Web Services: What's Real and What's Not?

Kevin J. Ma

Web services, service-oriented architectures (SOAs), and all the attendant buzzwords and acronyms are on the lips of just about anyone interested in network communication. Vendors are scrambling to provide infrastructures that have enough capabilities to warrant putting at least one of these terms in the accompanying marketing literature. SOAs in particular are beginning to eclipse Web services as the "latest paradigm." But the ideas underlying SOAs are hardly new. The term may be chic, but the notion of replacing proprietary interprocess communication with abstracted, well-defined, and ubiquitously invokable services has long been the goal of system designers. And computer scientists have never ceased their quest for greater abstraction and further encapsulation at all levels, from programming languages to application interfaces.

Understanding the state of Web services is the first step toward implementing the architectures that support their use.

I suspect that SOAs are garnering interest, not because they are a novel approach, but because Web services are becoming sufficiently mature for developers to realize and reevaluate their potential. The evolution of Web services is at a point where designers can start to see how to implement true SOAs. They can abstract a service enough to enable its dynamic and automatic selection, and Web services are finally offering a technology that is rich and flexible enough to make SOAs a reality. But that is exactly why it might be premature to start marketing SOAs as a paradigm while no one is still completely sure what Web services can offer. If Web services are a de facto technology for implementing SOAs, deeply understanding the former must be a prerequisite for developing the latter. Unfortunately, since the inception of Web services, a barrage of hype has hidden many of the realities and issues surrounding them. Hopefully, this article will clear away some of the fluff.

"WEB" SERVICES

In the most literal sense, a Web service is any service provided over the Web, which by inference might include common gateway interfaces (CGIs), HTML forms, Practical Extraction and Report Language (PERL), Java scripts, and the like. In practice, however, most Web service designers view the "Web" part of Web services as a misnomer. Although most Web services rely on the Hypertext Transfer Protocol (HTTP) binding as a transport mechanism—hence the Web moniker—it is certainly not a requirement. In Gartner's definition, Web services use XML as the data format, Internet protocols for transport, and one or a combination of the Simple Object Access Protocol (SOAP), the Web Services Description Language (WSDL), or universal description, discovery, and integration (UDDI)—currently the three most recognized Web services standards (W. Andrews, "Web Services in Action," Aug. 2001, http://www3.gartner.com/resources/109000/109040/109040.pdf).

In large part, the term "Web" services evolved because HTTP emerged as the primary choice of

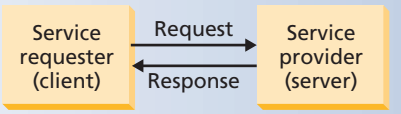
Inside

Published by the IEEE Computer Society

1520-9202/05/$20.00 © 2005 IEEE

transport binding. This is hardly surprising, given the prominence of the Web in the last decade. Academia has embraced Web-based registration, distance learning, homework submission, and research. Businesses rely on Web-based portals for online purchasing, electronic pay stubs, and database interfaces. Consumers rely on e-commerce and e-mail, and enjoy recreational surfing. Small wonder many everyday services have migrated to the Web and that by extension HTTP has become ubiquitous. It helps that HTTP has a robust infrastructure for both transport and security and a large constituency of developers and technicians. And although implementations of HTTP servers and clients are platform dependent, the ubiquity of the protocol across platforms makes HTTP itself as good as platform *in*dependent.

Figure 1. Service-oriented architecture in a simple form. In many ways, an SOA is merely a client-server architecture in a new context.



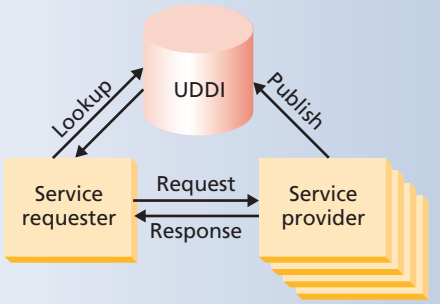
OLD IDEAS IN A NEW PACKAGE

Although many vendors view SOA as a unique paradigm, you can implement a SOA's functionality in many ways. Information is a commodity. For decades, techniques have been refined for acquiring and parsing information. Gathering data over the Internet has evolved quite a bit, from basic file retrieval via the file transfer protocol (FTP), to the dynamically generated content and the personalized GUI of Web pages today. The data has become much more sophisticated; it is no longer enough to just return static files. Web servers and browsers alike run scripts, invoke services, and process and collate data. Web services serve simply as a means for implementing remote procedure calls (RPCs). Ignoring the amorphous levels of abstraction, encapsulation, and extensibility, Web services, RPCs, and service or method invocation seek the same outcome. Web-enriched media content has extended the idea of autonomous transactions; Web services are just the next evolution in network-based information exchange.

