Extreme Markup Languages 2002

Montréal, Québec
August 6-9, 2002

Roots of the REST/SOAP Debate

Paul Prescod
ConstantRevolution Consulting

Abstract

In order to communicate over networks we need standardized data formats and protocols. But how do we move forward toward this goal? One popular debate centers around the best way to define new data formats. XML dominates this area and so the primary question left is how and whether to use schemas and if so, what schema language to use. This paper will address a different question: How will we standardize the protocols used to transport the XML documents? This paper will describe three different strategies and attempt to summarize their strengths and weaknesses from an admittedly partisan point of view.

Roots of the REST/SOAP Debate

Table of Contents

Roots of the REST/SOAP Debate

Paul Prescod

§ 1 Understanding the Problems

What problem are web services trying to solve? It sounds like it should be an easy question to answer but web services are seldom described in terms of their problem domain. The general gist of most definitions seems to be "Web Services allow programs written in different languages on different platforms to communicate with each other in a standards-based way." Any network programmer will tell you that this is also the definition of a language-independent *protocol*. Protocols are languages that computer programs use to communicate with each other over networks. So SOAP and WSDL are ways for describing and implementing new protocols.

But what are the business problems for all of these new protocols? There are a few main areas that seem to attract most of the attention for the web services technologies:

Enterprise Application Integration	The integration of legacy software systems within an organization in order to allow the systems to have a more complete and consistent world-view. This is essentially an internal manner.
Cross-Organization Integration or "B2B"	The creation of public interfaces to allow partners and customers to interact with internal systems in a programmatic fashion. For example you might sell a service to allow customers to programmatically retrieve a current price for a commodity. You might also allow them to buy or sell the commodity programmatically.

These two problem domains seem related but they vary radically in their details. The first is entirely within a single administrative domain. If a new protocol does not work perfectly, it can be ripped out and replaced. In the cross-business environment, ripping it out affects your customers. They may have no incentive to upgrade to your new protocol and will be annoyed if it changes constantly. Within a business, demand for a service can be fairly easily judged. On the external interface, demand can spike if a service turns out to be wildly popular with customers. Within a business, security can (to a certain extent) be maintained merely by firing people who abuse it. On the external interface you should extend a much lower level of trust.

It is unfortunately common for people to believe that the cross-organization problem is essentially similar to the enterprise integration one. In reality, they are quite different and may require different technical strategies altogether.

§ 2 Brave New Protocols

There are many different levels of protocols but the ones that most directly relate to business and social issues are the ones closest to the top, the so-called "application protocols." An application protocol could be roughly described as a protocol that describes the solution to some networking problem. For instance: "how do I move files from one computer to another." FTP (file transfer protocol) solves that problem. "How do I send electronic mail from one computer to another?" SMTP (Simple Mail Transfer Protocol) protocol solves that.

One of the defining characteristics of application protocols is that they set up address spaces for sources, targets and containers of information. HTTP has a notion of "resource" which supports the operations "GET", "PUT", "POST" and "DELETE". SMTP has a notion of "mailbox" into which you can put messages. FTP has a file-path paradigm. Application protocols also allow you to define the allowed flows of information between clients, servers and intermediaries (known collectively as "nodes").

Historically, new application protocols have only been deployed slowly and laboriously over years. Today, many feel that the Internet and Intranet environments have evolved to the stage where this should no longer be the case. The Internet is now pervasive. Every major organization in the developed world is connected to every other one. Any of them can send IP packets to any other one. XML has freed us from the constraints of HTML.

It seems that we should be able to integrate any system with any other system easily. The traditional means for doing this has always been through new application protocols. Therefore it is natural to want new protocols.

This paper will consider three strategies for doing these sorts of integrations:

1. Custom protocols
2. Protocol frameworks
3. Horizontal protocols

§ 3 The Custom Protocols Approach

The first strategy is the traditional one used for Internet protocols. A team studies the problem domain and defines a protocol based upon some existing protocol. Historically new protocols were based upon one of TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). More recently it has become popular to build on top of HTTP (HyperText Transfer Protocol) or SMTP (Simple Mail Transfer Protocol).

This strategy does not allow a deep level of infrastructure reuse between different protocols. For instance, RosettaNet's protocol is quite a bit different than the ebXML transport binding which is in turn quite different than Jabber.

They all reuse the underlying Internet protocols and in some cases reuse the MIME and MIME/multipart message formats. Today most new protocols are built on top of XML. XML provides a layer of reuse, but not a deep one. Overall, not much else is shared.

This strategy presumes that new protocols come along rarely and that they do not share enough to spend the time or effort building common infrastructure for them. This assumption seems to have held in the early days of the Internet where network development was more coordinated and gradual. This model was also popular in the early days of XML protocols before the concept of Web services crystallized. For instance early versions of ebXML, ICE and RosettaNet were built on top of HTTP, SMTP and MIME.

This strategy also implies a willingness to settle for a fixed list of bindings to known protocols and add other bindings if other protocols become relevant. It is possible for a specification to merely say that it is "transport independent" but if clear bindings are not articulated somewhere then interoperability will remain illusive.

Most custom protocols bind either directly to TCP or to protocols that run on top of TCP. For instance, RosettaNet uses HTTP and SMTP. This in turn implies that RosettaNet depends on IP and the Internet -- at least until such time as new bindings come along.

The strength of this approach is that it does not require waiting for any other standardization project. The weakness is that it is potentially more expensive for all of the usual reasons that custom development projects are more expensive than projects that build upon some other framework or abstraction. Custom-built protocols all work quite differently, which makes client and server development more expensive. Custom-built protocols may not work well together. They may repeat mistakes others have made or miss the opportunity to incorporate their design ideas.

