP resultan ser incorrectas. El modelo de SOAP volvería a ser suficientemente bueno y el uso de REST se limitaría a ser el hipertexto de la Web. En un segundo escenario, la metodología de SOAP revelaría gradualmente sus defectos hasta que fuesen intratables. REST o algo parecido se convertiría en la arquitectura dominante y SOAP estaría relegado a tener un papel mucho menos importante. En el tercer escenario, encontraríamos una manera de que ambas arquitecturas trabajasen juntas. Cada una podría tratar diferentes problemas e incluso podrían interoperar.

**¿Puede ganar un solo protocolo?** Los negocios electrónicos van a necesitar algo más que software orientado a RPC o a mensajes. Todos los negocios tendrán que estandarizar sus modelos de direccionamiento y compartir una interfaz común con el resto de negocios. SOAP no hace esto por si mismo, en este aspecto, hace más daño que la ayuda que ofrece.

Para que los negocios puedan interoperar sin necesidad de programar manualmente los enlaces con otros negocios, necesitarán estandarizar los protocolos antes que inventar nuevos. Tendrán que estandarizar un modelo de direccionamiento antes que inventar uno nuevo. REST propone un alto grado de estandarización. Los servicios Web estarán estancados hasta que se estandaricen en una arquitectura unificada, y REST es esa arquitectura. Si esto es cierto, entonces no importa si gana REST o SOAP, porque SOAP viviría dentro de la arquitectura REST para servicios Web.

**¿Pueden coexistir REST y SOAP?** Recientemente SOAP ha tomado una dirección que le acerca a REST, pero también conserva su núcleo basado en RPC. La cuestión por tanto no es si SOAP y REST pueden coexistir. Pueden. Pero si la visión del mundo basado en REST es correcta, entonces todo el proyecto SOAP/WSDL sería un error. ¿Por qué necesitamos un marco de trabajo de protocolos para tunelar nuevos protocolos de aplicación sobre cualquier transporte, cuando tenemos muchos protocolos de aplicación?

Una respuesta es que SOAP está ganando en cuanto a seguridad, enrutamiento y otros aspectos que no pueden ser direccionados con HTTP, o por lo menos no tan robustamente. Un escenario horrible para la futura Web tendría un conjunto de características disponibles para SOAP (seguridad, enrutamiento…), y otro conjunto distinto para HTTP (direccionamiento, métodos conocidos…), y ninguna manera de conseguir lo mejor de ambos.

Una posible solución sería reinventar un protocolo REST (orientado a URI y genérico) en lo alto de SOAP. Lo llamaremos “SHTTP”. Sería una solución potente pero existen muchas dificultades técnicas. Dada una URI en particular, debe existir un y solo un modo de mapear la URI en una representación. Por tanto, se requeriría un nuevo esquema URI. Este nuevo esquema URI sería incompatible con la gran variedad de clientes HTTP que hay desarrollados y podría tardar años en propagarse.

Además, HTTP se usa constantemente para repartir datos binarios, pero SOAP no es bueno en ese aspecto. Los motores HTTP son más eficientes que las herramientas que deben parsear XML. Hay otro problema por el que los mensajes SHTTP no podrían

correr en lo alto de SOAP, SOAP suele ir encima de HTTP. Esto es ineficiente, confuso y puede desembocar en agujeros de seguridad donde el mensaje de la capa SHTTP es muy distinto al que envía la capa HTTP.

Pero lo más importante, aunque superemos estas incoherencias sintácticas, no responderíamos a ninguna de las cuestiones que surgieron en la crítica de REST a SOAP. ¿Necesitamos muchos protocolos o sólo uno?, ¿Necesitamos muchos modelos de direccionamiento, o sólo uno?, ¿Podemos conseguir la interoperabilidad con un sistema que usa múltiples protocolos y modelos de direccionamiento?, ¿Es realmente algo idealizado conseguir unificar todo en un simple protocolo y un modelo de direccionamiento?, ¿Debemos exponer servicios o eliminar los límites de los servicios y exponer recursos?, ¿Deben ser prescriptivos o descriptivos los servicios Web estándar?, ¿Se puede manejar la herencia sólo con gateways?, ¿Cambiarían las respuestas a esas preguntas si integrásemos las aplicaciones dentro de organizaciones en vez de tener límites organizacionales?

Cada parte cree conocer las respuestas. Sólo el tiempo demostrará qué visión es la correcta.

### 6.1.2 Algunos pensamientos sobre la seguridad de REST-SOAP [15]

Este artículo que exponemos a continuación, hace una comparación en materia de seguridad entre REST y SOAP.

SOAP es una especificación intrínsecamente compleja. Normalmente se sitúa en lo alto de HTTP y por tanto hereda cualquier bug o agujero de seguridad en la implementación de éste. SOAP además esta diseñado para saltarse los firewalls como si fuese HTTP, suele decirse que: “SOAP pasa los firewalls como un cuchillo a través de mantequilla”. Partiendo de la base de que SOAP hace varias cosas mal, la primera de ellas es hacer un túnel en los firewalls usando HTTP.

Filtrar SOAP es algo bastante complicado si nos basamos en el conocimiento de las especificaciones. SOAP usa el mismo puerto que HTTP. El mundo ya ha experimentado los agujeros de seguridad que se producen al correr múltiples servicios sobre el mismo puerto con el protocolo RPC de Microsoft. SOAP usa el método HTTP POST cuando realmente debería usar una extensión de este método. La cabecera de SOAPAction está ahora obsoleta, la única manera de reconocer SOAP es haciendo un parsing XML en el firewall. Suponiendo que hacemos esto, ¿Cómo decidimos si el mensaje debe pasar el firewall? SOAP no tiene ni un modelo de direccionamiento uniforme, ni una estructura interna fiable. A veces hay una cabecera, otras veces no. A veces el cuerpo está codificado como RPC, otras veces no. A veces aparece el nombre de un método, otras veces no. Roy Fielding (uno de los inventores de HTTP) y Jim Whitehead (de WebDAV) ya se plantearon anteriormente este problema.

#### 6.1.2.1 UN EJEMPLO

Digamos que un troyano o un virus están rondando por Internet. El administrador de un sistema puede decidir monitorizar los datos que vienen del exterior. SMTP y HTTP tienen bien definidas las maneras de transportar los datos y usando unos patrones simples, se pueden detectar los virus. Por tanto, es fácil monitorizar todo el tráfico SMTP y HTTP que cruza un firewall.

Es trivial excepto si el tráfico que cruza el firewall está codificado con SOAP. La versión SOAP debe tener un paso de decodificación extra porque SOAP tiene (al menos) tres maneras diferentes de transportar datos. Esto no sería un problema si SOAP usara su propio puerto porque en ese caso sería fácil desactivar el puerto del mismo modo que se hace con el de SSH o FTP. Desafortunadamente SOAP hace un túnel sobre HTTP y sobrecarga este puerto.