Through this lens, SOA adopts a much more familiar look. As Figure 1 shows, the fundamental interaction of a SOA looks very much like the client-server architecture. A service requester (client) sends a message to the service provider (server), and the provider returns a response, either the requested information or confirmation that some action has occurred. The transaction might be synchronous or work via asynchronous call back, and it is protocol independent. Though some might consider this an oversimplification, in reality, the needs are fairly simple and the underlying paradigms should not be needlessly complex.

Web services provide a method for implementing the simple architecture in Figure 1, while offering additional features and extensibility. Figure 2 depicts the basic Web service architecture. The transaction starts with the service provider communicating its intent to offer services to the UDDI registry. The service provider stores a WSDL representation of the service in the registry to be distributed upon request. At some point later, a node wishing to consume a service contacts the UDDI registry to find a service provider. If the registry finds a match, it returns the WSDL description, which the client interprets. The client then formats a request and forwards it to the service provider, which returns a response. All the messages and data are in XML and encapsulated in SOAP messages.

Figure 2. Web service architecture. The lines represent Web service transactions between entities, most likely with an HTTP transport binding.



This is the general idea of how a Web service transaction should occur, although admittedly, the picture lacks some detail and some of the basic technologies are still too immature for general consumption. The next step is to abstract this transaction hierarchically, such that the service invoked can, itself, be one or more Web services. The Business Process Execution Language (BPEL), described later, provides just this functionality. As such, it offers a means of orchestrating Web service conversations, providing the automation aspect of SOAs. This, coupled with the dynamic nature of UDDI service resolution, represents a suite of technologies that with XML, SOAP, and WSDL are suitable for implementing SOAs.

KEY LANGUAGES AND TECHNOLOGIES

Many technologies enable Web services, and research is already focusing on some that might streamline future implementations. Although many of these technologies are maturing quickly, Web services in general are still in their infancy, so this selection represents only a snapshot of significant developments.

Extensible Markup Language

XML is a subset of Standard Generalized Markup Language (SGML) that allows the use of tags and attributes to format data. Unlike Hypertext Markup Language (HTML), currently the most prominent markup language, XML has no restrictions on the tags or attributes that mark up a document. Consequently, it is much richer and more descriptive than HTML, which defines a limited set of tags and attributes sufficient for describing only basic Web pages.

At present, SOAP seems to be the technology of choice for Web services.

One of XML's more useful features is its plain-text-based encoding, which means no arcane knowledge of proprietary data formats is required to decipher messages and data. This feature is also one of XML's biggest drawbacks. During the transmission of XML-based messages, the verbosity of this encoding method significantly increases bandwidth use, and its plain-text readability is a security concern. Service providers often employ secure sockets layer (SSL) encryption to add privacy and mitigate security risks. HTTP compression (usually with gzip encoding) can reduce bandwidth consumption, although it increases processing. Another alternative under investigation is the use of binary XML, which is the basis for products like Expway's BinXML series (http://www.expway.com). Nonetheless, although these other encoding methods are becoming more widespread, plain-text-based encoding is still the predominant method. The sidebar "Binary XML?" explains the advantages and disadvantages of text and binary XML.

XML schemas

Given this absolute freedom to define new proprietary markups, application designers required a way to place constraints on their document sets. With no way to validate their documents' conformance, applications that processed the documents were susceptible to data errors, malicious documents, unhandled cases, and the like. SGML's document type definitions (DTDs) offer a method for describing and validating documents, but they take validation only so far. XML schemas, written using XML, let document creators define simple and complex data types as well as constraints such as sequence order, number of occurrences, and valid data values. As such, XML schemas provide a way to conduct a basic integrity check of XML-formatted data. This capability makes them key to a Web service's application-level security.

Extensible Stylesheet Language Transformations

Extensible Stylesheet Language Transformations (XSLT) is a subset of the Extensible Stylesheet Language (XSL) that lets document creators define a translation to reformat XML documents. XSLT accomplishes this by creating scripts that specify text for insertion when the tag or attribute matches a rule defined in the script.