Finally, there is just the effort of designing and maintaining the custom protocol and its bindings. This is expensive even for experts. There also exists a consensus (sometimes explicit, sometimes

implicit) that creating new protocols is beyond the abilities of ordinary business developers. This is considered an issue because there is a presumption that the standards created by professional standardizers will not be sufficient for the ad hoc "glue" required to integrate disparate, proprietary internal systems.

Examples of this approach include the original ebXML TRP (prior to its reinvention on top of SOAP), the 1998 version of ICE, RosettaNet and Jabber. Pre-XML examples include most of the popular Internet application protocols such as HTTP, SMTP and FTP.

§ 4 The Framework Approach

The next approach is based upon the idea that we will need to create new protocols constantly and therefore it makes sense to establish a framework for doing so. This is what SOAP does. Some of the features of the framework could be:

- common type system
- common service description language
- common addressing model
- common bindings to lower-level transports
- common security or routing frameworks
- common mechanisms for breaking the message into components like header and body. This is known as "framing".

The SOAP/WSDL pair form the basis for the most popular current framework. Another, less famous framework is the BEEP/BXXP pair.

When executed properly, this strategy has a variety of advantages over the custom protocols strategy. First, it obviously allows a pooling of intellect and talent. This should allow the framework developers to get things right. Second, it allows for the development of a common software infrastructure. We have seen this through the emergence of the "SOAP toolkit."

One advantage of this model over the former is that it could bring the development of new protocols into the realm of possibility for average business software developers. Visual Studio.NET provides tool support so easy that you can turn an existing class into a web service with the click of a button.

This not an unalloyed-advantage, however. The first problem it creates is a false sense of confidence. Even senior Microsoft employees have admitted that the service you get "for free" from Visual Studio.NET is probably not of sufficient quality that you would want to deploy it in an enterprise situation. The problem is that the environment helps you to generate services where the XML is more or less invisible to the programmer. Unfortunately the XML is the part of the system which is responsible for extensibility and managing change. Service frameworks that hide it are as brittle as those using pre-XML technologies like DCOM and CORBA.

The simple tools get you started and then abandon you when you run into issues of extensibility or interoperability between services. These problems are very reminiscent of the problems people had with software-generating wizards in the early nineties. They would get you to the point of demo-ware and then leave you stuck - usually just after you gave your boss the impression that the problem was almost solved.

The tools do not solve any of the standard distributed computing problems such as latency, handling of deadlocks and race conditions, object-level interoperability, distributed resource management etc. They do not yet even address more basic problems such as security, routing and asynchronous bi-directional communication.

Let's presume, however, that all of these problems will be solved one day. A more subtle issue is that too much success adds up to a kind of failure. If we make it so easy to define new protocols, people will go out and design many new protocols. Unfortunately, two systems that speak different protocols cannot communicate with each other unless there exists some form of custom-coded bridge. It is therefore extremely inefficient for every business to design its own protocols for payroll management and then glue those payroll systems to custom-designed protocols for human resources management.

If possible, it would be better to have the big Enterprise Resource Planning (ERP) and Human Resource (HR) software vendors standardize their protocols and build the links between them built on standards. Business developers should develop custom protocols only as a last resort (just as they should develop custom software only as a last resort!). In all of the excitement over Web Services it seems that this is often forgotten. For instance Microsoft says: "Just as ubiquitous support of HTML enabled the World Wide Web, ubiquitous support of SOAP enables XML Web services." But this analogy is poor: agreeing on SOAP buys very little actual interoperability between applications. You need to agree on *how you use* SOAP. That is where you get application-level interoperability.

In the late nineties it was common to believe that choosing to expose an XML interface to a service would intrinsically give one application compatibility with other applications also exposing an XML interface. This faith has moved up a level to SOAP but it is as wrong now as it was then. Unless two pieces of software standardize not just on the framework but also on the details of the application protocol (method names, calling conventions, etc.), they will not interoperate. Given a SOAP client and server, you can only move files between the two computers if you invent an application protocol that allows that, a "SOAP-FTP". You can only do supply-chain management if you invent an application protocol that allows that. And so forth.

These protocols have been very slow to materialize. One theory is that the very mindset promoted by the standards and tools is that everyone should invent their own protocols. Perhaps it will take time for people to move beyond this belief. An alternate theory is that they are still waiting on more framework infrastructure. Perhaps they need final standards such as WS-Security, WS-Routing, WS-Inspect and so forth.

This raises another issue with the framework model: timing. There is a period while the framework is being developed where vital work on higher layers tends to pause, waiting for the outcome of the "framework wars." XML deployment in many companies has been delayed while they "wait and see" what emerges from the W3C and WS-Interoperability organizations. This is unfortunate because it is the data dictionaries, vocabularies and ontologies that they develop that will be the fundamental basis of their web services, no matter which way the technology goes.

A weakness of this model relative to the custom-protocols model is that a mistake made at the framework layer will be inherited by all software created on top of the framework. This explains the ferocity of the arguments about what the framework should look like. Millions of dollars will be spent or wasted in the next several years building upon the "Web Services Architecture." If it has mistakes, they could be costly. There is reason to believe that there are mistakes in the SOAP/WSDL/UDDI trinity and this drives part of the SOAP/REST debate. This paper will examine these flaws in more detail later on.

