Making Web Services that Work

Steve Loughran
HP Laboratories,
1000 NE Circle Boulevard
Corvallis, Oregon, 97330, USA

slo@hplb.hpl.hp.com

September 30, 2002

Abstract

This paper explores the techniques and technology needed to make building production-quality web services a manageable problem.

It approaches this problem in both software process and in the specifics of some areas of a Web Service —such as API, security and session management. The software process is more foundational: this paper argues that the traditional waterfall model of deployment is not appropriate for Web Services. In these services, integration with external callers is one of the key problems, yet this only surfaces on live systems. Debugging on, and rapid updates of, the live system are the only way to address interoperability and other problems that surface on the production service. Furthermore, as management of a running system is the main ongoing cost of a service, the needs of operations cannot be left as an afterthought to the system design.

We propose the incorporation of the deployment process into the iterative development process through the inclusion of operations use cases, tests and defect tracking into the normal software cycle. When combined with a fully automated deployment mechanism, the foundations for a rapid web service development process are in place.

On the technical aspects of Web Service development, there are core issues related to security, interoperability, scalability and robust design that need to be considered from the outset. We explore these topics using the experience gained from encountering many of the issues when developing production systems. We then go on to explore how the component models of the future could adapt to better suit server-side and Web Service development. We argue that uniform APIs for standard services —logging, configuration, management, testing— permit reusable components to be used in a production server environment. We discuss how neither of the two platforms competing most vociferously to be the foundation upon which developers should build their services, Enterprise Java and Microsoft .NET, have room for improvement in this area, and propose some strategies for improvement.

Finally, this paper explores the concept that individual Web Services will become independent components within the emerging Web Service Federation; the collection of independent systems interworking using common Web Service protocols. As these protocols evolve they may become viable to connect programs and computers inside an individual Web Service, although this is only partially viable today.

Contents

Chapter 1

Introduction

This paper explores issues that arise when developing production quality Web Services, and techniques to address them. Before delving into the topic, here is a quote from a telecomms software developer on the process he uses to deliver quality products:

> "I don't think there is any secret to high availability systems; you just make sure you don't leak memory, stop the users from changing the configuration, have detailed trace information which can be enabled dynamically, then test for six months before you ship"
>
> Mark Syrett, developer of telecommunications server software [Syr01]

Web Services take on high availability requirements with delivery timescales that never include the luxury of a six month test period. They add the problem of integrating with otherWeb Services, none of which were developed to such a rigorous standard as the telecomms systems. They encounter problems that few prior systems have encountered, yet are being written by more developers than before, all of whom have been convinced by vendors that web services are easy to write. The hard part is not writing the service, it is getting a working, deployed implementation to on the date committed to, and keeping that working. Those aspects of the problem have been neglected to date, because everybody is still discovering the issues.

This document combines the lessons we learned building and deploying production services with opinions upon how Web Services could be done better in future. The improvements are both technical and procedural; the proposed changes to process are more fundamental than the technical issues, and perhaps

more controversial. We believe that both are essential.

1.1 What is a Web Service?

At its most abstract, a Web Service provides functionality to remote applications and web sites, by exchanging XML and binary data. This can be provided via a standardized XML marshalling protocol such as XMLRPC [Win00] and SOAP [W3C01, W3C02a]. An alternate approach is to by use the REST conceptual model [Fie00] to offer a service that responds to GET, POST and other HTTP protocol methods against different URLs representing objects and their attributes.

Despite the differences in how requests and responses are sent from client to server, all share the core notion of exporting computation services from one system to callers across the Internet, using HTTP as the underlying substrate [FGM+]. They also share a lot of the underlying implementation details: they are likely to be run on an application server of some sort, managed by an operations team, and maintained by a development team. This means that many operational and process aspects of Web Service developent are independent of the actual Web Service architecture.

1.2 Why are Web Services Hard?

Software engineers have been designing and deploying distributed systems for years. Why should Web Services be any harder?