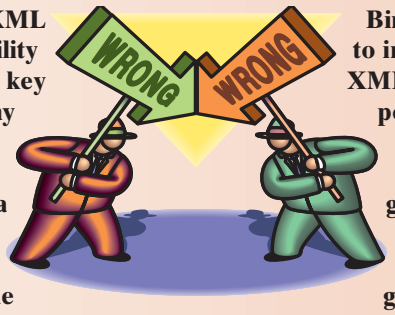
Initially, XSLT found use in reformatting XML into HTML with the aim of creating Web-browser-friendly representations of XML documents (similar to cascaded stylesheets). More recently, it has also found use in bridging gaps between legacy data formats and nascent XML encodings. It is suitable for performing primitive data processing functions to enhance aesthetics, such as converting an XML document into HTML for viewing through a Web browser, or to enable interoperability, such as converting FiXML back to Fix when communicating with a legacy server. Although XSLT is not necessarily directly related to service invocation, it can be critical to deploying a Web service if the desired response is not raw XML data.

Simple Object Access Protocol

At present, SOAP seems to be the technology of choice for Web services. It is a platform-independent, XML-based protocol for remote (or local) method invocation. SOAP defines an XML-based framework for unidirectional message exchange, accommodating the use of either HTTP or the Simple Mail Transfer Protocol (SMTP) as the transport mechanism. SOAP defines a message encapsulation format that comprises an <Envelope>, any number of <Header>s, and a single <Body>. It also specifies rules for processing <Header>s at intermediate hops but restricts <Body> processing to the end receiver. The <Body> contains the user's XML data, and the <Envelope> specifies the service to be invoked. The data in the <Body> must be in a structure that both the requesting and receiving parties understand. Typically, those interfaces are described in WSDL, and the requester has to make sure that the request conforms to the interface. SOAP also separately defines transport-layer bindings to specific, frequently used protocols. Currently, the only defined bindings are for HTTP and SMTP.

Services typically employ a request-and-response architecture. An HTTP session is inherently synchronous, as is the underlying TCP connection. The RPC-like transaction model is thus a natural extension of the synchronous, HTTP-based, SOAP messaging infrastructure. The requester will typically embed input data into a HTTP POST request and the service will respond with the data embedded in the HTTP response. SOAP also supports asynchronous messaging (through the use of Web service call backs), though they are less prevalent. In this case, the

Binary XML?



One of the founding principles of XML is its human readability and compatibility with SGML. Human readability is a key benefit because humans perform many of the design and debugging tasks in creating documents and testing applications. But although representing data in a way that humans find intuitive helps decrease the performance time of human tasks, it increases the machine processing time, since machines must parse and process the text. In other words, data designed for humans by humans is not optimal for machines. Computer scientists have long waged war to close the human-versus-machine thinking gap, and the latest battle seems to be between text-based and binary XML.

Binary XML exists in many forms, and proprietary formats are permeating monopolistic applications. The chief benefit is processing time. Businesses invest huge amounts in computers and data storage and demand peak efficiency from their capital expenditures. The wasted space, bandwidth, and cycles spent storing, transmitting, and processing human-readable XML that no human has read since its creation takes its toll on the bottom line. Reverting to a proprietary binary XML format to recoup time and money is a powerful business argument against the technological moral high ground—the one that says binary XML somehow subverts human productivity.

Some doubt the need to go to the other extreme, suggesting that a standard gzip encoding can compress text to an acceptable degree. Many storage devices provide gzipping functionality, and HTTP provides support for gzip as well. But although this solves the problem of disk space and bandwidth, it actually increases the required processing. Applications must gzip the data before writing it to disk or transmitting it, and then another operation must unzip it before the server can parse it. Some argue that riding the Moore's law curve more than offsets the extra processing required; others say that performance is still being left on the table.

Binary XML takes a different approach to increasing performance, and reducing XML document file size is only one component of the possible performance increase. The key to binary XML is that it is not a post-processed encoding, like gzip. Binary XML is a full representation of the document, which an XML parser can interpret. There is no "first gunzip then parse." There is just "parse." Gzip, while not lossy, uses backward pointers to reduce redundancy, which in effect rearranges the data and makes it hard to parse. Making the document itself smaller and retaining its order means that the server can parse it normally and thus faster.