§ 5 The Horizontal Protocol Approach

The third strategy is the most radical. It says that instead of developing domain-specific protocols (molecular modelling protocol, travel reservation protocol), we could use general-purpose protocols to transfer domain-specific data. Rather than choosing to start from scratch or use a protocol framework, we can all agree to use one or a few general-purpose application protocols.

Such a protocol would need to be well-tested because it would be the basis for the entire Web Services information system. The system would be even more susceptible to flaws than in the framework model.

It would have to be very generic. Presumably it would have a generic concept of object and would support common concepts like the creation, retrieval, update and destruction of objects. These comprise the "CRUD" design pattern.[1] It would have to encapsulate the right ideas about security, routing, addressing, caching and so forth. The "Highlander"[2] protocol would of course have to be able to address and transfer a variety of different information types.

There is such a protocol, the Hypertext Transfer Protocol (HTTP). HTTP is very general. Its main methods map to the "CRUD-pattern". They are GET/PUT/POST/DELETE methods. HTTP has well-understood wire encryption through HTTPS. Most important, HTTP embeds the most important of the Web's innovations, the URI namespace. Before we go into further discussion of HTTP, we should compare this strategy against the others in terms of strengths and weaknesses.

Bear in mind, however, that it does not matter for this analysis whether the one true protocol is HTTP or something HTTP-like with new features. What matters is the idea of having a protocol that is designed to be general purpose rather than specific to one narrow problem domain. The protocol would have a notion of addressable object and generic methods for managing these objects.

There could actually be a few of these protocols with different capabilities in terms of connections, directionality, reliability, security and so forth. As long as they logically interoperate (have a single concept of "thing" and "address"), you can think of them as a single meta-protocol. For instance HTTP and HTTPS are obviously related and you could imagine the addition of HTTPU for an extremely efficient but unreliable unidirectional version and there is already a (very rough!) proposal for HTTPR, a reliable version. You would select the appropriate variant of the Highlander protocol for the variant's extra features (security, reliability, performance etc.), not its differing application semantics.

One obvious strength of the one-true-protocol strategy is interoperability. If every service uses the same protocol then protocol ceases to be a potential point for interoperability failure. That does not mean that application-level interoperability becomes free. It means that one particular source of interoperability problems is removed.

Instead the interoperability problem hits after the protocol transfers some information and the client or server must attempt to decipher the file format or XML vocabulary. This problem of document format standardization is not specific to this "Highlander" protocol standardization model. The previously described models have the same problem. In particular, the framework model requires you first to agree on the framework and then on the concrete protocol (methods, addressing scheme, etc.) and finally on the data representation. In the Highlander model you know what protocol to use because there is only one. So all that is left to decide is the data representation.

Interoperability improves not only between clients and servers but between resources. One of the core concepts of the HTTP model is that there is no hard and fast boundary between one application and another. Instead, monolithic applications are decomposed into resources with links between them. For instance "acronym.com" has links to Amazon searches for that acronym. This globally unified web of resources is an application *in its own right*. Mark Baker says: "The goal is not building a better application. The goal is to replace the application as the predominant means of delivering value to customers."

Just as the hypertext Web fuelled the last Internet boom, there could be other Webs: the financial services Web, the supply chain management Web, and so forth. Because these Webs will use identical protocols and addressing mechanisms and will use representations based upon common technologies (XML and RDF), there will be no hard and fast boundaries between them. From a sociological perspective there may be many Webs but from a technical perspective there will be just one. Any

resource can be given a hypertext rendition so the hypertext web becomes merely a view over the Web as a whole.

This model is also strong in infrastructure reusability. Just as in the framework model, we can build tools that work across all domains using the protocol. The tools can be simpler than in the framework case because they will need less configuration in order to work together. For instance, an HTTP cache can work out of the box without being specially configured to know what to cache and what not to cache.

Development tools are harder to create for this model because they are forced to revolve around the protocol's model of the world rather than allowing the tool to merely expose its own internal model (as Visual Studio.NET does). Web-style tools like Cold Fusion, WebSphere and ASP help you to bridge the gap between your internal model and the HTTP model.

The learning curve issue cuts both ways. On the one hand, developers need to really understand the horizontally applicable protocol. So there is a start-up phase where they must learn to think in its model. On the other hand, once they have done that they can offload many protocol design issues.

In general, the requirement to fit into a single protocol limits expressiveness and the restrictions it applies (often ostensibly for your own good) may chafe. Performance of services may also suffer from the need to map into an extremely general structure.

This model wins on timing. HTTP is available today and people were using it to implement "web services" before the term was even coined. So although there may be reasons to extend HTTP, there is no reason to wait for further standardization before using it. Of course concrete representations (XML vocabularies) must still be standardized. But these must be standardized no matter what strategy is taken with respect to protocols.

The greatest weakness of this model is that it requires developers to have faith that the Highlander protocol's model is really general and powerful enough to fit their problem domains. This has been a problem for HTTP and is the primary reason that this model has not thus far been taking seriously by most developers.

§ 6 Role of REST

HTTP's underlying design model is called the REpresentational State Transfer (REST) model. Fielding articulated the ideas of REST but he did not argue that either REST or HTTP constituted a sufficient basis for a web services architecture. As far as I know, he has never articulated his personal vision of a web services architecture. He has also never said whether he believes that there needs to be a web services architecture distinct from web architecture. Others have come to the conclusion that there does not need to be. Web architecture and REST are already a viable basis for web services architecture.