Complex, large-scale distributed systems have always been hard to write. Technologies such as COM and CORBA have evolved to make writing such systems possible, and yet they have not made it easy. Small LAN-scale distributed systems using these technologies are tractable; a single distributed system can be built, deployed, and upgraded as one unit. Treating the distributed system as a single unit eliminates version problems, while local area networks have the bandwidth to permit 'chatty' object reference management mechanisms and the speed to hide the fact that object method calls are being made against remote systems.

Attempting to scale these technologies to the Internet introduces problems. The long latencies and lower bandwidth of the connections forces designers to rethink the granularity of communication, and how that communication is implemented. Method calls to remote objects cannot be transparent when the remote objects

are a few thousand miles away, connected over an unreliable link.

Even if these problems were taken into account in the design of a Web Service, the business model of Web Services: exported functionality for third parties, makes it dangerous to try and re-implement the tightly-coupled architecture of small-scale distributed systems. A Web Service will be expected to implement a stable interface for as the lifespan of the client application, or at least for the lifespan of the Service Level Agreement (SLA) between the client application and the provider of the Web Service.

Technically, that pushes developers towards a loosely-coupled, asynchronous communications mechanism, one in which the caller has to clearly state its version expectations of the remote service. It may be that message based communications is the correct network architecture to compensate for reliability and latency issues [Mic02a]. This still leaves versioning as a long term problem that Web Service developers have to address. Historically, the software development industry has been very bad at versioning, with COM and DLLs the reference example of how not to do it. As Parnas explained so long ago, the exported interface of a module comprises its signature and its behaviour [Par74]. Any mechanism which merely validates that the remote program still has the same signature as before is insufficient. Unfortunately, current remote invocation technologies are restricted to verifying signatures, as without design-by-contract, or other machine readable specification of behaviour, there is no way for the underlying system to compare the expectations of the client with what is currently offered by the server.

Internet-scale distributed applications have to be completely independent of the implementation platform. This need arises because it is impossible to upgrade all clients and servers simultaneously, and over time one will need to add new components to the system, components written in new languages and running on new platforms. If the protocol is platform specific, such as with DCOM, or framework specific, such as Java's RMI or the `tcp:` protocol in .NET remoting, then one is stuck with the original platform for the lifespan of the entire distributed system. Clearly designing for implementation independence is of strategic importance, not just over the Internet, but for any complex distributed system with the potential to last. The use of XML in Web Services promises this independence, yet introduces the whole issue of interoperability.

1.3 The Operational Aspects of Web Services

Regardless of what the Web Service does, there are some implicit requirements from the customer perspective:

- It must always be available.
- It must support the load placed on it by all callers.
- It must be consistent, that is, provide the same interface and explicit and implicit behaviour over an extended period.
- It must be robust.
- It must be global.
- User data will be kept secure and private.

Developers calling the web services also need an implementation of the service as early as possible, so they can start working with it. They also expect stability of the service and the exported API from the moment the first version becomes available. They will also assume that rapid deployment enables instant bug fixes: this creates the risk that deploying a prototype too early can lead to a barely controlled firefighting exercise related to the first prototype, rather than controlled enhancement of the quality and functionality of the codebase.

The operations perspective comes from the need to satisfy these requirements as easily as possible:

- It must be easy to bring up.
- It must have a very low operational cost.
- It must work for an extended period without human intervention.
- It must be secure.
- It must be instrumented for auditing and billing.
- It must be possible to make a live backup of the system.
- It must be possible to make a hot update of a running service.
- It must be robust against transient failures of the surrounding infrastructure.

From an operations perspective, any service that meets the customer requirements, yet is near-impossible to install, requires continual nurturing and daily reboots is a failure. The system simply won't scale up because of the time and effort that would require. Moreover, because of the ongoing support costs it will not be economic to run at any scale.

From a developer perspective, if the service does not work then operations will be in contact with you to find out why. If it goes wrong at 5 am, then you get a phone call at 5:15 am, regardless of whether you are officially "on call" or not. To be able to sleep soundly at night, your system must not only work well, but when things start to fail for any reason, the processes must be in place for operations to diagnose and fix the problem without having to involve you.