Proponents of the status quo, beyond arguing that binary XML is just plain wrong and that human readability is key, feel that the existing XML infrastructure is too far along. It would take too long to create a standard and to upgrade applications, especially with the proliferation of homegrown binary XML formats. The task of developing a binary XML standard could become moot if proprietary formats become so entrenched that nothing can supersede them. Upgrading is an issue with most popular protocols, but bandwidth and computational power are commodities that command a premium.

The human readability argument goes something like this: Human readability increases worker productivity and permits a greater abstraction level, which means that if the XML version is intuitive enough, the worker's experience base becomes less important. On the other hand, workers and managers alike would probably argue in favor of paying workers for their familiarity with the product. And both would argue in favor of anything that increases data-processing throughput. I believe everyone would embrace a binary XML standard if it became available. A standardized way to convert from human-readable text to machine-optimized binary would, of course, please both sides. But until then, gzip and faster CPUs will have to do.

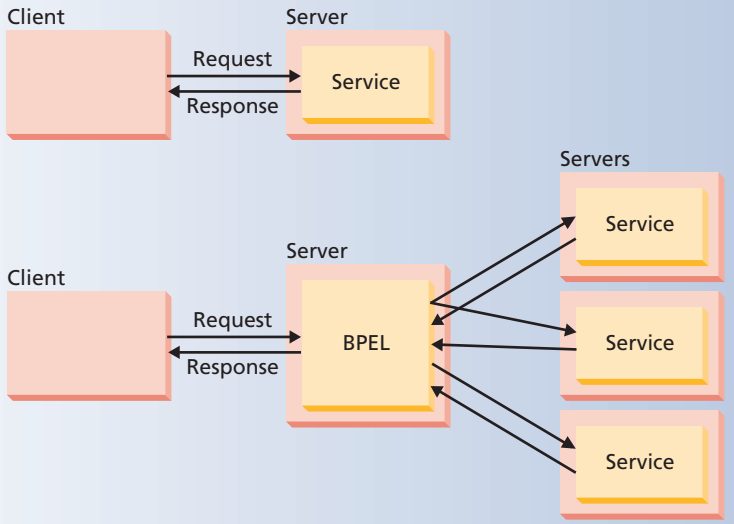
HTTP response would contain no data, and the service would respond with an HTTP POST of its own, at some later time. This call-back architecture does require, however, that the client have an HTTP server and a valid call-back service enabled.

Web Services Description Language

WSDL provides an XML grammar for describing all the pertinent information about a Web service, including its name, transport binding, and message format. Requesting agents can use the WSDL description to understand the particular service's interface, and UDDI uses WSDL for this purpose.

Work remains in refining the ability to interpret WSDL, however. WSDL describes the data structures involved in a Web service transaction, but the method a machine uses to parse and interpret user input is less defined, primarily because of the large gap in machine intelligence research.

Figure 3. Simplified diagram of how BPEL works. By describing conditional and parallel interactions among Web services, designers can build a service conglomeration.



Although WSDL can aid in verifying a constructed SOAP message, work remains to define methods for creating a message and populating the correct value using the information WSDL provides. A more practical application of WSDL is to use these high-level descriptions to automatically generate code stubs for methods and data structures, XML generation and parsing, HTTP messaging, and a Web services infrastructure. The result is a different abstraction level for Web services definition. A programmer will still be required for the full implementation, but service developers can ignore the end system's minutia during the service's design and investigation phase, which means that people other than software developers can help define, prototype, and implement the service.

Universal description, discovery, and integration

UDDI specifies a registry of Web services, similar to numbers in a phone book. Given a service's general description, it retrieves a list of matching services. Service providers can dynamically publish information and services, and service requesters can search through the registry to find personal information about a service provider as well as technical information about its services. The UDDI registry maintains a centralized database of Web services, classified by geography, service type, provider information, and so on; it also defines a standard API for requesters to query the database.

UDDI's intelligent searching mechanism is still in the early stages, much like the intelligent interpretation of WSDL. The ability to parse a human request for a service and correctly match it to a human description of a service without the give and take of human conversation is a thorny problem. Researchers and practitioners are continually refining the interfaces and semantics for creating a utopic UDDI, but work has yet to reach sufficient stability or maturity for wide deployment. In fact, it might not ever reach that state. Moreover, without this functionality, public UDDI's usefulness is low, relative to the other key technologies described here. However, private UDDI—where the same organization that produced the service deploys the service—is more useful. In this context, it offers a layer of abstraction that lets the client forego a hard-coded interface. Instead, the client can use a well-known identifier to search the UDDI registry for the most up-to-date interface information.

