Myths around Web Services

Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ), Switzerland
www.inf.ethz.ch/department/IS/iks/

1 Introduction

Web services and the technology surrounding them have become the dominant trend in the electronic commerce arena. XML, SOAP, UDDI, and WSDL, as the foundation of Web services, are all attracting considerable attention as potential bridges between heterogeneous systems distributed across the Internet. The assumption seems to be that soon most applications will speak and understand XML, that all systems will support SOAP, that everybody will advertise their services in UDDI registers, and that all services will be described in WSDL. Once that stage is reached, application integration and business to business (B2B) e-commerce will be straightforward.

Unfortunately, Web services are only the next step in the natural evolution of middleware. Therefore, by design, Web services are evolutionary rather than revolutionary. The most basic form of middleware are RPC engines. When such engines become transactional, they become TP-Monitors. Once object orientation aspects are included, TP-Monitors evolve into Object Monitors. Message oriented middleware (MOM) also originated from TP-Monitors since persistent queuing was a feature of many TP-Monitors until they became systems on their own. In fact, many MOM platforms are RPC based. Web services are, primarily, an extension to middleware platforms to allow them to interact across the Internet. Only from this perspective do many of the developments in the Web services arena make sense, e.g., that one of the first protocols to be wrapped as SOAP messages was RPC (and, at the time of writing, almost the only one to have been completely specified).

Of course, it is possible that Web services will trigger a radical change in the way we think about middleware, application integration or the way we use the Internet. In that case, Web services will evolve into something yet unforeseen. At this stage, however, this has yet to happen. In reality, and precisely because they were created with that purpose in mind, Web services are used today almost exclusively for conventional enterprise application integration (which may or may not happen in a B2B setting). It is this experience as an extension to middleware platforms that will define and shape Web services in the short and medium term.

There are nevertheless many proposals that take Web services well beyond their current capabilities: semantic web, dynamic marketplaces, automatic generation of B2B applications, seamless integration of IT infrastructures from different corporations, etc. These proposals are the basis for presenting Web services as revolutionary rather than evolutionary. Such speculations are the province of long term research but they tend to ignore the exact nature of Web services and the underlying technology. Many of these ideas also ignore the current limitations of existing middleware platforms although most of these limitations appear again at the Web service level. In this regard, Web services have to a certain extent become an outlet for ideas that proved impractical in the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

past. The result is a series of myths around Web services that make Web services, already quite a complex set of ideas *per se*, even more difficult to comprehend, understand, and analyze.

In this paper we discuss some of these myths and try to clarify how much truth there is in them. The goal is to bring to the fore a few fundamental aspects of Web services and discuss in detail their practical implications. These considerations can be seen as both a warning against expecting too much from Web services as well as directions for future research since they are all open problems that remain to be solved before Web services can be used in their full generality. The overall message is that, with the exception of standardization efforts (already a significant step forward), today's Web services may have little to offer over conventional middleware.

2 A quick overview of Web services

Web services are a series of specifications around a very generic architecture [Kr10]. This architecture has three components: the service requester, the service provider, and the service registry, thereby closely following a client/server model with an explicit name and directory service (the service registry). Albeit simple, such an architecture illustrates quite well the original purpose of UDDI, WSDL and SOAP. In all cases, the information managed by these specifications is in the form of XML documents.

UDDI The service registry is based on the UDDI specification (Universal Description, Discovery, and Integration). The specification defines how to interact with a registry and what the entries on that registry look like. Interactions are of two types: registration and lookup. Registration is the procedure whereby new service descriptions are added to the registry. Lookup corresponds to queries sent by service requesters in search for the right services. The entries contain three types of information: white, yellow and green pages. The white pages contain generic information about the service provider (e.g., address, contact person, etc.). The yellow pages include categorization information that allows the registry to classify the service (e.g., flight reservation, search engine, or bookstore). The green pages contain information about the services interface and pointers to the service provider (where the actual WSDL interface definition can be found).

There are already several UDDI registries maintained by software vendors. These public registries are meant as low level, generic systems supporting only the most basic of interactions. The underlying idea is that more sophisticated repositories (e.g., with advanced query capabilities) will be built on top of UDDI repositories. Such *service databases* are, however, not part of the specification. UDDI also describes how to interact with a repository using SOAP. Such support is intended not so much for dynamic binding to services (in the middleware sense) as for developers building advanced service databases and other applications on top of UDDI repositories. Finally, there are two types of UDDI registries: public and private. Public ones are accessible to everyone and play the role of open search engines for Web services. Private ones are those that companies or group of companies create for their own use. For obvious reasons, industrial strength Web service implementations are likely to be based on private repositories rather than on public ones. It remains to be seen to what extent private repositories use UDDI as much of its functionality is not needed for private use.