Management have a different set of perceptions. They are used to the notion that a web site can be revised in a minute, and infer from this fact that updating a human readable web site is trivial, that updating a machine readable Web Service is too. Certainly, automated deployment can take a matter of minutes, but that does not eliminate the need for applying all the rigors of a normal defect fixing process: tracking, replication, fix proposal, fix implementation, testing, regression testing and then finally deployment. This leads to an interesting difference of expectations that developers need to manage.

The designers and developers need to be aware of these needs and issues from the outset, so that their code and processes work with the Web Service model. If they don't, the service they will deliver will require extra support —including developer support, and the entire development process will be painful. The Web Service model is meant to make development and deployment easier, after all.

Chapter 2

A Deployment-Centric Development Process

One conclusion from our experiences of building and deploying Web Services work is that we need to move to a deployment centric development process; both popular processes, XP and the RUP are not complete when it comes to addressing the needs of Web Services and need tuning to work.

The key observation from our projects was that the process of "handing off" a web service to an operations group to run was essentially reverting to a waterfall model of development. The waterfall model is so discredited precisely because of its inflexibility: evolutionary development is viewed as essential for feeding back the experiences of the later stages of software development back into the earlier stages. With a web service, most of the issues related to installing, configuring running the service are only encountered once deployment begins. Without a feedback loop from from the operational system to the development team, the developers will never adapt their service to meet the needs of operations.

This is a problem which exists on any server system which is handed off to an operations group; modern Web sites would seem to fall into this category. However, most web sites are primarily standalone systems —Web Services add a new problem: integration. It may be possible to deliver a web site by running it in an isolated staging system for a test period, but Web Services only begin to encounter interoperability issues with third parties after deployment. This forces the developers and operations to work together, enabling developers to debug interoperability problems on the live systems.

The Rational Unified Process focuses on building code from a domain model

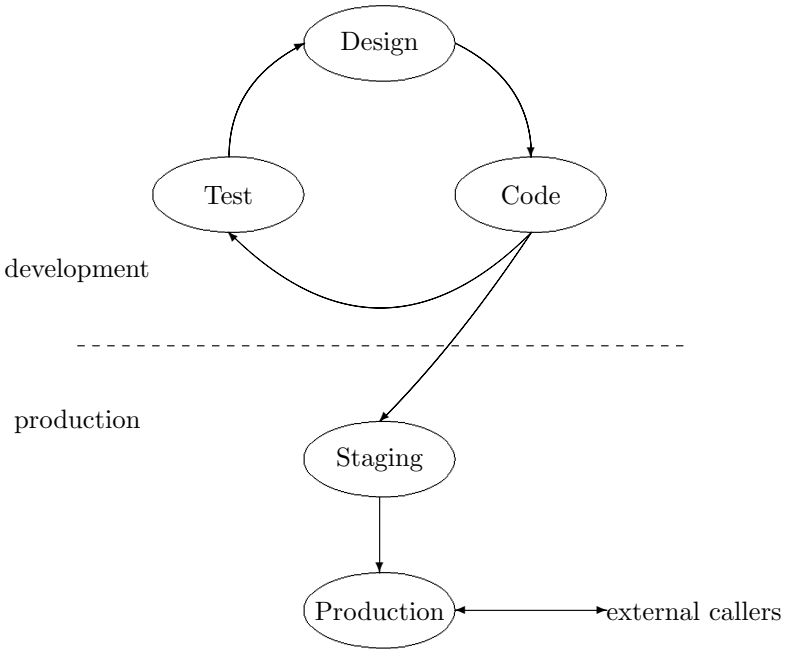

Figure 2.1: How iterative development processes revert to a waterfall when the service is handed off from R&D to production.

derived from use cases [Kru00]. This process is incomplete for Web Services, as it implicitly ends by handing the software off to production. Iterative design and development may be used but the waterfall model is returned to when it actually comes time to deliver the product.

Figure 2.1 illustrates the problem. Even an iterative build-test-redesign cycle hands off staging and deployment to operations.