Business Process Execution Language

A relative newcomer to the Web services arena is BPEL, an XML-based standard that major software vendors—BEA, IBM, Microsoft, SAP, and Siebel—are developing. As Figure 3 shows in a simplified form, BPEL enables the abstracted orchestration of multiple Web services by letting designers describe conditional and parallel interactions among Web services. The response value of one or many services can trigger the invocation of the next service or services. BPEL lets developers define multiple paths through the script, given different responses. In effect, this means that developers can create large, conglomerate Web services by conditionally stringing together constituent Web services. Some of the popular examples include online purchasing and online loan approval, where the Web site uses Web service interfaces to process orders and applications. Online retailers can automate supply chain management through Web service interfaces that individual vendors—the warehouse, shipping company, auditors, bank and credit card companies, and so on—supply. Theoretically the service provider can describe the generic purchase transaction, essentially the interactions between these vendor entities, as a business process and encapsulate it within a BPEL script. However, much like UDDI, these types of interorganization transactions are not yet practical enough to be prevalent. For now, organizations can use BPEL in controlled, internal transactions—those that don't traverse the public Internet or share data with anonymous entities. They can script the retrieval and collation of data from separate entities like databases and mass storage, ensuring a high level

of security and reliability for these transactions that stay within the organization or a network of trusted partners.

Although BPEL is not a component technology of Web services, it is key to their prolifieration because it advances their scope. Web services can move beyond simple RPCs, isolated transactions, and singular messages to a richer and more dynamic conversation-based model. BPEL increases the level of automation and empowers Web service consumers with the ability to orchestrate large, process-oriented interactions.

Security

Web service security is undergoing massive research, with data security being one of the major topics. SSL and transport-layer security (TLS) provide trusted message-level security through Hypertext Transport Protocol Security (HTTPS), which is essentially HTTP over SSL. The ability to address XML's flexible nature, however, requires a more granular, policy-based security architecture. The idea is complex, encompassing user roles within a data file, each with different trust levels, defined by policies, and enforced through multiple cryptographic and authentication algorithms. A host of Web service standards attempt to better define these ideas. Specifications like WS-Trust and WS-Policy, and all the other ancillary addenda and extensions to WS-Security, are still far from complete. Even the more well-defined standards like WS-Security and the Security Assertion Markup Language (SAML) are still fighting for traction. Although developments in security will accelerate, at least for the near term, the well-understood and well-deployed HTTPS is a prudent choice for security.

WEB SERVICE IMPLEMENTATION

As the previous description implies, the ultimate goal of Web service technologies is autonomous interaction through intermachine communication; this goal is not yet a reality. Even without such intelligent agents and sophisticated registries, however, proprietary Web service implementations still benefit any who can use an existing client-server architecture. Some of the most obvious benefits are

- the ability to customize data representation and processing,
- interoperability with existing Web-based infrastructure,
- platform independence through standardized interfaces,
- access to SSL/TLS security through HTTPS,
- bandwidth subscription mitigation through HTTP compression, and

Resources

Web Services Trends

- **"Time Well Spent: Web Services Standards Mature More," C. Abrams and D. Plummer, Nov. 2004; http://www3.gartner.com/teleconferences/attributes/attr103788 115.ppt.**
- **"Migrating to a Service-Oriented Architecture, Part 1," K. Channabasavalah, K. Holley, and E. Tuggle, Dec. 2003; http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa/.**