WSDL The interface to a Web service is defined using WSDL (Web Services Definition Language). By using WSDL, designers specify the *type* system used in the description, the *messages* necessary to invoke an operation of the service (and their format), the *operation* protocol (whether it returns a response, etc.), the *port type* or set of operations that conform an instance of a service, and the binding or actual protocol to be used to invoke the operations of an instance of a service (e.g., HTTP). Note that what is known as *service* is a logical unit encompassing all port types mapped to the same logical service (e.g., flight reservations through RPC or through e-mail, each one of them being a port type of the flight reservation service).

SOAP Interaction between requester, provider and registry happen through SOAP (Simple Object Access Protocol). SOAP specifies messages as documents encoded in XML divided into two parts: header and body. Both the header and the body can be subdivided into blocks. Header blocks carry information pertaining the interaction: e.g., security, authentication, transactional context, etc. Body blocks store the data used in the interaction, e.g., which procedure is being called, each individual parameter, etc. SOAP also defines bindings to actual transport protocols. A binding specifies how a SOAP message is transmitted using, e.g., HTTP.

SOAP can be best understood when it is considered as the specification of a protocol wrapper rather than a communication protocol itself. The main point of SOAP is to provide a standardized way to transform different protocols and interaction mechanisms into XML documents. As such, each concrete protocol needs a SOAP specification. An example is the specification of how to use RPC over HTTP. The specification describes how to encode an RPC invocation into an XML document and how to transmit the XML document using HTTP.

The syntax for these specifications is based on XML. Nevertheless, WSDL, and SOAP support alternative type systems. In all cases, WSDL and SOAP should be seen as templates rather than strict standards. The fact that two Web services use WSDL and SOAP does not immediately make them compatible. One of them could be an RPC service accessible through HTTP. The other could be a batch service accessible through e-mail and using EDIFACT. Both are Web services compliant with accepted specifications and yet perfectly incompatible.

3 Myth I: Web services and standards

The hype around UDDI, WSDL and SOAP has eclipsed many parallel (and previous) efforts along the same direction. As a result, there is an obvious trend towards systems that are UDDI, WSDL, and SOAP specific. Such trend thrives on the myth that Web services are an accepted and dominant standard. However, it is by no means clear that Web services will displace existing technologies. Christoph Bussler has pointed out the fact that *standard* no longer means *globally unique* in the B2B world. He predicts that, in addition to UDDI, WSDL and SOAP, up to a few hundred competing B2B standards may coexist [Bu01]. Examples of such established standards that will not simple go away are the Electronic Data Interchange (EDI) [EDI], used in manufacturing, and SWIFT [SWI], used in the financial world. Consequently, Bussler recommends developers to be agnostic towards B2B standards and has also shown how the architecture of such generic systems should look like [Bu02].

Generality is certainly a solution to the lack of standardization. If no standard dominates, a generic architecture can be easily adapted to whatever specification comes along. Unfortunately, generality comes at a price and undermines the standardization efforts. The reason is that, in practice, Web services are not being built from scratch. They are being built on top of existing multi-tier systems, systems that are all but general. Hence, many Web services are biased from the start towards specific protocols, representations, and standards, i.e., those already supported by the underlying middleware. The necessary generality will only be achieved, if at all, by yet additional software layers. Even the WSDL specification has allowed for such generalization by providing alternative entry points to a given Web service (the *port types* in the WSDL jargon). What is then wrong with this picture? Michael Stonebraker argues that there is already too much middleware with competing and overlapping functionality [St02]. He defends the need for integrating middleware functionality in just a few system types, namely data federation systems, enterprise application integration (EAI) and application servers. Otherwise, the sheer complexity of the IT infrastructure becomes unmanageable. This is an argument often repeated in the middleware arena [Be96, AM97], where real convergence was starting to take place. Web services, however, add new layers to the already overly complex multi-tier architecture typical of B2B interactions. Aiming for generic systems will make matters even worse. Translation to and from XML, tunneling of RPC through SOAP, clients embedded in Web servers, alternative port types, and many of the technologies typical of Web services do not come for free. They add significant performance overheads and increase the already extreme complexity of developing, tuning, maintaining and evolving multi-tier systems.