The Unified Process can address the up-front design issues, it just needs to take on board the deployment problem earlier on, preferably during the analysis phase. Deployment must actually begin during the implementation phase, and the results of that fed back in to the ongoing development of the software.

The XP process model focuses more on being flexible with respect to change, by not over-designing up front and instead keeping the implementation clean as developers extend, refactor and rewrite classes. Testing, including comprehensive unit and regression testing, is central to the XP philosophy, ensuring that old code always works and something is always ready to deliver.

The XP model is differently weak, because it is not robust against fundamental design errors, especially against those made in the public interface of the service or the foundational deployment model. Some errors can only be fixed by throwing everything away and starting again with a different implementation: the purpose of upfront design is to avoid that situation and the resulting costs. Furthermore, because Web Service interfaces have to be robust, it is impossible to change the interface and semantics of a published version of a service; design errors in the system get frozen, whether or not it is appropriate.

2.1 Introducing Continuous Deployment

Martin Fowler has introduced the notion of *Continuous Integration*, in which an automated tool continually rebuilds the source from SCM, runs regression test and notifies the team whenever the tests fail

[FF00]. This process ensures that nobody ever forgets to run tests, and that the code in SCM will always build and run. Sam Ruby has taken this to its extreme with *The Gump*, [Rub01], which is a six-hourly rebuild of all the popular open source Java projects hosted on Apache and elsewhere.

We are exploring extending this model to one we term *Continuous Deployment* —after passing the local test suite, a service can be automatically deployed to a public staging server for stress and acceptance testing by physically remote calling parties. One can implement this by automating the deployment tasks in an Ant build file and using the current generation of Continuous Integration tools [HL03, Ant02, urb01]. This certainly complicates the build process, and encounters resistance from operations, but promises to make it much easier to develop and deploy Web Services.

Figure 2.2 illustrates the difference. With deployment to staging brought into the core process, one can test against the external staging site; both internal functional tests and external integration tests. The final production site cannot be brought into the process so tightly, at least, not when it is in active use. Even the production system will still effectively be a test system to external callers, as they integrate their Web Service clients with the system. As a consequence developers will still need debug and diagnostic access to the system —the notion that after passing sucessfully through staging a service is perfectly functional is as valid as the idea that client software is bug-free at the time of its product launch. While granting developers access to the production system may seem an admission of fallibility, it is an effective way of taking advantage of the replicability of Web Service defects: with only one installation, all problems should be replicable.

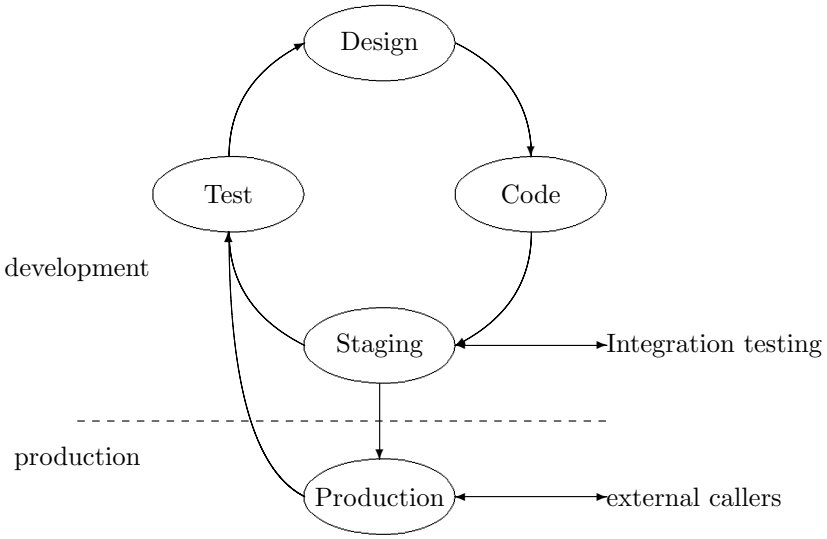

Figure 2.2: Continuous deployment integrates deployment into the core cycle

2.2 Operations Use Cases

Just as *Use Cases* are the basis for the RP process, *Operations Use Cases* help direct the development of server-side software to meet their needs, and so are critical to a deployment-centric process.