Ahora digamos que seis meses antes estuvimos desarrollando una aplicación P2P para compartir datos astronómicos. La llamaremos "starster". Tenemos tres posibles protocolos para usar. Una opción es usar un protocolo propio en un puerto propio. Esto es correcto desde el punto de vista de la seguridad porque si al administrador no le parece correcto el uso de este programa, puede ir al firewall y cerrar el puerto.

Una segunda opción es usar SOAP. En este caso, starster usará su propio protocolo como una adaptación de SOAP (un conjunto de métodos de este) y SOAP hará un túnel a través de HTTP. Esto significa que el administrador del sistema no podrá saber por defecto si starster se está ejecutando en el sistema. Tampoco podrá evitar el uso de este programa. Necesitaría detectar un espacio de nombres XML particular definido en un documento XML particular y que va encapsulado en un sobre SOAP, que a su vez va encapsulado en un mensaje HTTP sobre TCP.

Una tercera opción es usar HTTP, pero usándolo de acuerdo con la especificación HTTP, sin codificaciones extra. En este caso el tráfico de starter continuaría fluyendo por el firewall porque es indistinguible del tráfico Web. Pero lo más importante es que sería tan seguro como el tráfico Web porque el administrador del sistema podría usar un filtro de tráfico estándar para evitarse los problemas del virus.

Un protocolo de aplicación consiste principalmente en el direccionamiento de los recursos de datos y su manipulación. La manipulación se realiza mediante métodos. En HTTP, algunos métodos están definidos para escribir y modificar datos y otros sólo para leer datos. Esto significa que salvo que se haga un túnel, los firewalls se pueden configurar para que una parte de la red tenga sólo acceso a lectura. En los archivos de "log" se pueden buscar fácilmente las escrituras que puedan contener datos corruptos. El control de acceso de lectura/escritura puede ser separado.

SOAP no hace esta distinción. Cualquier método puede ser de lectura o de escritura. Incluso WSDL no permite hacer esta distinción. De esta manera una potente herramienta de filtrado no sirve de nada. En general, los administradores del sistema pueden leer los ficheros de "log" de una gran variedad de servicios basados en HTTP porque todos son parecidos: GET, PUT, POST, DELETE y URI. Por otra parte, SOAP permite mostrar los mensajes casi de cualquier manera. Está diseñado para tener una forma libre. Los mensajes de diferentes servicios pueden ser totalmente distintos.

Un ejemplo trivial, GET/getHistoricalStockQuotes?MSFT le dice a una persona responsable en seguridad: “Ok, es un GET, aunque sea un virus no puede modificar el servidor. Probablemente devuelva algún tipo de reporte del historial de cuota. Si hay un túnel o un bug no es mi problema, será culpa del programador.”

Cuando esta misma persona ve getHistoricalStockQuotes("MSFT") dice: “Probablemente devuelva el fichero de cuota. Pero, ¿Puedo estar seguro de que no va a modificar nada en el servidor? Puede que esté creando un nuevo objeto. De ser así, ¿Quién tiene permiso para crear ese objeto?, ¿Puede un hacker malicioso llenarlo de datos hasta que el servidor desborde la memoria? Mejor voy a leer la documentación para estas cosas, creo que trataré de encontrar al programador para asegurarme de que lo comprendo correctamente.”

Por supuesto que ambos son igual de simples: devuelven un reporte. Pero uno es muy explícito al asegurar que no modificará el estado del servidor, mientras que el otro no.

#### 6.1.2.2 HTTP/REST, LA WEB TIENE UN ESPACIO DE NOMBRES UNIFICADO

Para nuestras intenciones, los protocolos de aplicación son para direccionar y manipular datos y recursos. En la Web, todos los recursos forman parte de un sistema de direccionamiento global, URI. Una de las cosas buenas de usar URI es que proporciona una manera muy tangible para poner los permisos. Por ejemplo, podemos imaginar como se dan o quitan los permisos de un fichero de una orden de compra, y sería equivalente hacerlo con un recurso de orden de compra Web. Esta es una característica Web estándar y tiene sus raíces en las técnicas de permisos que la gente ha usado durante décadas.

La gran diferencia entre un recurso y un fichero es que el recurso es virtual: por ejemplo puede estar representado por una fila en una base de datos. Los permisos los aplicamos a la orden de compra virtual, no a un fichero.

SOAP no ha unificado el concepto de recurso y cada aplicación SOAP se inventa su propio mecanismo de direccionamiento. Al contrario que HTTP, los objetos de datos de SOAP normalmente no están disponibles por medio de una URI. Esto significa que no hay nada a lo que colocar permisos o restringir el acceso. Cada desarrollador de una aplicación tendrá que inventarse su propio mecanismo de direccionamiento e imaginarse como asociar los permisos por sí mismo.

#### 6.1.2.3 LA SEGURIDAD DE SOAP ES ERRÓNEA

SOAP pone gran parte de la responsabilidad sobre seguridad en manos del desarrollador, pero la documentación existente no lo prepara para esta tarea. Consideremos esta cita: “Para la seguridad, usa el protocolo Secure Sockets Layer (SSL) y las técnicas de autenticación Web”. Los defensores de SOAP creen que si tiene algunas características de seguridad, “tiene seguridad”.

Si leemos la documentación sobre seguridad en SOAP que tiene Microsoft, no se indica como hacer un servicio seguro. En vez de eso, se describe como limitar el acceso a ciertos dominios. Pero la función de los servicios Web es que podamos poner los servicios en la Web pública de la misma manera que ponemos sitios Web. La documentación menciona los firewalls pero no instruye a los programadores sobre como usarlos correctamente, teniendo un puerto específico para los servicios SOAP.

#### 6.1.2.4 SOAP ES NUEVO Y TODAVÍA NO ESTÁ TESTADO

La gente tiene una gran confianza depositada en el servidor Web Apache. IIS está considerado menos robusto, pero lo suficiente como para tener millones de sitios funcionando cada día. Consideremos el tiempo que tomó llevar a la seguridad hasta este punto. De momento no podemos confiar en que las herramientas de servicios Web hayan alcanzado este nivel de fiabilidad y seguridad. El mundo de los servicios Web debe tomarse las cosas con más calma y considerar las implicaciones de la seguridad de sus invenciones con mucho más rigor.

## *6.2 ROBERT McMILLAN*

Robert McMillan es un periodista y experto en Servicios Web que escribió varios artículos sobre el debate REST-SOAP. Lo más interesante de este autor es que además de las características técnicas del debate REST-SOAP, se ocupa de otros factores como la publicidad y promoción de las herramientas REST.

### 6.2.1 REST, sustituto de SOAP-RPC [16]

Pequeño resumen de lo que es REST y su relación con SOAP. Indica que una de las cosas por las que REST no se está extendiendo rápidamente es la falta de promoción.

REST dice que SOAP-RPC no está bien diseñado para compartir datos basados en XML u otros documentos sobre Internet de una manera eficiente porque esos estándares no son más que una repetición de DCOM y CORBA con algunos cambios.