Taken together, these two concerns configure a difficult dilemma for Web services. The proliferation of competing standards, whether based on the same syntax (XML) or not, will require additional software layers to address interoperability problems. Even in those cases where a single set of standards can be used, web services are being almost universally built as additional tiers over existing middleware platforms. Unfortunately, multi-tier architectures are already too complex and cumbersome. Adding more layers will not make them any better and the sheer complexity and cost of such systems may prevent the widespread use of the technology. Without widespread use, standards will fragment even further, thereby making it almost impossible to produce generic enough platforms which, in turn, increases again the development and maintenance costs. The resulting vicious circle can be the Achilles' heel of Web services as there is no obvious way out of it.

4 Myth II: Web services in conventional applications

One of the drawbacks of Web technology is that it is still too tightly related to humans and browsers. Web services have computers as their main users and are not based on browsers at all. Nevertheless, many of us still think about Web services in the same terms we think about a Web browser: our first image of a web service is that of an interactive one. Maybe with the execution driven by a computer instead of a human but interactive nonetheless. The examples available in the literature, and not only in the research literature, corroborate this bias. We have all seen many different variations of the traveling planner service, which has been misused so often that it should become a standard on its own. Flight reservations, car rental and hotel booking, or buying a travel guide, are all examples of interactive services. Moreover, all these services are typical Business to Consumer (B2C) interactions, rather than B2B exchanges. This is an interesting development since Web services are being pursued because of their potential impact on B2B not on B2C.

There are of course practical advantages in using Web services interactively and on-line. One example often mentioned are applications that embed a search engine by using Web services [Sh02]. Other examples are applications or operating systems that send periodic bug reports to the software vendor using a Web service, applications that automatically download and install patches, or systems that use a remote service to provide functionality that cannot be provided locally (e.g., access to a very large database that is not locally available). These are all very appealing scenarios but it is not immediately obvious that Web services are the best way to implement them. In some cases (e.g., information flow from the application to a server), this functionality is already being provided without Web services and it is not clear that switching to Web services will bring any significant advantages. In other cases, it does not seem reasonable to bloat the application with the whole machinery of Web services to implement just a fancy feature. If the operating system eventually provides support for accessing Web services to all applications, then this may make sense but we are quite far from that stage. Perhaps an even more decisive factor is that many of the features of Web services are irrelevant in these settings. For instance, application specific information does not need to be sent as an XML document. Likewise, interfaces used internally by a software vendor do not need to be described using WSDL (and certainly do not need advertising using UDDI).

From a practical perspective, it is also not clear how to build applications that rely on Web services for part of their functionality. The problem has been recently analyzed by Clay Shirky [Sh02]. He points out that Web services are still *plumbing for the exchange of XML documents using SOAP*. For interactive and on-line use within applications, he identifies several crucial issues that remain unsolved. One of them is trust: how far can the application trust and rely on external Web services which it does not control? Another one is the fact that we do not yet understand the impact of Web services on software design as many of the techniques for component based software development do not work with Web services. Answers to these questions are needed before Web services are widely used as extensions to conventional applications. As Shirky also points out, it will not be a trivial endeavor. It will require a whole new software engineering philosophy and tools that will not be available any time soon.

5 Myth III: Direct connectivity across corporate boundaries

Another myth resulting from ignoring the complexities of application integration and software design at a large scale is the claim that Web services provide a direct link between middleware platforms of different corporations. Most conventional middleware platforms are implemented on top of RPC: TP-Monitors, Object Monitors, CORBA implementations, and even message oriented middleware. Because of its pervasiveness, RPC over HTTP was one of the first interaction mechanisms specified using SOAP. By doing so, a Web service becomes an extension of existing multi-tier architectures but with the client residing now at the other side of the firewall and behind a Web server. Since B2B services are implemented using multi-tier systems, being able to use RPC through SOAP is seen by many as a gateway to interconnect the IT infrastructure of different companies.