REST emphasizes:[Fielding]

- scalability of component interactions,
- generality of interfaces,
- independent deployment of components,
- and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

The first, "scalability of component interactions" has been unquestionably achieved. It has scaled up as the Web has grown exponentially. One facet of scalability is shown in the variety of client software that can access and be accessed through the Web. Even as the Web browser market thins, the plurality of Web-based tools and devices continue to grow, from package management systems

to handheld devices, content-management systems to industrial monitoring systems. HTTP has proven its scalability and generality.

The second goal, "generality of interfaces" is a key part of what makes REST advocates believe that is it a better basis for a Web Services framework than are the SOAP-based technologies. Any HTTP client can talk to any HTTP server with no further configuration. This is not true for SOAP. With SOAP, you must also know the method names, addressing model and procedural conventions of the particular service. This is because HTTP is an application protocol whereas SOAP is a protocol framework.

Fielding requires "independent deployment of components" because this is a reality that must be dealt with on the Internet. Client and server implementations may be deployed and last for years or even decades. Ancient HTTP servers may never be upgraded and newer clients must be able to talk to them. Ancient clients may need to talk to new servers. Designing a protocol that allows this is quite difficult and I expect that this will be one of the holes in the protocols generated by business analysts with simplistic tools. HTTP allows extensibility through headers, through new URI-generation techniques, through the ability to create new methods and through the ability to create new content-types.

The final core goal of REST is "compatibility with intermediaries". The most popular intermediaries are various kinds of Web proxies. Some proxies cache information to improve performance. Others enforce security policies. Another important kind of intermediary is a gateway, which encapsulates non-Web systems.

REST achieves these goals by applying four constraints:

- identification of resources;
- manipulation of resources through representations;
- self-descriptive messages;
- and, hypermedia as the engine of application state

Identification of resources is always done with URIs. HTTP is a URI-centric protocol. Resources are the logical objects that we send messages to. Resources cannot be directly accessed or modified. We work rather with *representations* of them. When we use the HTTP PUT method to send information, it is taken as a representation of what we would like the resource's state to be. Internally the resource's state may be anything from a relational database to a flat text file.

REST dictates that HTTP messages should be as *self-descriptive* as possible. This makes it possible for intermediaries to interpret messages and perform services on behalf of the user. One of the ways HTTP achieves this is to standardize several methods, many headers and the addressing mechanism. For instance caches know that by default GET is cachable (because it is side-effect-free) but POST is not. They also know to watch out for specific headers that can control cache longevity. HTTP is a stateless protocol and when it is used properly, it is possible to interpret each message without any knowledge of the messages that preceded and followed it. For instance rather than "logging in" the way FTP does, HTTP sends username/password information with every message.

Finally, REST describes a world in which *hypermedia is the engine of application state*. This means that the current state of a particular Web application should be captured in one or more hypertext documents, residing either on the client or the server. The server knows about the state of its resources but does not try to keep track of individual client "sessions". It is the client that knows what mission it needs to complete for the end-user. It is the client's responsibility to navigate from resource to resource, collecting the information it needs or triggering the state changes that it needs to trigger.

There are millions of web applications that implicitly inherit these constraints from HTTP. There is a discipline behind scalable web site design that can be learned from the various standards and web

architectural documents. It is also true, however, that many web sites compromise one or more of these principles, either out of simple short-sighted thinking or in order to do things that are discouraged by the architecture, such as tracking users moving through a site. This tracking is possible within the web infrastructure but hurts scalability by turning an essentially connectionless medium into a connection-based one.

Although good REST and HTTP use relies on many rules of thumb and principles, many of them can collapse down to one simple one: "If I were designing a high-quality HTML site, how would I do that?" Instead of HTML representations you can switch to XML representations and get quite a good start on your design.

REST advocates have come to believe that these ideas are just as applicable to application integration problems as to hypertext integration problems. Fielding is quite clear that REST is not a cure-all. Certain of its design features will not be appropriate for other networked applications. Nevertheless, those who have chosen to adopt REST as a web services model feel that it at least articulates a principled design philosophy with documented strengths, weaknesses and areas of applicability.

In contrast, the conventional web services architecture has drifted directionless. Its various maintainers and developers have incorporated upon some important ideas and have ignored others. Arguably they have failed to embrace the most important of the Web's successful ideas.

§ 7 Perceived Limitations of SOAP

In the early days, SOAP was designed to be a remote procedure call protocol that would go through firewalls. Don Box says: "I got a call from Dave [Reed] in late winter of 1998 asking me if I wanted to come up to Microsoft for a while and work on replacing DCOM with XML. I said [that] when I figured out what XML is, I might be able to help....If you look at the state of the average organization, they use proxy servers and they use firewalls to prevent normal TCP traffic from making it from one machine to another. Instead, they set up this infrastructure to allow HTTP to work. So part of the problem was replacing the transport..."[DonBoxTechnetcast]

An early focus of this effort was to build a type system. It was thought that XML's major weakness was that it lacked a type system. This view has changed since the arrival of the XML Schema specification. The type system has been downgraded to an "annex" of the specification and Don Box (the editor of the early specifications) has declared that if the XML Schema type system had been available when he invented SOAP, he would never have bothered to do SOAP at all.[DonBoxInInfoworld]

He does not argue, however, that SOAP is now useless. After the first version, there was a period where the major focus of SOAP's growth was in extensibility. SOAP headers were given more prominence and became more sophisticated. SOAP gained a mechanism for passing arbitrarily extensible XML rather than fixed data types and data structures. SOAP grew a mechanism for running on top of multiple transport protocol bindings and a notion of "binding features." SOAP grew from being a protocol into a "protocol framework." In other words, SOAP grew and changed so drastically that now SOAP advocates can deprecate a feature (the SOAP type system) that was at one point considered the primary reason for SOAP's existence. Its goals have changed so drastically that the word "SOAP" went from being an acronym for "Simple Object Access Protocol" to being merely a meaningless name.