#### 6.2.1.1 EL RESTO DE LA HISTORIA

De acuerdo con los adeptos a REST, la Web ya tiene una arquitectura para sistemas distribuidos que es robusta, segura y ampliamente probada. Está basada en formatos comunes, un estándar de protocolos, y lo más importante, un sistema de nombres estándar: URI.

Específicamente, la arquitectura REST está basada en componentes probados que todavía se usan para crear servicios Web: XML, HTTP y URI. La idea es que en vez de exponer las APIs usando un modelo SOAP-RPC, usemos los métodos que han sido desarrollados para HTTP: GET, POST, DELETE y PUT. Como ejemplo, podemos usarlos para mantener un formato de datos XML organizado mediante URI.

Un sitio Web bastante importante, RESTwiki, describe por qué REST es tan bueno para compartir en Internet los datos y las colecciones de documentos. "No importa que servidor Web tenga, ni el cliente Web que acceda a él, si se ve una URL como http://conveyor.com/RESTwiki/, sabemos que podemos solicitar información usando los métodos HTTP, sin necesitar ninguna coordinación previa. También podremos manipular su contenido con PUT y POST (aunque nos responderán si es posible o no, y las razones de esta decisión)".

#### 6.2.1.2 EL FUTURO PARA REST Y SOAP-RPC

Una de las razones por las que REST no ha sido ampliamente aceptado todavía es que nadie lo está promocionando. Aunque las ideas de REST llevan un par de años rondando, no existe ni un consorcio REST, ni un estándar REST para desarrollar. Simplemente representa una manera diferente de pensar acerca de los servicios Web, aunque tiene las ventajas de que el funcionamiento, la seguridad y el espacio de nombres llevan probándose en Internet desde hace años.

¿Se podría decir que es REST contra SOAP? No exactamente, dicen los seguidores de REST. SOAP tiene un papel muy importante en el contexto de REST. Algunos adeptos a REST no son muy optimistas sobre su futuro porque SOAP ha llegado a convertirse en sinónimo de RPC, y la gente empieza a asociar servicios Web con SOAP.

### 6.2.2 Una aproximación a los Servicios Web basada en REST [17]

Entrevista a múltiples personalidades, de la que se obtiene un resumen de lo que es REST. Cabe destacar que al final del artículo se habla del futuro de REST desde un punto de vista de la aceptación comercial que pueda tener. Se discute si es necesario incluir REST en las herramientas de desarrollo para conseguir su consolidación.

¿Es complicado el desarrollo de Servicios Web? Un pequeño grupo de influyentes desarrolladores Web piensa que sí. Estos desarrolladores proponen una nueva aproximación más simple que el modelo SOAP, que está favorecido por desarrolladores de herramientas como BEA Systems, IBM y Microsoft. Esta nueva aproximación arquitectónica se llama Representational State Transfer (REST). Dicen que los resultados son más escalables.

Entre los más destacados partidarios de REST, se encuentran Roy Fielding perteneciente a la fundación Apache y Sam Ruby, un desarrollador senior y gurú de los servicios Web en IBM (aunque IBM no es afín a REST). Los desarrolladores de

Amazon y Google han experimentado con REST para crear interfaces a sus populares sitios Web.

#### 6.2.2.1 EL TRABAJO CON REST

REST confía en un solo protocolo de aplicación (HTTP), en URI y en un formato de datos estandarizado a través de XML. Emplea métodos HTTP establecidos como son GET y POST. En vez de crear un estándar, una manera que entiendan las máquinas para descubrir y usar componentes de aplicación en sistemas remotos (la manera que usa SOAP para los servicios Web), los desarrolladores de REST usan URI para crear un terreno común de manera que las aplicaciones puedan usar HTTP y XML para compartir datos. Los desarrolladores de REST usan documentos XML en vez de llamadas a métodos de aplicación para decirles a los programas distribuidos como usar los datos.

Los adeptos a REST dicen que usando el protocolo SOAP para acceder a las funciones de programas remotos directamente es un fallo del mismo tipo que los problemas de interoperabilidad que sufrían las antiguas arquitecturas de programación distribuidas como DCOM y Common Object Request Broker Architecture (CORBA).

Los problemas de seguridad también afectan a SOAP, dice Mark Baker, un consultor independiente de arquitectura Web y uno de los administradores de un sitio de recursos para desarrolladores. Como los firewall no comprenden el significado de los mensajes de los servicios Web basados en SOAP, nunca dejaran pasar esos mensajes.

Los mensajes REST no tienen este problema, dice Baker, porque solamente usan operaciones especificadas en el estándar HTTP (operaciones bien conocidas por los administradores y las aplicaciones firewall). Los comerciantes Web están solucionando este problema desarrollando estándares de seguridad en los servicios Web y productos, de la misma manera que se desarrollan firewalls y estándares de seguridad para HTTP.

#### 6.2.2.2 REST EL MEJOR

Antes de decidir usar REST para su servicio Web, Thomson consideró usar SOAP. Los desarrolladores eligieron REST porque ofrecía un mejor funcionamiento, fiabilidad y escalabilidad que SOAP.

Usando REST Thomson ahora puede usar documentos de datos de distinta fuente más fácilmente. Thomson dijo: “Antes de usar esta tecnología, teníamos que escribir una solución específica para cada fuente de datos, ahora lo podemos crear rápidamente usando REST”.

La aproximación centrada en documentos de REST se hizo particularmente apropiada cuando los usuarios compartían datos. Creaban documentos XML, y el GTS manejaba el resto: introduciendo documentos XML en el sistema de Thomson, los datos se hacían disponibles. “Aunque el sistema sólo puede manejar un proceso al mismo tiempo, el

usuario puede enviar cualquier número de peticiones". "El GTS maneja las tareas de mantenimiento de recursos, como la prioridad de los trabajos y el balance de cargas, de esta manera el conjunto del sistema se hace eficiente".

#### 6.2.2.3 UN SOÑADOR REAL

La experiencia de los usuarios de REST ha sido tan prematura como positiva, pero hay una gran carencia de herramientas. Esto es un gran obstáculo para la difusión. Todavía no existen grandes desarrolladores de aplicaciones que hayan adoptado REST, aunque parece que se lo están planteando seriamente. "REST tiene algunas grandes características que estamos examinando." Dijo David Orchard, director técnico de BEA.

Todavía hace falta que los diseñadores de herramientas adquieran un compromiso con REST para que éste despegue totalmente. Productos como Microsoft's Visual Studio .Net o IBM's WebSphere, automáticamente producen servicios Web basados en SOAP.

"Desde la perspectiva de un producto, REST es casi invisible", dijo Ronald Schmeltzer, un analista señor. "Si los adeptos a REST quieren triunfar, deben introducirlo en las herramientas que crean o consumen servicios Web."

Baker no está de acuerdo. Él dice que virtualmente cualquier herramienta HTTP podría servir para desarrollar servicios Web REST. "Hay un montón de herramientas REST disponibles, simplemente que la gente no sabe que existen", el dijo.

