Developing Web Services

Adam Bosworth
Crossgain Corporation
adambos@crossgain.com

Abstract

Web-based application developers are attempting to move towards Web Services as a mechanism for developing component-based Web applications. Unfortunately traditional tools and development models are inadequately architected to meet the rapidly evolving needs for the future of scalable Web Services. Today's Web Services development model is mired with complexity as traditional tools and technologies, focused on client-server application development, are ill suited. In this paper we examine the challenges of today's Web Service development model.

1 Introduction

Developing Web Services with traditional technology requires costly and scarce programming resources, and heavy investments in hardware and software that create lengthy and costly development cycles. These barriers to entry have delayed the rapid emergence of Web Services. Crossgain enables the Web Services Era through a uniquely productive platform service, development model and community.

1.1 What is a Web Services?

The term "Web service," describes specific functionality, value delivered via Internet protocols, for the purpose of providing a mechanism for another service or application to use.

Web Services enable the componentization and reuse of traditional web applications. An example is a Web Service to calculate sales tax. As a Web Service, applications have the ability to embed the sales tax function with other functions, or Web Services, to create an on-line consumer sales experience.

By exposing components of applications as Web Services, and enabling businesses to invoke these components, businesses can fundamentally transform their ability to interact and engage with both current as well as potential customers and partners. Web Services will fundamentally transform Web-based applications by enabling them to participate more broadly as an integrated component to an eBusiness solution.

The industry is attempting to take advantage of World Wide Web Consortium (W3C, see www.w3c.org) and Internet Engineering Task Force (IETF, see www.ietf.org) standards such as Extensible Markup Language (XML), HTTP, and Domain Name System (DNS) protocols to create specifications that define a way to publish and discover information about Web services. An example is the Universal Description, Discovery and Integration (UDDI) specification (see [1]).

1.2 Example Scenarios

Today, software applications deliver a wide variety of solutions to address a wide variety of customer and business needs. These same needs create a similar market for Web Services, which means that someday there will also be a wide variety of Web Services available. Here are three-example Web Service scenarios for business.

1. **Exposing Existing Infrastructure to the Internet**

Many companies already have significant investments in automating their business processes on back-end systems. One of the first steps these companies need to pursue is making these processes available to the appropriate partners, vendors, and customers on the Internet.

For example, suppose a shipping company wants to provide a Web Service that enables customers to determine what it costs to ship an item from point A to point B in a specified timeframe, to arrange for pick up and to track a shipped item. The business systems to track this information have already been developed and are used internally every day.

2. **Stand-Alone Internet Service**

The Web Service is a new application or functionality that is being developed specifically to meet the needs of a Web-based service.

An example of this type of service is a financial retirement analysis service. Anyone willing to pay can send their personal financial information and have a personalized retirement analysis done. This type of analysis is typically asynchronous, where the individual submits their personal information and awaits notification when the analysis is completed.

1063-6382/01 $10.00 © 2001 IEEE

For example, Step 3 is a potential bottleneck as the application resources sit idle and unavailable while waiting for the database to respond. This may deny customers service, as the Web server is unable to process additional requests. The idle time may also increase as the complexity, or the processing time increases. With a loosely coupled service, the application submits the query to the database then its resources are quickly freed to handle additional customer requests from the Web server.

2. **Unreliable Communications**

Because synchronous services assume reliable communications over traditionally unreliable communications mediums such as: Local Area Networks (LANs), Internet Service Providers (ISPs) and Dial-up links, the service may become slow or unreliable. Loosely coupled services can increase the speed or service reliability because messaging processing does not assumes an unreliable communications infrastructure. Messages are queued until they are successfully transmitted to the appropriate application queue.

A simple example is to contrast the use of the telephone, synchronous communications, to mailing a message, asynchronous communications. In order for the telephone to be an effective communications device, it assumes that there is another telephone user on the other end and can communicate back. Without both users on-line, or on the telephone, at the exact same time, User1 could not communicate with User2. With mail, since there is no assumption as to User2's availability, or that User2 can communicate back, User1 could create a piece of mail, or message, and send the mail to User2's mailbox and continue processing other messages for User3, User4, etc.... User2's message could be reliably delivered, irrespective of their current availability. User2 can read the message at anytime and choose to respond, or not, without impacting User1.

3. **Unreliable Access**

Because synchronous services assume system availability, unreliable access, like unreliable communications, may cause service slow-downs or failures that negatively impact the service performance. Loosely coupled services, not only don't assume a reliable communications infrastructure, they also don't assume service availability. This enables on-line, or running, services to continue processing requests while queuing off-line, not running, services for processing once they are back on-line.