There are several problems with such an interpretation. One is that RPC results in a tight integration that makes components dependent on each other. This is unacceptable in any industrial strength setting, specially if the components belong to different companies. Not only would the complexity of the resulting system increase exponentially, the mere act of maintaining the system would become a coordination nightmare with tremendous costs. This is why the vast majority of B2B interactions happen asynchronously and in batch mode, not interactively. Rather than direct invocations, requests are batched and routed through queues. Responses are treated in the same way. The actual elements of the interaction (client and server, to simplify things) are kept as decoupled as possible so that they can be designed, maintained, and evolved independently of each other. Systems based on EDI and SWIFT are, again, good examples of the typical loosely coupled architectures of B2B systems.

Proof of this is the strong trend towards asynchronous SOAP. The fact that the most widespread use of SOAP is to tunnel RPC does not contradict this statement. Many queuing systems are implemented on top of RPC. A message is placed on a queue and a daemon makes an RPC call to another remote daemon that takes the message and places it on the receiving queue. Technically this is not only possible, it is a reasonable way of implementing B2B interactions. From the point of view of Web services, however, it means that the Web service description will be far more complex than an RPC invocation encoded as an XML message. The description may have more to do with the interaction mechanism (the queues) than with the service interface itself. In fact, in many cases, the actual service interface will not necessarily be made explicit. For instance, a service may simply indicate that it is a queue that accepts EDIFACT purchase order messages without describing such messages (since their format is already known to those using them).

6 Myth IV: UDDI and dynamic binding

An UDDI registry is conceptually similar to a name and directory server. There are, however, significant practical differences between the two, differences that tend to be ignored and lead to the assumption that an UDDI registry has the same purpose as a name and directory server. The result is the widespread assumption that dynamic binding will be a common way of working with Web services. This is far from being the case and there are two very strong arguments against this assumption.

From the point of view of functionality, UDDI registries have been created as standardized catalogues of Web services. The information they contain is intended for humans, not for computers. First, there is the problem of the semantic interpretation of the parameters and operations defined by the interface. These parameters indicate the expected type but not what the parameter actually means (e.g., a price is given as an integer but there might not be any indication of the currency used). There is also the issue of how to deal with exceptions and how to link them to the internal business processes. The service might also provide different ways to proceed depending on the outcome of intermediate operations. Only a person can make sense of this information while using it will require careful analysis and a significant design effort. Second, interactions between different companies are regulated by contracts and business agreements. Without a proper contract, not many companies will interact with each other. To think that companies will (or can) invoke the first Web service they find on the network is unrealistic. Web service based B2B systems will be built through specialists who locate the necessary services, identify the interfaces, draw the necessary business agreement, and then design and build the actual application with the Web service either *hardwired* into the application code or defined as a deployment parameter.

From the software engineering point of view, dynamic binding is a double edge sword. If dynamic binding is used simply to determine the location of a well defined service, it is indeed an useful feature. Any other form of dynamic binding makes it almost impossible to develop real applications. CORBA already provided designers with very fancy dynamic binding capabilities. An object could actually query for a service it had never heard of and build on the fly a call to that service. Such level of dynamism makes sense only (if at all) in very concrete, low level scenarios that appear almost exclusively when writing system software. Application designers have no use for such dynamic binding capabilities. How can one write a solid application without knowing what components will be called? It is nearly impossible to write sensible, reliable application logic without knowing what exceptions might be raised, what components will be used, what parameters these components take, etc. In its full generality, dynamic binding does not make sense at the application level and this also holds for Web services. In regard to dynamic binding as a fault tolerance and load balancing mechanism, in the context of Web services, the UDDI registry is simply the wrong place for it. UDDI has been designed neither with the response time capabilities, not the facilities necessary to support such dynamic binding. Moreover, the UDDI registry cannot do any load balancing nor any automatic fail over to a different URI in case of failures. It is simply not designed to do that. Such problems are to be solved at the level of individual Web service provider using known techniques like replication, server clustering, and hot-backup techniques.

UDDI registries will, thus, be used by programs only to the extent that service publishing will be automatic in many systems and search over an UDDI registry will happen through specialized added-value tools built on top the UDDI registries.

7 Myth V: all data will be in XML

XML is a blessing as a syntax standard. It allows to build generic parsers that can be used in multitude of applications, thereby ensuring robustness and low cost for the technology. Unfortunately, this significant advantage does not compensate for the fact that XML is a performance nightmare. There are also many data types that do not get along well with XML. e.g., anything that is binary or nested XML documents [Sa02]. In many cases, even if it is possible, there is no point in formating the application data as an XML document. We have already mentioned an example: a Web service implemented as a queue expecting EDIFACT e-mail messages does not gain much by having the message encoded in XML. In fact, it only loses performance and introduces unnecessary software layers.