Las mismas herramientas que crean los Java Servlets podría usarse para construir servicios Web basados en REST, dijo Baker. "Siguen las especificaciones HTTP, y por consiguiente, implícitamente están compliendo las restricciones del estilo REST", el dijo.

BEA sugirió que REST debería coexistir con SOAP algún día porque los desarrolladores buscan múltiples técnicas para los estilos de servicios Web. "A veces hay más de una manera de hacer las cosas", él dijo.

## *6.3 ROY HOOBLER*

Roy Hoobler es el director de Tecnologías de Internet para la empresa Net@Work y tiene una visión muy simplificada del debate REST-SOAP.

### 6.3.1 REST puede acabar con SOAP [18]

Entrevista a Roy Hoobler en la que éste expresa su visión peculiar de REST. Desde su punto de vista con REST podemos solicitar información y recibirla, pero ahí es donde termina todo. Dice que de cara a realizar esta función, REST es mucho más sencillo y útil que SOAP, pero no podemos realizar Servicios Web que precisen procesamiento de datos.

¿Existe una manera simple de explicar REST?

Sí, viendo el ejemplo de Amazon. Si vamos a construir una aplicación XML, tendremos un fichero XML en una máquina. Sin embargo, en la mayoría de las aplicaciones XML, no tiene por qué ser el caso, porque podemos acceder al archivo XML a través de una URL válida. Por ejemplo, podemos ir a una página Web, conseguir un fichero XML y "parsearlo" en nuestro ordenador.

Si está en una base de datos, debe ser una base de datos local. No podemos conectarnos directamente a la base de datos de Amazon y ellos no van a dejarnos. Por tanto, deben tener un servicio SOAP, o deben realizar una búsqueda a través de HTTP, que es lo que realmente están haciendo. Lo que están haciendo (con REST) es construir un documento XML "al vuelo" y devolver así los datos vía HTTP. Podemos usar esa información en el sistema, almacenarla y procesarla. Hay muchas cosas que podemos hacer sin necesidad de SOAP ni de programar nada.

¿Es realmente tan simple?

Es un proceso muy simple. Si alguien tiene un archivo XML y nos da la URL, entonces podremos descargar el contenido. La programación queda prácticamente fuera. Alguien puede subir un fichero de texto y yo puedo descargarlo a través del navegador. Es un poco más complicado que eso, pero no demasiado.

SOAP es bastante lento comparado con REST, porque acarrea una mayor carga de comunicación. Cuando usamos REST, estamos cogiendo información como si estuviésemos buscando una página Web de un sitio. La conseguimos rápidamente. Con SOAP, hay mucha más información transaccional detrás de cada operación.

¿Es REST una metodología en vez de una especificación?

Es una técnica. Otros sitios que la han usado mucho son los blogs. Ofrecen lo mismo, pero usan un método llamado RSS (Resource Site Summary). Ofrecen casi lo mismo pero está en formato RDF (Resource Site Summary), que es un formato XML. Por ejemplo, si tenemos una lista de enlaces en nuestro sitio Web que queremos compartir con alguien, todo lo que tenemos que hacer es proporcionarle una URL desde donde puede descargar la información en RDF. Es algo parecido a lo que REST y Amazon hacen. Es una suscripción de servicio barata. No hay que escribir un servicio Web, simplemente vamos allí y cogemos el contenido.

Amazon usa actualmente un Servicio Web al estilo REST para compartir la información del precio de los productos con sus compradores. ¿Qué debería hacer SOAP para realizar la misma función?

Lo primero que debería hacer es implementar un servidor SOAP. Tendrían que desarrollarlo ellos mismos, lo que puede ser algo complicado. Una vez que tienen el servidor SOAP, tendrían que construir clientes SOAP para comunicarse con el servidor. Habría que realizar un gran trabajo de programación, y la mayoría de la gente todavía no ha usado SOAP. Con tecnologías como REST o RSS, toda esa programación no existe, y los servicios Web no precisan de demasiado esfuerzo.

Si es tan simple, ¿Por qué no se ha puesto tan de moda como SOAP?

No es tan potente. Con REST, solamente estás accediendo a contenido. No hay mucho procesamiento. Podemos realizar operaciones mucho más complejas con SOAP. Si Amazon necesitara enviar mucha información, y después hubiese que procesarla, SOAP sería lo más recomendable. REST no ha calado en el mundo de los negocios porque actualmente se necesitan grandes servidores de interacción y REST no es lo suficientemente potente.

Un ejemplo de ello:

Si estamos usando REST, podemos ir a un sitio Web que proporciona una lista de venta de tickets, pero no podemos modificar los tickets. Si estuviésemos usando SOAP, podríamos manipularlos y enviar el resultado de nuestros cálculos. Digamos que conseguimos los valores de un sitio, y queremos avisar de que son erróneos. Si quisiésemos enviar de vuelta algo que dijese que es erróneo con REST, no podríamos. Con REST podemos solicitar información y recibirla, pero ahí es donde termina todo.

¿Cómo de fácil o difícil resulta aprender a trabajar con REST respecto a SOAP?

Para trabajar con SOAP, necesitamos algún tipo de infraestructura como la de IBM o Microsoft. Otros como Cape Clear ofrecen implementaciones SOAP, pero necesitamos tener instalado todo el Software y conocer la infraestructura. Con REST, para visualizar esa información en una página Web, hace falta conocer XML y XSLT. Muchas plataformas (como Websphere y los servidores de aplicaciones Java) tienen esto integrado o es fácil añadirlo.

Por otro lado, si queremos usar SOAP, tenemos que usar todo el marco de trabajo .NET o usar Websphere Studio. Tendremos que usar una aplicación compleja, que se traduce en una gran inversión de tiempo y dinero.

¿Cómo puede saber una compañía si debería usar REST en vez de SOAP?

Si una compañía está trabajando con datos orientados a contenido, y no ha implementado SOAP, en ese caso REST es una buena opción para empezar a trabajar con XML y RSS. Si ya ha implementado SOAP y la infraestructura está montada, entonces no hay necesidad de usar REST.

## *6.4 AMIT ASARAVALA*

Amit Asaravala es un periodista independiente de San Francisco. Ha sido editor-jefe de una revista centrada en las técnicas Web y fundador de la revista *New Architect*.

### 6.4.1 Dando un descanso (REST) a SOAP [19]

En este artículo realiza un resumen de REST a la vez que lo compara con SOAP. Habla de la flexibilidad de la interfaz y de la seguridad. También comenta cuando deja de ser válido el modelo REST, y SOAP comienza a ser una buena opción.

REST es más una vieja filosofía que una nueva tecnología, mientras que SOAP parece que da un salto a la siguiente fase del desarrollo de Internet con nuevas especificaciones. La filosofía de REST expone que los principios y protocolos existentes en la Web son suficientes para crear servicios Web robustos. Esto significa que los desarrolladores que comprendan HTTP y XML pueden empezar a construir servicios Web de una manera correcta, sin necesidad de otras herramientas

#### 6.4.1.1 FLEXIBILIDAD DE LA INTERFAZ

La clave de la metodología de REST es escribir servicios Web usando una interfaz ya conocida y ampliamente usada: URI. Por ejemplo, un servicio de consulta de precios en el que un usuario pregunta por un objeto puede ser tan simple como hacer un script accesible en un servidor Web a través de la siguiente URI: `http://www.listadeproductos.com/precio?symbol=QQQ`.