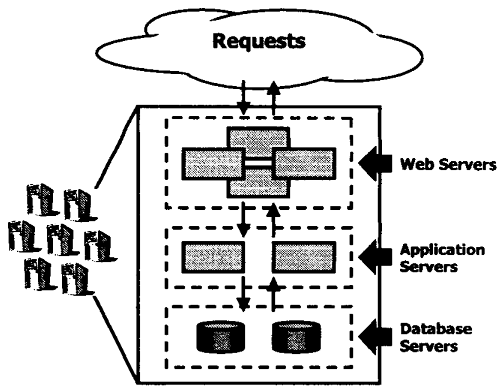
For example, in Step 3 the a service assumes the availability of the database and if the database is off-line, or unavailable, the application either waits for it to come back on-line, times-out and tries again or the process just fails. With a loosely coupled service, the application would continue to submit database requests into the database queue for processing. Once the database comes back on-line, requests can be processed either by priority, or in the order in which they were submitted.

Therefore loosely coupled services are often required for Web Services due to the uncertainties (e.g. unpredictable load, unknown users, unreliable communications and unreliable access) the Internet may present.

2.2 *Service should Support Dynamic Hardware Scaling*

So far we have limited our Web Service to a single server and assumed that by supporting a messaging-based communications model, the server would be able to address the service's scalability needs. However, how do we address the scenario where a single machine, no matter how the service is architected, doesn't possess the processing power required to handle the Web Service requests? In this scenario, the Web Service is deployed on a distributed multi-server architecture (functions such as the handling of web requests and the execution of application and database logic are distributed to individual servers). However adding additional hardware to solve scalability challenges potentially introduces a new set of development complexity.

Here is an example of the steps a distributed multi-server architected service would take to process multiple requests for inventory availability. 1) The requests are spread across multiple, or a pool of web servers. 2) Individual requests are forwarded to the appropriate application on the appropriate application server. 3) The application submits a query to the appropriate database server. 4) The database executes the query. 5) The result is returned to the application. 6) The application passes the result to the web service. 7) And finally, the web server replies to the requester. The Steps are almost identical with the exception of having multiple server available to process, and share requests.



However, by the virtue of being distributed, developers must solve the following problems.

1. **Dynamic Load Balancing** – since several servers can process requests, developers must determine how to

This service may also include the ability for groups to collaborate and create micro-communities around personal financing, a personalized layout for the analysis, ability to inspect work in progress and status through the web.

3. **Inter-Enterprise Integration Service**

The Web Service is an integration or routing service. The purpose of this type of service is to enable businesses or people to dynamically interact, exchange information, collaborate, and do business over the Internet.

An example of a routing service is the integration between sellers and suppliers. To start, the seller takes an order from a customer. Then the seller wants to send the order to the suppliers to send directly to the customer. The seller doesn't want to handle the communication, routing, and error handling between each of the suppliers. Instead, they send the order to an Integration Service. This service will communicate with each supplier, to track failure to respond, to deliver notifications when problems emerge to the appropriate users, and to do the necessary document translations. This system delivers value by providing custom connectivity, store and forward, administration user interface, custom transformations.

These are just a few of the many potential scenarios businesses will look to enable via Web Services. Therefore the ability to build and integrated these functions, or components, is critical to their success.

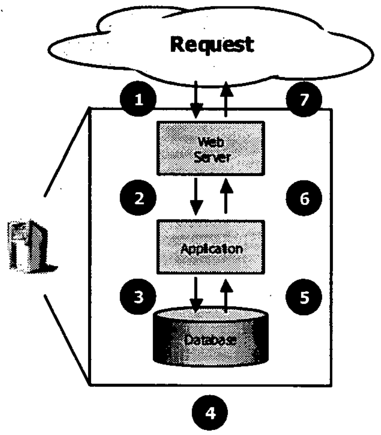
2 Developing a Web Service

Developing a scalable Web Service to address the scenarios noted above requires developing an infrastructure to address a few fundamental challenges related to offering a service. 1) Unpredictable loads, unreliable communications and unreliable access, 2) hardware scaling (i.e. the ability to arbitrarily throw hardware at scalability challenges) and 3) integration (i.e. the ability to interoperate with other systems and services).

2.1 *Services should be Loosely Coupled, or Messaging-based*

Before the web, most communications between applications in the client-server world were synchronous. The client sends a message and then waits for the server to respond. The majority of Web-based applications today begin as client server applications on a single multi-function server (the single server handles multiple functions such as the handling of web requests and executing application and database logic). However, businesses may quickly outgrow this setting and require developers to re-engineer many of these web-based applications.