Web Services Standards and Protocols

- **"XSL Transformations (XSLT)," J. Clark, ed., Nov. 1999; http://w3c.org/TR/xslt/.**
- **"The TLS Protocol Version 1.0," T. Dierks and C. Allen, Jan. 1999; ftp://ftp.rfc-editor.org/in-notes/rfc2246.txt.**
- **"The SSL Protocol Version 3," A.O. Freier, P. Karlton, and P.C. Kocher, Nov. 1996; http://wp.netscape.com/eng/ssl3/.**
- **"Security Assertion Markup Language (SAML) 2.0 Technical Overview," J. Hughes and E. Maler, eds., July 2004; http://www.oasisopen.org/committees/downloads.php/7874/sstc-saml-tech-overview-2.0-draft-01.pdf.**
- **"SOAP Version 1.2 Part 0: Primer," N. Mitra, ed., June 2003; http://w3c.org/TR/2003/REC-soap12-part0-20030624/.**
- **"Business Process Execution Language for Web Services Version 1.1," S. Thatte, ed., May 2003; http://www-128.ibm.com/developerworks/websevices/library/ws-bpel/.**
- **"UDDI Version 3," L. Clement and colleagues, eds., Oct. 2004; http://uddi.org/pubs/uddi v3.htm.**
- **"Web Services Description Language (WSDL) Version 2.0," D. Booth and C. Liu, eds., Dec. 2004; http://w3c.org/TR/2004/WD-wsdl20-primer-20041221/.**
- **"Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)," A. Nadalin and colleagues, eds., Mar. 2004; http://www.oasis-open.org/wss/2004/01/oasis-2004010-wss-soapmessage-security-1.0.pdf.**
- **"XML Schema Part 0: Primer," D. Fallside and P. Walmsley, eds., Oct. 2004; http://w3c.org/TR/xmlschema-0/.**
- **"Extensible Markup Language (XML) 1.0, 3rd ed.," F. Yergeau and colleagues, Feb. 2004, http://w3c.org/TR/2004/REC-xml-20040204.**

Web Services Infrastructures

- **Apache Web Services Project, http://ws.apache.org.**
- **gSOAP, http://www.cs.fsu.edu/ engelen/soap.html.**

Figure 4. gSoap's Web service tunnel paradigm.

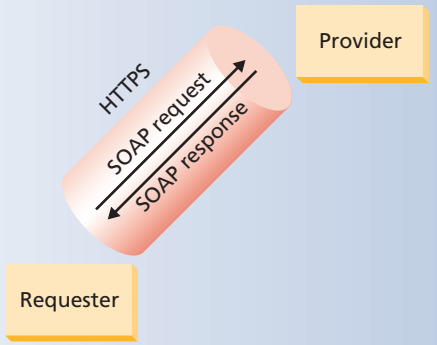
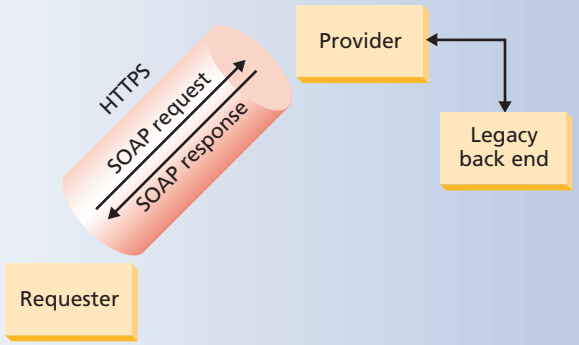


Figure 5. Apache's Web service proxy approach.



- the promise of a fully dynamic Web service architecture with role-based security policies and authenticated trust.

Issues

The ability to customize data structures in an easily decipherable format and still have a standard API for parsing and processing all proprietary data is a win-win scenario from the developer's perspective, although its benefits are hard to quantify overall. The idea of accessibility and scalability are also somewhat less tangible. The ability to easily integrate functions into a Web portal with a standard Java infrastructure means that services can become available to anyone with a browser. When offered through a ubiquitous interface, proprietary applications lose their limitations. Any Web-based method can achieve accessibility. The Web services approach can be as simple as formatting messages for transmit using a SOAP encapsulation. For developers, this standardized formatting is a boon and as the industry moves forward, the interoperability and data abstraction that Web services provide can mean less duplicated effort and thus higher efficiency.

From a more concrete standpoint, the ability to take advantage of the existing SSL/TLS infrastructure is another reason for the popularity of Web-based approaches. HTTPS is well deployed, well understood, and well supported. Although more-specific security standards for Web services are under development, SSL/TLS provides a tried and true mechanism for blanket security.

Most of these bonuses do not come without a cost, however. The overhead of plain-text versus binary encoding can be large. Descriptive XML tag names only compound the size bloat of messages. The SOAP encapsulation and HTTP headers are required as well, which can make data-transfer sizes quite large. The exact effect of this drawback is hard to gauge. The use of SSL/TLS security certainly adds additional messaging overhead for connection setup and teardown and can introduce a monumental increase in processing resources because of the costly nature of cryptographic operations. Moreover, an infrastructure based on platform-independent, interpreted languages (Java, PERL, or Python, for example) can be much less efficient than platform-compiled code, resulting in additional performance hits.