Cualquier cliente o aplicación del servidor con soporte HTTP podría llamar a este servicio fácilmente con un comando HTTP GET. Dependiendo de cómo escribió el script el proveedor del servicio, la respuesta HTTP podría ser tan simple como algunas cabeceras estándar y una cadena de texto que contenga el precio actual. O podría ser un documento XML.

El método de la Interfaz tiene beneficios significantes respecto a los servicios basados en SOAP. Cualquier desarrollador puede imaginarse como crear y modificar una URI para acceder a diferentes recursos Web. SOAP, por otra parte, precisa del conocimiento de una nueva especificación XML, y la mayoría de los desarrolladores necesitarán un conjunto de herramientas SOAP para formar las peticiones y "parsear" los resultados.

#### 6.4.1.2 MÁS ÁMPLIO Y ESCLARECEDOR

Otro beneficio de la interfaz REST es que las peticiones y las respuestas pueden ser cortas. SOAP requiere una envoltura XML para cada petición y respuesta. Una vez que se ha declarado el espacio de nombres, la respuesta a una petición en SOAP puede ocupar hasta diez veces más bytes que la misma respuesta en REST.

Los adeptos a SOAP argumentan que toda esta cantidad de bytes es una característica necesaria de las aplicaciones distribuidas. En la práctica, la aplicación de petición y el servicio conocen los tipos de datos antes de tiempo, por tanto, transferir esa información en la petición y en la respuesta es gratuito.

¿Cómo podemos conocer los tipos de datos y su localización dentro de la respuesta antes de tiempo? Al igual que SOAP, REST necesita un documento que resuma los parámetros de entrada y de salida. La parte buena es que REST es suficientemente flexible como para que los desarrolladores puedan escribir ficheros WSDL para sus servicios si fuese necesaria una declaración formal. En caso contrario, la declaración sería tan simple como una página Web (que puede ser leída por los humanos) que diga, "Dé a este servicio una entrada, en el formato q = producto, y le devolverá el precio actual de ese producto como cadena de caracteres"

#### 6.4.1.3 GARANTÍAS DE SEGURIDAD

Probablemente el aspecto más interesente del debate REST contra SOAP es la seguridad. Aunque SOAP insiste en que enviar llamadas a procedimiento remoto a través del puerto estándar HTTP es una buena manera de asegurar que los servicios Web se pueden soportar a través de los límites organizacionales, los seguidores de REST argumentan que esta práctica es un gran defecto que compromete la seguridad de la red. Las llamadas a REST también van sobre HTTP o HTTPS, pero con REST el administrador (o firewall) puede distinguir la intención de cada mensaje analizando el comando HTTP que se usó en la petición. Por ejemplo, una petición GET siempre puede considerarse segura porque, por definición, no puede modificar ningún dato. Sólo puede solicitarlos.

Por otra parte, una petición SOAP típica usará POST para comunicarse con un servicio dado. Sin mirar dentro del sobre SOAP, una tarea puede ser a la vez un recurso o un consumidor. No existe ninguna manera de saber si la petición sólo quiere solicitar un dato o borrar todas las tablas de una base de datos.

Al igual que para la autenticación y la autorización, SOAP sitúa toda la responsabilidad en las manos del desarrollador de la aplicación. La metodología de REST por el contrario, se da cuenta de que los servidores Web ya tienen soporte para esas tareas. A través del uso de certificados industriales estándar y un sistema común de mantenimiento de identidad (como un servidor LDAP), los desarrolladores pueden hacer que la capa de red realice todo el trabajo pesado.

Esto no sólo es útil para los desarrolladores, hace más fácil la tarea de los administradores, quienes pueden usar algo tan simple como un fichero ACL para mantener sus servicios Web de igual manera que lo harían con otra URI.

#### 6.4.1.4 NO ES VÁLIDO PARA TODO

Para ser justos, REST no es la mejor solución para todos los servicios Web. Los datos que deben mantenerse seguros, no se pueden enviar como parámetros en las URI. Además, grandes cantidades de datos, como los detalles de las órdenes de compra, pueden hacerse rápidamente muy voluminosos e incluso desbordarse dentro de una URI. En esos casos SOAP es una solución bastante consistente. Lo más importante es

tomar REST primero y usar REST sólo si es necesario. Esto ayuda a que el desarrollo de aplicaciones sea simple y accesible.

Afortunadamente, la filosofía REST está cuajando entre los desarrolladores de servicios Web. La última versión de la especificación SOAP permitía que ciertos tipos de servicios fuesen expuestos a través de URI (aunque la respuesta siga siendo un mensaje SOAP). De manera similar, los usuarios de la plataforma Microsoft .NET pueden publicar servicios de manera que usan peticiones GET. Todo esto significa un cambio en la forma de pensar a la hora de mejorar el interfaz de los servicios Web.

Los desarrolladores necesitan comprender que enviar y recibir mensajes SOAP no es siempre la mejor manera de comunicar aplicaciones. A veces una simple interfaz REST y texto plano funciona (y salvamos tiempo y dinero en el proceso).

## *6.5 SAM RUBY*

Sam Ruby es un desarrollador de Software que ha hecho grandes aportaciones a los proyectos de *Apache Software Foundation* y a la estandarización de la Web.

### 6.5.1 REST + SOAP [20]

Este artículo es un breve resumen de las diferencias entre recursos (REST) y servicios (SOAP). Al final del artículo, expone las carencias de cada uno respecto al otro REST – SOAP = XLink y SOAP – REST = Procedimientos almacenados.

Desde la perspectiva de un protocolo la diferencia (expuesta en los términos más simples), es decidir que parte va dentro de un sobre y cual fuera. Cuando enviamos un cheque a una compañía de tarjetas de crédito, ¿Debemos poner el número de cuenta dentro (SOAP) o fuera (REST) del sobre? Esta diferencia puede parecer un poco exotérica, pero la revolución de la orientación a objetos puede ser expresada en términos similares.

La diferencia más grande entre los dos modelos se encuentra en el nombre del recurso. En REST, el recurso está identificado por medio de Uniform Resource Identifiers (URI). Mientras que SOAP descarta esta posibilidad, la mayoría de los servicios SOAP no se diseñan de esta manera.

REST y SOAP contienen características que la otra carece:

REST – SOAP = XLink

La característica más importante de la que SOAP carece actualmente es la posibilidad de enlazar recursos. SOAP 1.2 hace progresos importantes en esta dirección. Es de esperar que WSDL 1.2 complete este trabajo.

SOAP – REST = Procedimientos Almacenados.