Here is an example of the steps a client server web service would take to process a single request for an inventory look-up, where the look-up returns a single value indicating availability. 1) The web server receives the request. 2) This request is forwarded to the appropriate application. 3) The application logic submits a database query and sits idle awaiting the response. 4) The local database executes the query. 5) The result is returned to the application. 6) The application passes the result back to the web service, which is freed to process another request. 7) The web server forwards the result to the requester.



In this simplified example, there is a predictable load (i.e. the single request), a simple response (i.e. short processing time) over a reliable communications (intra-computer) infrastructure with reliable access (i.e. high service availability). For this example, a tightly coupled, or synchronous, service is acceptable.

However, by the virtue of being on the Web, the service can be exposed to unpredictable loads (i.e. can't predict the number users) from unknown users (i.e. can't predict the users intent) over an unreliable communications (i.e. inter-computer) infrastructure with unreliable access (i.e. can't predict the availability of other systems or Web Services). On the Web, synchronous applications are often too fragile and inefficient to handle this level of uncertainty and a loosely coupled, or messaging-based infrastructure, architecture is required. The following scenarios highlight the fragility of a synchronous, client server, Web Service.

1. **Unpredictable Load**

Because loads can be unpredictable, systems can become backlogged, or unable to process requests in a timely manner. This may lead to system failure as services time out or become unavailable. Loosely coupled services can increase the number of simultaneous requests a service can support, during the same unexpected surge in Web Service activity, by quickly freeing valuable resources, and thereby enabling the service to accommodate more load.

most efficiently allocate, or spread, the requests across the server resources available.

2. **State Management** – since it's possible for multiple servers, in the same server pool, to handle the same user session, developers must determine how to maintain the state, or session context, as the session moves between servers.

3. **Caching and Cache Coherency** – since communications infrastructures can become saturated, potentially causing performance problems, by processing redundant requests, developers must determine how to cache frequently requested data. Not only must a developer determine an appropriate caching algorithm, they must develop cache coherency, or cache synchronization, algorithms in order to ensure cached data accuracy.

Loosely coupled services along with a distributed, multi-server architecture may provide consumers with a more satisfactory experience as it provides consumers with the perception of a rapid synchronous process. As services can collect the necessary information for a business process, then begin processing once the consumer is off-line.

As developers apply more hardware to solve critical scaling challenges, they potentially increase the complexity by introducing new factors into the architecture. Key factors such as: load distribution, state management and caching must be taken into account.

2.3 Service should Support XML for Integration

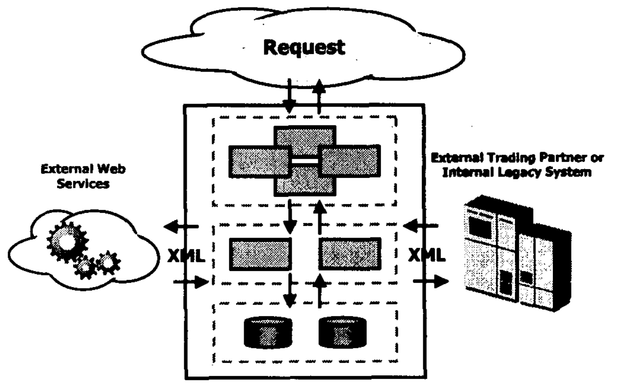
As defined earlier, a Web Service needs to "provide a mechanism for another service or application to use", or a method of integration. So far we have focused on developing a loosely coupled, or messaging-based infrastructure that will enable a service to handle some of the uncertainties of the Internet as well as providing hardware scalability.

The message format is at least as important as the communications infrastructure. For example, if a supplier is able to successfully transmit a document to a buyer, but the document is in an unrecognizable format, the buyer may not be able to consume the information in the document. Therefore, without a standard syntax, a lingua franca, for messages between Web Services, they will not be able to efficiently and effectively integrate to create powerful new web-based business solutions.

XML is such a standard. Defined by the World Wide Web Consortium (W3C, see www.w3c.org), XML provides a self-describing, structured representation of data that can be implemented broadly. The advantages of XML are: 1) XML is self-describing, 2) XML is platform independent, 3) XML is a general-purpose description language that can encode many types of information such as data, programs and types, and 4) XML is independent of applications or vendors, it provides an open mechanism for businesses to define and express business documents and services.

While XML itself is straight forward, some of the accompanying open and standard technologies make XML a flexible open standard for message format. Some of these technologies include:

- **Document Object Model** – Called the DOM, the DOM enables programmatic access to the XML document by exposing a logical document structure. Via this logical structure, programmers are able to extract necessary data as input into other processes or applications.