The most recent changes have been to respond to some of the criticisms from REST advocates. We have pointed out ways in which SOAP abuses HTTP and fails to integrate properly into Web architecture. These changes are not yet deployed in any toolkits and may never be. But the changes are nevertheless a sign of hope!

Today, SOAP is many things, from a syntax to a framework, from an RPC protocol to an abstract, one-way messaging environment. It is difficult to make unequivocal statements about what SOAP

is or what it allows. Given this, we cannot even say with confidence whether SOAP and REST are in competition. At this point, all that is clear is that standard SOAP usage is vastly out of sync with web architecture. The SOAP 1.2 specification is sufficiently flexible that we do not yet know what SOAP will look like when it is actually deployed. Therefore my criticisms are based in part on how today's SOAP toolkits encourage developers to structure services.

7.1 *Web Services and Web Architecture*

One major difference between the REST point of view and that of the standard web services architecture is that REST advocates view the Web as an information system in its own right and we advocate that other systems should be integrated into the information system through gateways.

Consider Tim Berners-Lee's vision of the Web:

> The Web is simply defined as the universe of global network-accessible information.[BernersLee]

Berners-Lee's point of view is that the first goal of the Web is to establish a shared information space. Legacy apps can participate by publishing objects and services into the space.

This is very different than the terminology that tends to surround Web services. In the Web Services world it is common to talk of the Web as being a "transport" for messages which are essentially only interpreted by systems outside of the Web. In that view, the Web is transport middleware for the systems. This approach is destructive in the long run because the two applications communicating in this way have not established any common ground with all of the other applications on the network. Adding a third application will not be much easier than integrating the first two.

The core of the Web's shared information space idea is the URI. The Web services standards have never adopted the notion of a Web as a shared information space and thus have not yet fully adopted the Web's model of URI usage. They have rather always presumed that every application would set up its own unique namespace from scratch. Stock quote servers would use a stock name-based namespace. Microsoft's Hailstorm used "Passport User IDs" (PUIDs), InstanceIds and XPaths. UDDI uses Universally Unique Identifiers (UUIDs). I do not know of one deployed web service that uses URIs as an addressing mechanism and I know of very few that even share an addressing model between them.

The Web Services standards were never designed to use URIs as resource identifiers and in fact WSDL makes this essentially impossible. There is no way to say in WSDL that a particular element in a SOAP message represents a URI and that the URI points to a web service. Therefore each WSDL describes one and only one Web resource and provides no way to describe links to other resources. SOAP and WSDL use URIs only to address a heavyweight "endpoint" which in turn manages all of the objects within it. For example, to send a message to a particular account you would do something like this:

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

To get the balance associated with an account, you would do something like this:

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

Note that the account itself is not directly addressable. It has no URI, only an account number. You can only get at the account through the service, not directly. Not also that the bank account numbers consist of a separate namespace, distinct from the Web's URI namespace. In the Web-centric version of the service, the accounts would be individually addressable and each would have a URI.

Let me offer an analogy. Suppose you were living temporarily in a hotel. The hotel might not have direct dial connections from the outside. In order to call a room you have to contact the operator first (this is like contacting the "SOAP endpoint") and then ask them to connect you to your room. Now imagine that there is an outside service that you would like to buy. It is a once-a-day automated wake-up call and horoscope service. You try to sign up for the service but when you are asked to enter the phone number to call back you realize that there is no single number you can provide. The service must contact the operator first and then the operator must patch them through to you. Obviously the computer on the other end is not going to be smart enough to know to go through the operator and the operator will not know to patch the call through to your room.

If everybody lived in a hotel like that, the horoscope service would be practically impossible. A particular application of the phone system would simply cease to exist. [3]

Note that the problem is not obvious in the design of either system. It is when you try to unite the two systems that you wish that the hotel had used the international standard phone addressing "syntax" rather than having an extra level of misdirection through the operator. SOAP "endpoints" are like the operator. The objects they work with (purchase orders, bank accounts, personnel records) are the hotel rooms. The data is at the mercy of the SOAP endpoint. If the interfaces of the client and server do not exactly align, the two cannot communicate. A third party application trying to integrate them might have to explicitly extract information from one service and push it into the other. This is still not a good solution because the third party would thereafter be responsible for moving the data from one to the other. Ideally you would prefer to just introduce the two data elements allow them to keep each other informed of their state changes.

In contrast, HTTP strongly promote the use of URIs as identifiers. In fact, Web architecture specifically encourages the use of URIs that can be resolved into documents. Typically this means "http:" URIs. Needless to say, URIs are the central unifying concept of HTTP. You may invent your own headers and in some cases your own methods. You may use any content-type as the body (or none at all) but you must *always* send a URI as the target resource for every HTTP request.

This choice has real consequences. If two HTTP services can agree on an information representation (for instance RSS (the Rich Site Summary) or SAML (Security Assertions Markup Language) then those two services can communicate about these information resources merely by passing URIs to resources. For instance if I have an RSS document representing the information on a weblog, I can feed that to a weblog watching system like Meerkat merely by handing it a URI. Meerkat does not need to be told what addressing mechanism I am using because it is standardized. Meerkat also does not need to be told what method to use to fetch the information because the HTTP "GET" method is implied. You could also imagine the opposite service. You could ask Meerkat to push RSS feeds to you by doing a PUT to some URI you specify.