Mirando como otros sistemas a gran escala realizan actualizaciones, nos damos cuenta de algunas cosas importantes en las áreas de producción para futuras investigaciones de REST.

Finalmente mencionamos que simplemente porque un servicio use HTTP GET, no significa que sea REST. Si estamos codificando parámetros en la URL, probablemente estemos haciendo uso de una petición RPC a un servicio, y no recuperando la representación de un recurso.

## *6.6 TIM BRAY*

Tim Bray es un desarrollador de Software, fue la persona que más contribuyó a formar los estándares *XML* y *Atom*. También es un empresario (cofundó *Open Text Corporation* y *Antarctica Systems*). Actualmente Tim es el director de tecnologías Web de Sun Microsystems.

### 6.6.1 The SOAP/XML-RPC/REST SAGA [21]

Artículo en el que comienza dando unas nociones de los métodos GET y POST para después pasar a explicar con un ejemplo las discrepancias entre REST y SOAP a la hora de su puesta a punto. Tim Bray asocia GET con REST y POST con SOAP.

#### 6.6.1.1 MANERAS DE HABLAR A UN SERVIDOR WEB

Un fragmento de Software que quiere hablarle a un servidor Web (un navegador por ejemplo, aunque hay muchos otros), generalmente usa el protocolo HTTP, que soporta todas las clases de transacciones, las más usadas son GET y POST.

GET es simple: enviamos una petición codificada en una URI al servidor, y obtenemos de vuelta del servidor un paquete de información. Esto es lo que ocurre cuando hacemos clic en un hiperlink; nada puede ser más fácil, la gran mayoría de las transacciones Web son GET. Hay más reglas que acompañan a GET, no está bien usarlo para hacer una compra o cualquier cosa que haga cambiar el estado. Es sólo para solicitar información.

Cuando usamos POST, contactamos con el servidor y enviamos un paquete de información, no una URI, y recibimos otro paquete de información. Deberíamos usar POST si queremos hacer algo más que solicitar información, por ejemplo comprar algo. La mayoría de las veces cuando rellenamos un formulario en un sitio Web y presionamos enviar, se está usando un POST.

Todo esto no tiene muchas complicaciones si consideramos que la Web son navegadores y páginas. Pero mucha gente quiere usar la Web para más; para sindicar blogs, ejecutar motores de búsqueda, incluso para programas B2B a gran escala.

¿Cómo hacemos que esto encaje en el mundo GET/POST? Hacemos notar que normalmente enviamos desde el servidor XML orientado a una máquina antes que HTML orientado a humanos. Si hacemos una petición con un POST, el paquete de información que enviamos normalmente es XML también.

#### 6.6.1.2 POST Y GET ¿IMPORTA ESO?

Realmente si. Lo primero de todo, si vamos a hacer algo que necesita algo más que solicitar información, debemos usar POST. Si por el contrario solo vamos a solicitar información, debemos usar GET, incluso si pudiésemos usar POST.

La razón de usar este método es que el parámetro de entrada de GET es una URI, así de simple, una maravillosa cadena de caracteres puede ser usada para direccional cualquier cosa en la Web, lo que más o menos es todo en estos días. Esto significa que podemos mandar un e-mail a esa URI, o llevarla a un motor de búsqueda, o almacenarla en el bookmark, o hacer una serie de cosas que probablemente todavía no se hayan inventado.

Esto también significa que los elementos como caches y proxies pueden realizar su trabajo más rápidamente y por consiguiente hacer la Web más rápida.

#### 6.6.1.3 REST

A la gente (como el W3C TAG) que trata de promover el uso de GET, se les dice que son discípulos de REST. REST es un acrónimo inventado por Roy Fielding que forma parte de la fundación Apache y fue uno de los autores de la especificación URI. Lo inventó para su tesis en la que trataba de formalizar la manera de trabajar de la Web.

Por otra parte, hay bastante gente acostumbrada a pensar en forma de complicadas estructuras de datos y complejas APIs. Esta gente tiende a pensar que GET es demasiado simple, inflexible y poco sofisticado. Están en lo cierto puesto que en los casos en los que solicitamos información es cuando entra en juego la ventaja de nombrar con URI, lo que hace que las transacciones con GET sean tan sumamente simples.

La gente que no está de acuerdo tiende a usar POST; prefieren codificar un paquete que contiene argumentos con una complicada API, en algunos casos como complicadas estructuras de datos, enviándolas y recibiendo de nuevo más estructuras de vuelta. Esta visión está fuera de la semántica Web en términos de ser una realidad práctica y útil.
El conflicto entre REST y SOAP/XML-RPC fue sacado a la luz por Paul Prescod, cuando escribió acerca del camino que estaba tomando Google. Fue una primera vista de los temas a tratar.

#### 6.6.1.4 SOAP y XML-RPC

Hay dos niveles de protocolos de bajo nivel que podemos usar para enviar (POST), SOAP y XML-RPC. SOAP, que creció del trabajo de Dave Winer y Microsoft, es bastante complejo y sobre-equipado para lo que debería ser una simple tarea, aunque también existen buenas implementaciones. XML-RPC desarrollado casi entero por Dave, es por el contrario demasiado simple y poco equipado, incluso para lo que debería ser un trabajo simple, aunque también existen buenas implementaciones.

#### 6.6.1.5 EL CASO DE TECHNORATI

Technorati es un sitio Web que maneja cientos de miles de blogs. La mayoría de los autores lo visitan regularmente para saber cuantos enlaces les apuntan. Dave Sifry de Technorati, inteligentemente, llegó a la conclusión de que los resultados deberían estar disponibles en XML, de manera que cualquiera que quisiera, pudiese conseguir la información fácilmente. La manera que propuso fue puramente al estilo REST, indicó como construir una URI para encontrar quién estaba apuntando y saber a qué apuntaba. Él simplemente documentó el XML que devolvería cuando se realiza un GET.

Dave Winer señaló que sería más fácil para los programadores con herramientas XML-RPC o SOAP conseguir esto vía POST si la interfaz fuese SOAP o XML-RPC. También señaló que es realmente poco útil que la URI tuviese que incluir un "appkey" (es una marca que da acceso a usar el servicio. Hay un límite de veces que se puede usar al día, excepto si pagamos dinero).

Realmente la postura de Dave Winer no es buena del todo (para no crear confusión entre ambos Daves, les llamaremos DW y DS). Como primer punto diremos que, esto es una aplicación al más puro estilo dame-algo-de-información, y construir las URI genera un poco más de trabajo, pero conseguimos que los resultados de Technorati sean ciudadanos de primera clase dentro de la Web y estén representados por cadenas cortas.
Obviamente si queremos construir una aplicación para enviar dinero a DS por usar su servicio, querremos enviar vía POST XML-RPC o algo equivalente.

El segundo punto de DW tampoco es totalmente correcto, aunque es cierto que si publicamos una URI con el appkey en ella, entonces cualquiera que consiga entrar y usarlo, estará haciendo uso de nuestra cuenta. No podemos saber si las cuentas basadas en appkey encajan bien en la arquitectura Web, es algo que deberá investigarse.
Hay otros métodos para realizar cuentas en los sitios Web; desafortunadamente para DS, la mayoría de ellas necesitan más trabajo para configurarlas que la aproximación por appkey.