- **XML Schemas** – XML Schemas enable developers to declare the structure (e.g. document structure, valid elements and element attributes) of an XML document. Via XML Schemas, developers can validate entire XML documents or elements of the XML document. Since the XML Schema is defined in XML, trading partners for example can agree on a document format.

- **Extensible Style Language for Transformations (XSLT)** – XSLT enables developers to extract and transform information XML Documents. This transformation could be into another XML format as well as into other formats such as HTML, or traditional B2B document formats such as EDI.

Utilizing the open XML standards for Web Service integration, Web Services, and Web Service developers, are able to exchange documents and document schemas in an open, standards based format. Because it's an open industry standard, developers can utilize a myriad of open tools and technologies (e.g. XML DOM and XSLT) to manipulate and integrate dynamic data from various sources.

2.4 *The Complexity and Cost of Developing a Web Service*

So far we have identified a few of the technologies that are required to build a Web Service infrastructure. We have, for the most part, ignored the complexities and costs associated with this infrastructure and ultimately have ignored their impact on a businesses ability to meet time to market objectives.

Developers, or teams of developers, must grapple with the complexities of architecting a messaging-based infrastructure to maximize resource utilization and minimize the traditional Internet disruptions such as unreliable communications, unreliable access and unpredictable loads. The complexity becomes exponentially greater as the load increases and the number of servers increases because developers must now deal with sophisticated load balancing, state management, caching and cache coherency algorithms. Developers must also embrace XML and the complexity associated with creating schemas and XSL Transforms as well as the XML DOM. Each of these areas requires it's own programming expertise, and teams of developers must work together to ensure success.

To the complexity associated with developing the infrastructure, and the skills required, the challenge of building a scalable data center, or the hardware to deploy your infrastructure, must be factored. Scalable data centers are not an after-thought, they must be architected for scale from the start. A typical Web Service data center requires expertise in data center planning, system administration, database administration, router administration, systems management and hardware systems configuration. As with developers, data center teams must work together, and with developers, to ensure the success of the Web Service.

So far we have only dealt with some of the basic requirements for a building a Web Service infrastructure, and we have assumed that there are developers and data center resources available to address these challenges. This assumption couldn't be more false, as businesses all over are struggling to fill even the most basic of programming skills needs. Unfortunately for businesses, the skilled programming resources required to build the scalable Web Service infrastructure are even scarcer. This leaves many businesses aggressively bidding for these rare programming resources and leaving many businesses, especially those without significant financial resources, unable to meet their development needs.

Software development and data center operations costs are just the beginning. Businesses must invest heavily, and continually re-invest, in the underlying hardware and software required for the infrastructure. Not only are there these basic licenses, but also in order to meet service goals, or service availability, businesses must invest in redundancy, or multiple systems that can act as backup to each other. This includes systems such as expensive intelligent routers, communications infrastructure, Web Servers, Large Application and Database Servers. And finally, businesses must also license the operating software required to power the hardware.

A key attribute of a Web-based service must include the ability to evolve and adapt to ever changing technologies and market conditions. This ability is vital in ensuring businesses can address evolving market conditions or an aggressive competitor. Unfortunately, once we have paid for all the expertise, hardware and software we have to realize a certain amount of time is required to put it all together. Businesses must also factor that innovation from traditional technology providers may be slow potentially impact the business needs to remain competitive.

For example, a typical software delivery model requires 18-24 months for a major system release, 3-6 months for a Service Pack release, not to mention lengthy testing cycles, time lost re-architecting systems for a new feature(s) and the potential for service interruption during system rollouts.

When a business considers the infrastructure development, hardware and software licensing and data center operational costs and time, they often realize that a significantly large portion of their resources is dedicated to non-revenue bearing investments. This often unknowingly transforms businesses with a vision and expertise for a simple Web Service, such as a tax calculation Web Service, into business focused on building technology.

2.5 *In Summary*

In order to build a scalable Web Service, or a Web Service that can scale to handle unpredictable loads over unreliable communications with unreliable access, which can interoperate with other Web Services and systems, the service should support: 1) Loosely Coupled, Messaging-based Communications, 2) Dynamic Hardware Scaling, and 3) XML.

In order for businesses to achieve their Web Services expectations, both internally and externally, the service must be architected from the ground up to support these. However, when a business considers the complexity, costs and time required to develop a scalable Web Service (e.g. infrastructure development, hardware and software licensing and data center operational costs), they often realize that a significantly large portion of their resources are dedicated to infrastructure, and away from the resources required to building the application. Therefore businesses must weigh: the complexity, the costs and the time to market, against the value of building, deploying and managing a Web Service.