XML encoding makes sense when linking completely heterogeneous systems or passing data around that cannot be immediately interpreted. It also makes sense when there is no other standard syntax and designers must choose one. When Web services are built based on already agreed upon data formats, then the role of XML is reduced to be the syntax of the SOAP messages involved. This is why there is such a strong demand for SOAP to support a binary or *blob* type. There are several ways of doing this [Sa02]: using URLs as pointers, as an attachment or with the recently proposed Direct Internet Message Encapsulation (DIME) protocol [Ni02]. Whatever mechanism becomes the norm, expect an increasing amount of Web service traffic to contain binary rather than XML data.

The use of binary rather than XML for formatting application data has a wide range of implications for Web services. First, it will provide a vehicle for vertical B2B standards to survive even if Web service related specifications become dominant. This is directly related to the discussion above on Myth I. In practice, Web services become just a mechanism to tunnel interactions through the Internet, their original intended goal. The actual interaction semantics will be supported by other standards, those use to encode the data in binary (e.g., once more, EDI or SWIFT). The question will then be whether Web services provide enough added value to justify the overhead. Second, related to Myth III, Web services implemented over binary data will describe only the interaction. They cannot specify the actual programmatic interface of the service as this is hidden in the binary document and, therefore, cannot be controlled by the Web services infrastructure. This will reduce even further the chances of having tightly coupled architectures built around Web services. Finally, related to Myth IV, Web services based on binary formats will increase the dependency on humans for binding to services as much of the information needed to bind to a service might be external to the Web service specification.

8 Conclusions

After the burst of the Internet bubble, several critical voices have been raised against the hype around e-commerce, e.g., [Co02]. Web services are, to a great extent, still riding that hype and generating their own. To keep things in the proper perspective, it is useful to understand the original purpose of a given technology. This does not prevent the technology from becoming revolutionary, but it helps to identify those that are merely evolutionary. Web services are, at the current stage, only a natural evolutionary step from conventional application integration platforms. In spite of the many grand ideas being proposed in both industry and academia, there is a fair chance that market forces and sheer practicality will keep the role of Web services to, indeed, mere plumbing for B2B exchanges.

Acknowledgments

Part of this work is supported by grants from the Hasler Foundation (DISC Project No. 1820) and the Swiss Federal Office for Education and Science (ADAPT, BBW Project No. 02.0254 / EU IST-2001-37126).

References

[AM97] G. Alonso, C. Mohan. WFMS: The Next generation of Distributed processing Tools. In: Advanced Transaction Models and Architectures. S. Jajodia and L. Kerschberg (Eds.). Kluwer Academic Publishers, 1997.

[Be96] Philip A. Bernstein. Middleware: A Model for Distributed System Services. CACM, Vol. 39 No. 2, Feb 1996.

[Bu01] C. Bussler. B2B Protocol Standards and their Role in Semantic B2B Integration Engines. IEEE Data Engineering Bulletin, Vol 24. No. 1, March 2001.

[Bu02] C. Bussler. The Role of B2B Engines in B2B Integration Architectures. Sigmod Record, Vol. 31 No. 1, March 2002

[Co02] T. Coltman et al. Keeping E-business in perspective. CACM, Vol. 45, No. 8, August 2002.

[EDI] United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport. http://www.unece.org/trade/untdid/welcome.htm

[Kr10] H. Kreger. Web Services Conceptual Architecture (WSCA 1.0). IBM. Available from: *http://www-4.ibm.com/software/solutions/webservices/pdf/WSCA.pdf*

[Ni02] Nielsen H. F. et al. Direct Internet Message Encapsulation (DIME). Internet draft, draft-nielsen-dime-02, June, 2002.

[Sa02] R. Salz. Transporting Binary Data in SOAP. Published on XML.com http://www.xml.com/pub/a/2002/08/28/endpoints.html

[Sh02] C. Shirky. Web Services and Context Horizons. IEEE Computer, Vol. 35 No. 9, September 2002.

[St02] Michael Stonebraker. Too Much Middleware. ACM Sigmod Record, Vol. 31 No. 1, March 2002

[SWI] S.W.I.F.T. SCRL. http://www.swift.com/