Operations will have a large number of tasks they need to perform on a system. Many will be uniform across all systems: "apply security patch to IIS", "back up complete system image" and "defragment hard disk". Others will be unique to an application: "install new fonts", "verify file quotas", and "update postcode data". If operations need to do it, developers need to be aware of the need early on, and have some means of supporting them. It may be code; it may be process; it may be a mixture. The more automated the use case support is, the more likely it is the process will be carried out consistently.

Here are some use cases we have found useful:

- Determine which version of all components are running on a remote system
- Hot update of running service
- Immediate rollback of service to previous version
- Apply security patch to OS/app server/web server.
- Copy entire service configuration to a cloned system
- Housekeeping of disk space: clean up files and move directories.
- Hot reconfigure of running service

One category of deployment use cases are the *when things go wrong* events:

- Diagnose and correct network trouble.
- Diagnose and work around intermittent failure of dependent component.
- Investigate the suspected compromise of a server.
- Reboot entire system after complete power outage.

The final item may seem out of place, but it is a perfect example of why operations use cases are so important. Consider what the boot sequence of the entire stack would be. Does the Domain Controller or DNS server need to be available

before the rest? What about the database server? What if they take longer to check their file systems than the other boxes? It is easy to design a system where operations can bring up each server individually, but an unplanned simultaneous system reboot does not come up by itself.

Engineering may have their own use cases involving the deployed server. These could also be categorized as Operations Use Cases, as they do require the active involvement of the operations team to be viable:

- Debug interoperablity problems with a client by running a debugger against the server
- Debug security and authentication problems.
- Post mortem a crash dump.

The classic deployment model of staging and production does not normally permit developers to debug the production system. However the system integration nature of Web Service development effectively requires that developers have some means of debugging integration problems on a live system. The organisation and debugging tools need to be able to permit this.

2.3 Develop Operations Test Cases

Just as one can have use cases for operations, it is possible to have automated tests that verify that the system is "happy". This is separate from functional tests, that verify the system works, as these tests are lower level and can be used for diagnostics. They can verify the components and configuration the service depends upon are correct.

How does one know what tests to write? Firstly, after making a list of basic requirements and operations use cases, a first set of tests should be obvious. Secondly, after something goes wrong —such as the filestore being eight hours behind the rest of the cluster and housekeeping deleting files as they are created, or a recurrent installation issue with a COM object, developers should write a unit test to verify the problem before fixing it. Verify the test fails, fix the problem then verify the test passes after the next deployment.

This is the same process as for code development; we are just using it in a different context. Here are some of the tests we have written:

1. A "temp" directory exists and is writeable.

2. The cluster-shared directory exists and is writeable.
3. The timestamp of files created in the temp and shared directories matches the local clock to a few minutes.
4. The configuration lists the URLs of two servers we depend on.
5. Certain remote URLs must be accessible by the service.
6. The XML parser must be able to parse certain XML files.
7. An external executable must exist on the path.
8. A COM object the application depends upon must be creatable.

What can these tests be used for? Firstly, they can be used in any health test pages or management calls. Secondly, developers can write a unit test wrapper class which performs the same tests, and the server side Ant deployment file can run those tests before the system goes live.

Example: We wrote a health page for Apache Axis [Axi01], `happyaxis.jsp` that probes the system for classes being available, to address installation problems that many users encounter. This test page is automatically probed in the automated post-deployment httpunit test suite, and it is also made available for end users. The test page therefore benefits an continuous deployment process, with its automated tests, and anyone trying to diagnose a problem on a new installation.

It is impossible to write simple test automated cases for all deployment problems —such as pulling out network connections, turning off a server with no warning, or forced deletion and recovery of all backed up data. These will need to be generated by hand, and with care.

2.4 Have Defect Tracking for Deployment Issues

Configuration and deployment problems need to be logged in the defect tracker, just as with code defects. In particular, any event that returns error messages to customers or in the log should be marked as a defect with causes and fixes added, so that in future the database can be searched when the same behaviour re-occurs.