Addressing is not discussed anywhere in the "Global XML Web Services Architecture", Microsoft's vision for the future of Web Services. In fact I cannot find the issue discussed in any web services architecture from any large software vendors.

Bear in mind that SOAP's weakness around addressing does not stem merely from ignoring the lessons of the Web. In fact, older standards like CORBA and DCOM did much better in this area. Every object had an address and although the address syntaxes were not URIs, they were at least standardized within the domain of each RPC protocol. SOAP lacks any equivalent addressing model. Although it is the new, new thing, it is actually less sophisticated in this way than its predecessors.

If there is one thing that distinguishes the Web as a hypertext system from the systems that preceded it, it is the Web's adoption of a single, global, unified namespace. I believe that the leading Web Services will also come to adopt this model and it is merely a question of whether they do so sooner or later.

That point bears repeating: I believe that the *primary* thing that distinguished the Web from hypertext systems that languished in obscurity is its strong support for a single, global, namespace. If this is true, then it does not bode well for web services that ignore that fact.

7.2 SOAP and XML Technologies

SOAP does have a binding to HTTP. And that binding inherits the ability to use a URI-based model from HTTP. This binding also inherits the standardized HTTP methods. So there is hope that these SOAP/HTTP services will use URIs in a deep way. At this point it is primarily a question of tools and proper support in the service description languages.

If SOAP-based services do not migrate to a URI-based model then they will find that they are out of sync with the variety of W3C technologies. For instance it will be impossible to use RDF to assert things about resources that do not have URIs. It will be impossible to use topic maps and XLinks to build relationships between them. It will be impossible to use RSS to track changes to the objects. It will be impossible to use XPointer to point into their XML representations and to use XInclude and XSLT's "document" function to integrate them.

In many ways the standard SOAP model is at odds with the basic ideas of what you might call XML philosophy. The XML family of standards all presume that that all useful information is available in an XML form at a standardized address. Web services technologies are almost always used to set up alternate address spaces and these address spaces are essentially never URI-addressable. Examples of these alternate address spaces include UDDI and Microsoft's .NET My Services. These address spaces are fundamentally incompatible with URI-centric XML technologies.

The loss of RDF and other semantic Web technologies is particularly grievous. These technologies could be used to greatly improve the extensibility of web services and to help them support advanced logical features such as subclassing and inferencing. They could also become the basis for declarative business rules and distributed discovery for web services.

In short, the semantic web technologies have the potential to fill many of the holes in the web services picture. But they can only work with web services that identify resources with URIs. Web services can only identify resources with URIs if the web service toolkits allow them to. Web service toolkits will hopefully follow the lead of the standards and there is hope for further progress on that front.

7.3 SOAP and Web Middleware

There is a reason for SOAP's weakness around addressing. If SOAP unambiguously stated that the addressing syntax for SOAP is URIs then that would be equivalent to saying that SOAP is designed for use on the Web and only on the Web. But SOAP advocates are quite clear about the fact that Web protocols and Web addressing models serve only as "transports" for SOAP. That is like saying that the Web is a taxi driver and its only job is to get SOAP from place to place. The SOAP message can get out of the taxi and into another vehicle for the next leg of its trip. In technical terms, SOAP "tunnels" through the Web. This is not what the Web was designed for, nor what it is good at.

The REST viewpoint is that the Web is itself an incredibly scalable and well-designed information system. It was explicitly designed to integrate disparate information systems. But the Web does so on its terms, by binding them all into a single namespace and encouraging them to use a single protocol.

Web-izing two systems consists of making it such that the two systems can then talk to each other by virtue of the fact that they are using common namespaces, protocols and formats. In the "transport" model you just connect two services and those services define the method semantics and addressing model. In the second model you use the Web as a sort of "integration bus" with predefined method semantics and addressing model.

Many SOAP/WSDL advocates are quite open about the fact that they view the Web as RPC or message-oriented middleware. At O'Reilly's Emerging Technology Conference in spring 2002, Rohit

Khare (a SOAP advocate) asked a group of panellists what Web Services had to do with the Web. The unanimous answer from the panel was "nothing".

Dave Winer is one of the creators of the original SOAP. He maintains that it is a "web technology" but at the same time says that he sees it as a way to allow scripting languages to talk to each other across the network. In other words, an extension of a scripting object model across the network. Don Box has been quite open about the fact that he saw SOAP as a way to do DCOM over the Web without getting caught in firewalls. In other words, they both saw SOAP as RPC middleware that uses Web protocols. Although SOAP has changed over the years it has not lost this fundamental goal of being as unobtrusive and undemanding as possible.

It is fine to think of the Web as middleware. But it is a new kind of middleware. In particular it is a kind of unified, global middleware where everybody maps their systems into the same model, using the same addressing scheme, a small number of globally understood protocols, shared information formats and XML vocabularies. It makes little sense to adopt the least powerful parts of Web infrastructure (the syntax of the HTTP POST method) and ignore the most successful and powerful part: the URI namespace.

7.4 Mechanism versus Policy

There is a deep divergence between the approaches of REST and SOAP advocates in the relationship between "mechanism" and "policy".

SOAP development is driven in large part by software vendors. They want the move to a Web Services world to be easy for their customers. For this reason, SOAP philosophy tends to attempt to wrap legacy stuff in as unobtrusive a layer as possible. Conversely, the Web philosophy is to try to wrap legacy stuff in a very rich layer, with standardized methods, standardized addresssing models, etc. There is no question that this is difficult and expensive. Entirely new platforms (e.g. WebSphere and WebLogic) and languages (e.g. ASP, PHP, Java) have arisen to help with this task.