¿Compensa usar la potencia de nombrar todo con URIs teniendo en cuenta los inconvenientes de tener que introducir complejas peticiones en ellas? ¿Compensa la opción de empaquetar las peticiones en XML con el inconveniente de perder la potencia de nombrado?

## *6.7 BENJAMIN CARLYLE*

Benjamin Carlyle es un desarrollador de Software. Este autor se ha mostrado muy activo a lo largo del debate REST-SOAP.

### 6.7.1 REST vs Orientación a Objetos [22]

Este artículo no compara REST directamente con SOAP sino con la Orientación a Objetos (OO). Explica los comienzos y orígenes de REST y OO. En cuanto al tema concreto de REST, este autor tiene una visión un poco particular de REST en términos de propiedades.

Las URIs son muy parecidas a los OIDs (punteros, referencias, o similares). Son cosas que se pueden dereferenciar para conseguir el recurso que buscamos, por tanto, no son comparables con las ideas de espacio de nombres global o variable local. Esto es una importante distinción de la que todavía no sabemos sus repercusiones.

Estamos en un estado en el que podemos comparar REST y la Orientación de Objetos desde un punto de vista práctico. Ahora, cuando la mayoría de la gente habla sobre REST, se están refiriendo a su uso como la mejor guía de diseño práctica para la Web. Cuando nosotros hablamos sobre ello, lo vemos desde el punto de vista de un desarrollador de software modular y distribuido. Hablamos de servicios Web y no del uso de HTML y los navegadores Web. Hablamos de diseño de software, no de diseño de sitios Web.

#### 6.7.1.1 SIMILITUDES

Desde nuestra perspectiva, los conceptos de Orientación a Objetos (OO) y REST son comparables. Ambos buscan identificar “cosas” que corresponden a algo que conocemos y con lo que interactuamos. En OO lo llamamos objeto. En REST lo llamamos recurso. OO y REST permiten algunas formas de abstracción. Podemos reemplazar un objeto por otro de la misma clase o del mismo tipo sin cambiar la manera en que el código del cliente interactúa con el objeto. El tipo es una abstracción común que representa cualquier objeto que encaje igual de bien. Podemos reemplazar un recurso por otro sin cambiar la manera en que los clientes interactúan con él. Los recursos son conceptos ligeramente menos pesados que los objetos, cuando hablamos de recursos normalmente necesitamos mencionar el espacio URI (el espacio de nombres de los recursos). Los clientes pueden interactuar con un nuevo software de servidor a través de sus recursos de la misma manera que interactuaban con el viejo software del servidor. Los recursos que se presentan representan a ambos igual de bien.

#### 6.7.1.2 DIFERENCIAS

La principal diferencia es el enfoque. Los objetos están enfocados en el tipo como un conjunto de operaciones que podemos realizar a un objeto en particular. Por otra parte REST dice que el conjunto de operaciones que podemos realizar a los recursos debe ser casi completamente uniforme. En vez de definir nuevas operaciones, REST enfatiza la creación y nombrado de nuevos recursos. Tanto como la limitación de verbos, REST busca reducir el número de tipos de contenido que entran en juego. Podemos dibujar REST como un triángulo con tres vértices que representan "nombres", "verbos" y "tipos de contenido". REST busca inclinar la balanza desde los verbos y tipos de contenido hacia el vértice de nombres. La orientación a objetos esta equilibrada en algún lugar entre los verbos (funciones) y los tipos de contenido (tipos y lista de parámetros). Cuando comprendamos los extremos de este triángulo, aprenderemos más sobre como inclinar la balanza para un problema en particular.

En OO los nombres de los objetos son siempre relativos a objeto actual o al espacio de nombres global. Normalmente vemos todo el acceso restringido a *esto->algo*, *parámetro->algo* o *algunacosa->algo*, donde *algo* es normalmente un verbo. Es difícil navegar más profundo en este nivel porque la manera en que OO mantiene la abstracción está demasiado oculta al conocimiento de otros objetos. En vez de eso, el diseño de OO normalmente proporciona una llamada que puede referenciar el estado o llamadas a funciones de sus propios objetos.

REST dice que el espacio de nombres debería ser lo más importante. Cada objeto que pueda contactar con otro objeto debe tener un nombre. No cualquier nombre, un nombre globalmente accesible. Si llevamos esto al extremo, cada objeto que pueda ser accesible desde otro objeto debe ser también accesible desde cualquier lugar del mundo por un identificador único y global:

En principio, cada objeto que alguien pueda querer o necesitar para citarlo debe tener una dirección que no conduzca a ambigüedades.

Douglas Engelbart en 1991 creó un documento sobre el nombrado, Berners-Lee escribió: "Este es probablemente el aspecto más crucial del diseño y estandarización en un sistema abierto de hipertexto. Concierne a la sintaxis del nombre que un documento o una parte de éste sea referenciada desde cualquier parte del mundo."

REST proporciona cada abstracción por medio de su espacio de nombres jerárquico en vez de intentar esconder el espacio de nombres. Desde el momento en que todos los objetos accesibles participan en esta interfaz simple, la línea entre los objetos está borrosa. OO está fija al concepto de un objeto detrás de una abstracción, pero REST permite desacoplar el conocimiento incluso sobre el objeto que está proporcionando el servicio que pedimos.

#### 6.7.1.3 HISTORIA

La historia que vamos a describir puede decirse que está fundamentada en rumores más que explicar exactamente como emergió cronológicamente cada aspecto. Podemos seguir la ruta de la historia de la OO desde la práctica de la programación estructurada. En la programa programación estructurada usamos muchos bucles "for" y "while". Atajamos un problema pensando en los pasos que debemos seguir para ejecutar la solución. En aquellos tiempos era difícil mantener estructuras de datos, porque a menudo significaba tener que mantener grandes cantidades de código base. Una implementación con listas enlazadas necesitaba insertar el código de la operación muchas veces, de manera que cuando cambiábamos de lista simplemente enlazadas a doblemente enlazadas era difícil asegurarnos de que todo el código seguía funcionando correctamente. Esto necesitó de la abstracción para conducirnos a la noción de Tipos de datos abstractos (ADTs).

ADT fue un gran hito. Definiendo un conjunto de operaciones legales en una estructura de datos y manteniendo todo el código en su lugar, podíamos reducir los costes de mantenimiento y manejar la complejidad. ADT se convirtió en una abstracción que podía representar diferentes implementaciones igual de bien. Las ventajas fueron tan importantes, que los detalles de las implementaciones subyacentes como las variables miembro se escondieron al código del cliente. Evitar fallos en la disciplina de la programación fue un gran enfoque.