Options

Developers have many options for implementing basic Web services. All the major software vendors are offering infrastructures, including Microsoft with .Net, IBM with WebSphere, and BEA with WebLogic. These infrastructures are complete with custom applications and interfaces, integrated UDDI and BPEL support, and the like. For those seeking a less sophisticated (but also less expensive) approach, the open-source community also has some packages. A few of the more popular ones are Apache Axis Java, Apache Axis C++, and gSOAP C/C++. Each of these open-source options offers the ability to implement a simplified Web service architecture, but they are not as sophisticated as the packages that for-profit vendors offer. Their subset of features encompasses the core functionality of SOAP encapsulation, HTTP transport, SSL/TLS security, and Zlib compression.

Despite their lack of sophistication, these open-source options are more than adequate for most current applications, although the support level might not meet the needs of all Web service providers. Open-source options are cost effective approaches to software purchasing, but providers should also consider the frustration and delays that stem from sparse documentation and long development cycles. Fast, cheap, and functional is certainly achievable with open-source software and simple scenarios, but they might be less suitable for those developing complex services.

gSOAP. In 2001, a group at Florida State University developed gSOAP. The group still manages gSOAP, but has made it available to the open-source community through

Source-Forge. gSOAP provides good documentation, platform independence, and a compiler tool that does a nice job of generating most of the necessary code for the low-level SOAP infrastructure. Given a C header file (or WSDL definition file), gSOAP generates stub functions for constructing and parsing the XML data representation. It also includes built-in lightweight HTTP support, as well as an OpenSSL extension to support HTTPS and a Zlib extension to support HTTP compression.

Using gSOAP, developers can create custom end-to-end applications through the tunnel paradigm in Figure 4. gSOAP handles the TCP, HTTP, XML, and SOAP infrastructure behind the scenes. It automatically generates the XML code and provides a standard library for TCP, HTTP, SSL, and compression support. A complete tool, gSOAP in effect creates an interoperable and secure logical pipe through which to tunnel requests from the client to the server, in a complete end-to-end package. With all this infrastructure included, developers are free to concentrate on the application's true functionality and value.

Apache. In 2000, Apache began working on the Apache SOAP project, but abandoned it after beginning its Axis project in 2003. Apache's approach is slightly different from gSOAP's. Apache initially implemented the Axis package as a Java servlet, which sits atop the Apache Web server and Apache Jakarta Tomcat servlet infrastructure. Axis provides SOAP protocol support and service administration tools, taking advantage of the Apache Xerces XML processor for XML parsing. Apache's approach fits well with applications that run on a Linux-based Web server and provide services to a Web browser-based interface. Apache is well known for its open-source Web infrastructure and many ancillary projects. OpenSSL and Zlib support are included with Apache Web server.

With Apache, developers can create applications with more of the proxied transaction paradigm in Figure 5. This paradigm fits well with scenarios such as Web portals, SSL-based virtual private networks, and connections to legacy back ends. The architecture can exploit existing infrastructure to mitigate risk when migrating to Web services. The full-service Web server and Java infrastructure will, however, have greater resource requirements (for both memory and CPU) and thus affect performance more significantly, relative to the lightweight gSOAP infrastructure.

A POWERFUL ROADMAP

Web services have been around for quite a few years, but they are just starting to find their way into industry. The goals and ideals are not new: standards-based protocols, platform independence, conversational transactions, machine semantics, and machine autonomy. The problems are the same; it's just that technologies are finally making inroads toward solving them.

Web service technologies still need time to mature and require more research to realize their ultimate potential, but this does nothing to diminish their current usefulness. Both proprietary and standardized protocols offer many options for internode communication, and SOAP is just one of them. However, with industry momentum building behind XML and SOAP, Web services appear to be pulling ahead of the other options. Even if they never realize their full potential, Web services are still a viable means for implementing network communication. And they can be a roadmap for an incredibly powerful communication toolset and infrastructure. ■

Kevin J. Ma *is a software engineer at Cisco Systems. Contact him at kema@cisco.com.*

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.