Even though this process has been very lucrative for software vendors, it is arguably the case that they could never have designed the Web. Good software vendors are customer-driven. Their customers would have complained that the system was too incompatible with their legacy systems. "You're saying we'll have to learn NEW programming languages like ASP, PHP and JSP, new programming models and new application protocols just to interface with this Web thing? No way!" "Project Web" would never have gotten off of the drawing board.

"Project Web" only became viable as a part of a responsible business after the non-commercial users had created a critical mass of customers who demanded Web access to information. By that time, the standards were set and there was no way to make them more "legacy friendly." So companies hired the webmasters, web designers, ASP, JSP, Perl and Java programmers and figured out how to map their data into this information system built by academics.

SOAP is designed not by academics but by commercial software developers for their customer. It always holds out the hope that you can keep using your legacy protocols, keep using your legacy addressing models and just generate a web service wrapper for them. Consequently, SOAP takes a point of view that it provides *mechanisms* for communicating between computers, not *policies* about how to do so.

At every point where there has been controversy about the applicability of the SOAP specification to a new problem domain, SOAP has grown. It has never risked telling people that they must figure out how to map their problem onto the existing rules or perhaps even choose a different protocol. "Keep your current diet, maintain your current exercise schedule - but we'll help you lose five pounds per week."

Conversely, Web Architecture is all about enforcing policies. The Web point of view is that you cannot build an information system without enforcing some rules. In fact, the most prescriptive parts

of the SOAP specification only take effect when SOAP is used over HTTP and were pressed upon the SOAP developers by REST advocates.

"Mechanism, not policy" results in a more flexible design. The phrase is most strongly associated with the famously pliable X-Windows system. Nobody could accuse either X-Windows or SOAP of being inflexible. But standardization is precisely the formulation of policies. Without at least some policies there can be no interoperability. X-Windows is also famous for its wildly divergent user interface conventions and poor interoperability on issues as simple as cut-and-paste.

Similarly, SOAP does not provide a canonical notion of thing/object/entity nor an addressing mechanism for referring to "things". It ignores the long-established policy of Web architecture that "things" are called "resources" and that they are addressed by URIs. In my opinion, it leaves a vacuum where a set of policies should be.

There is still an opportunity that WSDL can correct this flaw. Another potential outcome is that REST advocates could formally define some sort of profile of WSDL and SOAP that enforces the policies necessary to build an information system rather than a mere collection of services.

§ 8 What Can SOAP Teach REST

I've made the case that SOAP ignores important things we've learned from the XML and Web experiences. And yet SOAP continues to gain support in industry and with vendors.

REST people should see SOAP as a question, rather than an answer. The question is: "how do we make the Web do Web Services things." The fact that people feel they need SOAP indicates that they are dissatisfied with HTTP. Therefore it is valuable to catalog the things that SOAP is supposed to add beyond the contributions of HTTP.

There are a few themes that arise when REST advocates talk to SOAP advocates.

Asynchrony	This often means different things to different people but the most common thread is that two communicating programs (e.g. a client and server) should not be need to stay in constant communication while one of them is doing a lengthy calculation. There should be some way for one participant to call back to the other. SOAP does not directly support this but it can nevertheless be used in an asynchronous manner by piggy backing on an asynchronous transport. This is not yet standardized but will be one day. There are a variety of approaches to doing asynchrony in HTTP, because there are a variety of aspects to asynchrony. It has been proposed that these should be standardized. One such specification is called "HTTPEvents".[HTTPEvents]
Routing	HTTP messages are already routed from clients to proxies to servers. This is a sort of network-controlled routing. There is also a case to be made that sometimes it makes sense for the client to control routing explicitly by defining a path between nodes. Some REST researchers are experimenting with variants of SOAP Routing for this.
XML	We are now in a situation where XML use is a requirement in the creation of new specifications whether it adds value or not. The working group that is standardizing SOAP within the W3C is called the "XML Protocol Working Group." In the current environment it is likely that pre-XML technologies will be reinvented as XML technologies merely because it is trendy.
Reliability	Reliability is another word that means different things to different people. Most people mean once and only once delivery of messages. This is relatively easy to achieve using HTTP, but it is not a built-in feature.

Security	Although HTTP has security features, it will be useful to compare them to the features that arise as part of WS-Security.
Extensibility Model	SOAP has an extensibility model that allows a SOAP message creator to be very explicit about whether understanding of a portion of the message is optional or required. It also allows the headers to be targeted at particular intermediaries (e.g. proxies, caches). There is an extension to HTTP with many of the same ideas (and in fact the SOAP version is probably based upon the HTTP version) but it is not as well known nor as syntactically clear.
Service Description	SOAP has WSDL but HTTP has nothing similar. Because REST is an document-centric model, REST service description may grow out of schema languages and in particular semantic schema languages like DAML. It may also be possible to use the next version of WSDL in a constrained manner as a type description language for HTTP resources.

§ 9 Can SOAP Win?

New application protocols have traditionally been adopted only rarely. There is a strong tendency to try and bend existing protocols to new uses rather than deploy new ones. For instance the FTP protocol is in many ways a bad choice for a Web protocol (it is user-centric and connection-oriented whereas most web sites are anonymous and connectionless) but it is nevertheless relatively popular for distributing large files on the Web.