La orientación a objetos se presenta cuado decimos "Esto funciona tan bien, ¿Por qué no lo aplicamos a estos conceptos?" En vez de aplicar la técnica solo a las estructuras de datos, encontramos que podemos aplicarlo siempre que necesitemos abstracción. Podríamos aplicarlo a algoritmos o a cualquier cosa conceptualmente abstracta. Desarrollamos patrones de diseño con intención de explicar a los demás como usar objetos para resolver problemas.

#### 6.7.1.4 LA HISTORIA DE REST

REST tiene una historia obvia en la red, donde la abstracción es un concepto fundamental. Los recursos de la Web operan a través de protocolos y otros mecanismos que nos fuerzan a esconder la implementación de un objeto de la de otro. Las semillas de REST se encuentran en el software "puro" también.
Veamos los Java Beans. Las propiedades de los Bean son las siguientes:

1. Cada clase Java Bean debe implementar el interfaz java.io.Serializable
2. No debe haber constructores sin parámetros
3. Las propiedades deben ser accedidas usando los métodos get y set.
4. Debe contener los métodos manejadores de eventos requeridos

Esto es un paso significativo desde un modelo orientado a objetos hasta lo que es REST. Dejaremos de lado la implementación de serializable que permite almacenar, transmitir y desempaquetar una representación en el otro extremo. También dejaremos de lado el constructor por defecto. Lo que consideraremos realmente importante es el uso de las propiedades, o como podríamos llamarlas: Recursos.

Es mucho más fácil tratar con un conjunto (“real” o de otro tipo) de propiedades que tratar con llamadas a funciones. Los seres humanos encontrarán más fácil tratar con esos objetos. Incrementando la presencia de objetos en el espacio de nombres y definiendo un conjunto de operaciones que pueden ser usadas con cada elemento presente en ese espacio de nombre, conseguimos un aumento de la facilidad en el trato con cada elemento. No perdemos ninguna abstracción de las que ganamos al cambiar a ADT, en primer lugar porque ahora esas propiedades no están revelando la implementación interna de nuestro objeto. Están formando parte de la interfaz. Cuando configuramos esas propiedades o las conseguimos, el código todavía se esconde a la hora de hacer lo que los objetos eligen. El conjunto de propiedades puede representar diferentes tipos de objetos igual de bien.

Esto es un ejemplo del triángulo que forman los verbos y tipos de contenidos para ir hacia los nombres. Parece ser más importante conocer como se llama algo que el tipo de datos que es. Este resultado no es obvio desde los primeros principios.

#### 6.7.1.5 ¿CÓMO HACER QUE NUESTRO SOFTWARE SE ACERQUE MÁS A REST?

Trataremos de exponer nuestras funciones como propiedades o sub-objetos en vez de funciones, exponiendo un simple conjunto de funciones en esas propiedades. Una buena práctica de REST actualmente significa que usar GET, PUT y DELETE es la mayor parte de lo que necesitamos. Deberíamos usar POST como una operación “Crear y PUT”. Trataremos de dar nombres globalmente accesibles a los recursos. La teoría es que otros objetos sólo accederán a ellos si les damos un hiperlink, de manera que la privacidad no sea un problema. Usar tipos que sean lo más simple y uniforme posible.

Cualquier código que acceda a este nombre, verbo y tipo de contenido operará simplemente usando hiperlinks. Podemos elegir abrir el acceso a otros espacios URI, como hace HTTP y los esquemas de ficheros. De esta manera podemos hacer hiperlinks desde y hacia esos espacios sin alterar el significado de código del cliente.

## *6.8 Disenchanted dictionary*

Disenchanted dictionary es una página Web donde los autores cuelgan artículos de manera anónima. En este sitio se pueden encontrar artículos relacionados con el mundo de los Servicios Web.

### 6.8.1 REST [23]

Este artículo realiza una comparación entre REST y SOAP en cuanto a flexibilidad y complejidad.

Cualquier página Web a la que miremos, está representando algún dato. Esto significa un estado específico, y se transfiere al navegador cuando hacemos una petición de un recurso disponible a una determinada dirección (algo como `http://www.disenchanted.com/dis/lookup?node=1837`). REST es una manera de describir como cada petición de una página Web en Internet funciona, pero también es una manera de construir servicios Web.
REST compite en el nivel más alto con SOAP, un protocolo de llamada a procedimiento remoto que trata los servicios remotos como funciones que se invocan con argumentos. REST puede llegar a hacer lo mismo que SOAP, pero siguiendo un modelo diferente. En vez de tratar los servicios remotos como programas que corren y necesitan argumentos, REST solo conoce los recursos por su URI. La diferencia con la que cada modelo trata los recursos remotos implica algunas ventajas para REST.
Con SOAP, se espera que un programa corra en una máquina remota cada vez que se llama a una función, lo que se devuelve se crea momentos antes de usarse. REST no asume eso, sólo conoce que hay algún dato en una dirección. La dirección apunta al tipo de datos que quiere y al estado en el que debe estar. Es como la diferencia entre pedir a una secretaria que calcule una factura (SOAP) o pedirle simplemente que la devuelva (REST).

Independientemente de la manera en que formulamos la petición a la secretaria, obtendremos los mismos datos. La cuestión es determinar cuanto trabajo está implicado para la pobre secretaria. ¿Tendrá que coger la calculadora y hacer uso de las matemáticas o simplemente abrirá un fichero y cogerá un papel que fue escrito días o semanas antes?

Esta es la ventaja que tiene REST sobre SOAP. La URI que copiamos en el navegador Web (u otra aplicación) define exactamente lo que queremos y el estado en que lo queremos (como puede ser el tiempo en Nueva York el dia 1 de Abril de 1999). Si para darnos este dato el servidor tiene que ejecutar un programa que genere la información "al vuelo", o simplemente la recupera de un fichero creado años antes, es algo que no inmiscuye ni la red ni a la persona que se ocupa de configurar el servidor. Con SOAP esto no es posible, y probablemente nunca lo será, porque el paquete SOAP es más complejo. SOAP hace difícil saber si un dato ha sido calculado antes cuando ejecutamos un simple programa de recogida de datos.

REST es más simple, pero depende de HTTP como protocolo que envía peticiones en una dirección y datos en la otra. Esto podría ser una ventaja porque HTTP tiene un grupo de funciones diseñadas y testadas, como la autenticación y el mantenimiento de recursos, mientras que cualquier cosa construida en SOAP necesita reinventar todo eso.

También puede ser una desventaja. SOAP puede hacer un túnel sobre cualquier protocolo, no solo HTTP (podemos enviar paquetes SOAP por e-mail, por ejemplo, si el retraso no es un problema en nuestra aplicación). Con SOAP, la petición puede ser más sofisticada; por ejemplo, puede codificar tipos de datos.

## 6.9 Conclusiones

En este capítulo se ha dado una visión general del debate que ha existido en la Web entorno a la comparación entre REST y SOAP. Se han expuesto los artículos más relevantes que se han escrito hasta el momento. En el siguiente capítulo se comentarán algunos de los Servicios Web que están actualmente funcionando de acuerdo al modelo de arquitectura REST.