2.5 Deploy Early, Deploy Often

In iterative development, one cannot wait till the end of the project to deploy; deployment must be a part of every iteration, and the lessons learned in deployment and operations the system fed back in to the design. Viewing operations and deployment as high-risk activities enables this, as it helps management recognize that the activities need to be addressed up front.

This process introduces the risk of diversion into firefighting, as developers get pushed into to continually fixing and redeploying the previous iteration instead of working on the next version. Defects should be filed and fixed. A rigorous update policy of regular daily updates to the visible services —and no more than daily, sets expectations clear from the outset.

In the early stages of a Web Service project, most support calls are not operations related so much as defect related, as the service is still stabilizing. In this period, developers may be needed more often than operations.

Management should clearly specify a service level that explicitly excludes no evening or weekend support. If the engineers are working at those times, great, let them work without interruptions. If they aren't, let them take a rest. To put them on call requires a proper mechanism for balancing the load among staff, and some form of compensation to those on call. Given the effect that such a rôle can have on normal productivity, offering a downgraded availability, such as six hours a day, five days a week is much more preferable.

2.6 Use the Target Application Server from the Beginning

Start working on a multi-processor system with the same application server as the production system. Ideally, this should be a remote box, deployed with the same automated process as the production cluster. That includes SSH based deployment, router configuration for security and round robin load balancing, everything the production site will have. Waiting for staging to offer these facilities is hiding problems

It is imperative to never to switch from one J2EE server implementation to another midway through a project. That is, not without expecting to spend a significant amount of time learning the differences and getting the application to work again. We have found every such server to have its own quirks and epiphenomena; behaviours which aren't really designed in but arise from how

the product is implemented. They also have their own interpretations of the J2EE specification [1].

Developers for the .NET platform would seem to be exempt from this problem, but the arrival of .NET server and version 1.1 of the framework introduces the problem, albeit on a lesser scale.

2.7 Automate Deployment

Ant can deploy using FTP, Telnet and HTTP, and there are custom deployment tasks for many application servers [Ant02]. SSH will tunnel these protocols over to a secured remote site. There is no technical reason not to use the same deployment target to deploy to both local staging and remote production web sites in a fully automated manner. This is not only a prerequisite to *Continuous Deployment*; it reduces the chance of installation errors when updating a cluster of systems.

It is valuable to include version files containing a a build number and timestamp; serve these up and the Ant deployment target can finish with a `<get>` to verify the deployment just went through; unreliable deployment can be a real time-waster if it goes undetected.

2.8 Treat the Router as an Integral Part of a Cluster

Any complex Web Service will have something at the front balancing requests, ideally something like a Cisco/ArrowPoint Level 7 router, routers that understand URLs, can probe servers for happiness and take them out of rotation when they are unhappy, and generally act as a filtering and load balancing front end to the cluster. If the service is going to have one of these, start using it from the beginning. Design a low-load but informative status page for the servers, and test the router with the appropriate HTTP/HTTPS protocol. Tests need to verify that the service continues to work when one of the cluster elements is off-line, and that the load balancing does not interfere with any session state stored in the cluster nodes.

[1] This exposes the flaw with test-centric processes such as XP; you always have less tests than you need, and certainly the J2EE compliance test suite must be vast. A design-by-contract programming language and a formal specification may be superior for a specification that is implemented by multiple independent parties.

The router/load balancer must be aware of the state of the system at a higher level than simply 'servicing requests'; it needs to recognize when a node is not successfully executing requests and so needs to be taken out of service. This requires a status page and a load balancer that probes the status page on a regular basis. Deployment test cases make ideal candidate tests for the status page. If these take a long time to execute, they do not need to run every time the load balancer hits the status page; they can run once a minute and cache the result: [Bul00] shows how to do this.

Example: At one point we had a load balancer that would route requests to the application server with the shortest queue, but it lacked any mechanism to probe the server instances for health. When a service instance had a problem that would allow it to receive requests but be unable to service them, it would respond to incoming requests with a SOAP fault. The load balancer noticed that this instance had a shorter queue because it was servicing requests faster, and so routed more requests to the broken system than the healthy ones.