Why is the world so reluctant to adopt new protocols? One theory is that it is difficult to program for new application protocols. This would imply that what we need to do in order to encourage creativity in the protocol space is distribute tools like Visual Studio.NET that help to remove much of the drudgery. Another popular protocol in this space is XML-RPC, which makes it trivially easy to invent new protocols for use in dynamically typed programming and scripting languages.

An alternate theory is that the problems in deploying new protocols come down to issues such as adminstration, security, trust, extensibility and reliability. If this is the case, you might argue that the slow deployment of new protocols is actually a wise reaction to the issues raised - and in any case inevitable.

There is one kind of protocol that is relatively easy to deploy: it is one where the same company maintains all of the software running on all of the nodes. For instance if you were gluing together two internal systems within a corporation, then you could upgrade the protocol merely by updating the programs running on either side. Once you get to ten or twenty systems using the same protocol it becomes more tricky but as long as the systems are still running under a single administrative roof, the problem is managable. Similarly, Microsoft can easily upgrade the MSN Messenger protocol by telling its customers that the service has changed and they must upgrade within the next six months or lose connectivity. Most widely deployed peer-to-peer protocols seem to still be in this category. Their creators are typically able to force upgrades on their users from a central location.

When you cross organizational boundaries and get into deployments of thousands or tens of thousands, however, there is no opportunity to upgrade through either consensus or coercion. No organization could decide that it is time to phase out HTTP 1.0 now that HTTP 1.1 is available. If we could start from scratch, mail protocols might also be radically different. But we can't.

New SOAP-based application protocols will therefore have a huge hurdle to overcome before being deployed and will have even larger challenges adapting and evolving once they are deployed. It remains to be seen whether these can be overcome.

The picture is brighter within organizations. There will probably be several years of successful deployments within organizations. Many view SOAP as just a standardized, new-fangled form of

DCOM or CORBA. SOAP should be at least as successful as these technologies were in easing the point to point integration of internal systems. If, however, a REST alternative becomes dominant on the public Internet, it will inevitably seep into corporate internal systems as the Web did.

§ 10 Can REST Win?

Electronic business is going to need more than RPC-oriented or message-oriented middleware. All businesses everywhere will have to standardize their addressing models and expose common interfaces to their business partners. SOAP does not do that by itself and arguably it hurts more than it helps.

In order for businesses to interoperate without manually programming explicit links to their partners, they will need to standardize on protocols rather than inventing proprietary ones. They will have to standardize on addressing models, rather than inventing proprietary ones. REST promotes this high degree of standardization. Arguably, Web services will remain balkanized *until* something like REST catches on. If this is true (for many people that is a "big if") then it is not a matter of whether SOAP or REST will win, but rather whether SOAP will survive into the age of REST-architecture Web Services.

§ 11 Can REST and SOAP Co-exist?

As I have said, SOAP has recently been stretched in some REST-like directions. It also retains its original RPC core and other paradigms it has picked up through the core of its evolution. SOAP requires very little of people who intend to use it and can therefore be used in a REST-like manner or in other ways. The question therefore is not really whether SOAP and REST can co-exist. They can. But if the REST-centric view of the world is correct then the whole SOAP project could be viewed as misguided. Why do we need a protocol framework for tunnelling new protocols over arbitrary transports when we already have all of the protocols we need?

So the real question is which has the right philosophical underpinnings. In other words, do we need many protocols or just one? Do we need many addressing models or just one? Can interoperability be achieved in a many protocols, many addressing models system? Is the goal of unifying everything under a single protocol and addressing model unrealistically idealistic? Should we expose "services" or eliminate the boundaries between services and merely expose resources? Can the legacy issue be handled entirely with gateways? Are the answers to these questions different when integrating applications within organizations versus across organizational boundaries?

Each side believes they know the answers to these questions. Only time will tell which vision is ultimately correct.

Notes

1. CRUD is not the most appealing terminology but nevertheless widely used jargon!
2. In the movie "The Highlander", the constant refrain among supernatural warriors is that there can be only one. The one winner inherits all of the strength of the others.
3. Telemarketers would find their job a little harder too, but I think that they would adapt!

Bibliography

[BernersLee] Berners-Lee, Tim. The World Wide Web: Past, Present and Future, http://www.w3.org/People/Berners-Lee/1996/ppf.html

[DonBoxInInfoworld] Don Box Interview in Infoworld, http://www.infoworld.com/articles/hn/xml/02/06/06/020606hnbox.xml?s=rss&t;=news&slot;=3

[DonBoxTechnetcast] Doctor Dobb's Journal Technetcast, http://technetcast.ddj.com/tnc_play_stream.html?stream_id=416

[Fielding] Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000, ://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[HTTPEvents] HTTPEvents Proposal, http://internet.conveyor.com/RESTwiki/moin.cgi/HttpEvents

The Author

Paul Prescod
ConstantRevolution Consulting
Vancouver
BC
Canada
paul@prescod.net
http://www.constantrevolution.com

Paul Prescod is a researcher and implementer of markup technologies. He worked within the XML Working Group of the World Wide Web consortium to develop the XML family of standards and co-wrote the most popular book on that family of standards: The XML Handbook. Paul is an enthusiastic proponent and implementer of open source technologies. He is especially well-known for his Python advocacy. His formal education was in mathematics and computer science from the University of Waterloo. His research interests include formalisms for document modelling, queries, protocols, and schemata.

Extreme Markup Languages 2002
Montréal, Québec, August 6-9, 2002
This paper was formatted from XML source via XSL
Mulberry Technologies, Inc., August 2002