The lesson from our experience was that the load balancer needs to be intimately aware of the health of the services and what the expected behaviour of instances should be. Perhaps the services should recognise themselves when they have problems and take themselves out of circulation, somehow, rather than rely on an external agent.

Write management routines from the outset

With operations use cases written up, it should be easy to start adding management code from the outset. This includes passive routines: logging code, instrumentation for load reporting and other performance statistics, and active management code. The latter are the pages, web service endpoints and classes added for service management.

A good stage in development comes when you start writing JUnit tests instead of `main()` entry points; an even better level of sophistication comes when you write JMX MBean entry points to which the JUnit tests can refer. Unfortunately, the singleton nature of the MBean factory does not mesh well with JUnit. Many MBean based tests can avoid registering/unregistering themselves in the factory, so encounter no test problems.

Note that XDoclet [XDo01] radically simplifies the JMX development process by autogenerating the MBean interface from the implementation and extra Javadoc tags.

Chapter 3

Addressing Scalability

Web Services need to scale. They need to scale up if they become popular; they need to scale down to reduce operational costs by running many instances simultaneously. There are different ways to achieve this. The exact approach taken may depend on the lifecycle stage of the product. If there is a limit how it scales, then the maximum size of the service is limited. If it doesn't scale down, then the service cannot survive any lean times. Getting the scalability right is clearly important.

3.1 System Configurations

The underlying hardware configuration of a service has a significant effect on the system. Many different configurations used to address different aspects of scalability and availability. Development machines are usually much simpler, which is dangerous. It is critical for the core test systems to resemble the actual deployment hardware; otherwise, the code will not have been tested properly.

Standalone Uniprocessor. This is the simplest of server configurations. It lacks high availability, but may still need to be managed remotely.

Standalone Multiprocessor. This is a minor extension of the uniprocessor system, but one that can break very badly designed software. Mediocre software may still work, yet not exhibit any speedup. It is a good for a local development

system to be multiprocessor, as it catches trouble early. Load tests seem to be best for uncovering race conditions, although such bugs are always hard to track down.

Interconnected Stack. In the stack design, a number of server and services are interdependent, each providing a piece of the functionality of the whole. A failure of one of the elements of the stack can result in degraded functionality, or it can cause the entire stack to fail. A stack is very common in a three-tier or four-tier system, with a database and file server supporting the front end application servers. A stack can be hard to set up, as it needs cross-system configuration, and its cost keeps the number available in a project to a minimum. This is unfortunate, as one stack per developer would be better for development productivity, even if the operations overhead would be high. Unless of the service is designed for easy configuration and low operational cost.

Distributed Stack. Here the machines comprising a 'system' are remote, each only connected over long haul links. Shared filestores are not available; the probability of RPC failure is high. For example, even short HTTP requests may be interrupted, so checking of content-length headers and perhaps even checksums are needed to verify the completeness of downloads —even between systems in the same workgroup. Testing systems for performance over slow and unreliable links is much harder than basic functionality testing.

Load Balanced Farm. This is a farm of servers with a load balancer above the farm balancing requests. How does the load balancer work? Either randomly, round robin, or with some awareness of the actual load on the machines. The first two work best if all machines are symmetric; the latter needs more computation but helps balance load over a farm of systems with different performance, or in situations where the load requirements of a request vary significantly. Effective load balancing of Web Services is a fruitful area for further development.

Failover Farm. A farm of servers which not only has load balancing, it is somehow designed to be robust against failures. Any unsuccessful request can be automatically re-issued till it succeeds, such that the client does not need to implement retry behaviour itself. Realistically, the clients still to implement timeouts and retries, because requests can fail before they even reach the web farm.

Grid. Grid computing is an interesting model for implementing a scalable web service. If the service runs on a grid infrastructure, then it could "rent" a fraction of the available computing power. Scalability would come by using more of the available computing and storage services, and from adding more systems to the grid. If the services can run across many systems, yet only use a fraction of their CPU time, then the service can have the redundancy that a high-availability system needs without incurring the up front costs needed to build that infrastructure from scratch.

If the programming model for writing Grid based Web Services is simple enough to use, and the deployment model workable, then the Grid could be the ideal foundation for Web Services, combining low-entry costs with near-unlimited scalability.

Federation. A federation is a collection of systems that work together, systems operated by different entities, perhaps with different implementations. To interoperate the systems in the federation must implement compatible network protocols, and agree to communicate with each other by some set of social conventions, from formal SLAs to informal agreements.

Few would set out to implement a single Web Service as a federation. However, the total set of Web Services slowly emerging on the Internet will form a federation: the federation of systems that agree to interoperate using the XML-based Web Service protocol stack.

Any Web Service that depends upon external Web Services is implicitly part of this federation; its successful operation depends upon the availability of those external services. Any client application that uses a Web Service is also dependent upon the availability of the Federation.

If the Web Service vision succeeds, the duration of the overall federation will be longer than any other distributed computing technology to date. Existing federated systems, from the ATM networks, and the VISA authentication network to the Airline booking and billing systems, the international financial trading networks, and even the vast SS7 system that coordinates the planet's telephone calls —all may one day be subsumed into the Web Service Federation.

Unlike the existing systems, the Web Service Federation will be heterogenous, and designed to work over a potentially insecure Internet from the outset. This is a weaker foundation than the previous federations, and it will take time for the federation to stabilize. Its reliance on the public Internet as its communications infrastructure will also place some fundamental limits as to its availability; this is one of the problems that dependent applications will have to cope with.

3.2 Software Solutions

Is it possible to design one system to work in all these configurations? Perhaps, if written wisely. The same code designed for a multiprocessor failover web farm should be able to scale down to a single system: but the reverse is unlikely to be true. Grid computing is probably the most unusual platform to develop for, and it is still an evolving technology. Short term, targeting the load-balanced or failover farm at the high end, a multihomed system at the low end is going to be sufficient. For any research oriented project, the Grid and the Federated models would be more interesting.

Multiprocessing

To use a multiprocessor the system needs to have enough threads to to useful work, but not too many busy at the same time. Ideally, there should be one working thread per CPU. Housekeeping threads can be given a lower priority than request handling, but then you need prevent starvation: consider a high priority but rarely scheduled thread that can bump up housekeeping's priority if it has not run for a long time.

Design for basic multithreading through:

- Using immutable objects for thread safety, this can also benefit security. Bloch covers this in [Blo01]
- Documenting synchronization rules.
- Thinking about the threading and re-entrancy issues of all parts of the system.

Design for high performance multithreading by minimizing contention. Instead of using synchronized methods, synchronize on individual variables, array entries and the like. There are some exceptions to this.

1. Split code into the parts that need synchronization with the parts that do not. For example, iterating through a synchronized list of files, deleting some of them could be split into two stages —a synchronized selection of files to delete, and an unsynchronized deletion phase.
2. If all threads access a block of variables in a group, have one synchronization lock for the group as it reduces the lock acquisition and release overhead.

3. Developers may be able to get away with not synchronizing reads to an integer, as in Java, Win32 and C# these reads are atomic. If this is done, it is critical to use the keyword `volatile` to indicate that accesses should not be cached or re-ordered. This trick is dangerous as it will breaks if the date is ever changed to a type whose accesses are not atomic, such as a `long` integer, or if someone uses non-atomic operators such as `++`, `+=`, `--`, or `-=`.

4. In both Java and C# the overhead of calling a synchronized method is slightly less than when the synchronization code is implemented inside the method itself.

Object pooling is often touted as a performance trick to minimize object creation overheads, but on a multithread system, one can sometimes get away with having a thread local instance of core objects that can be re-used every request. For example, a per-thread 128 KB buffer can be used to for buffering reads and writes on blocking IO, rather than having a pool of buffers that threads have to acquire and release.

Test, test, test on MP systems, as this is the key way to be sure that race conditions really don't exist. Don't wait till deployment time before discovering that you have thread problems. The other way to do is using formal mathematics [Mil80], but this is very hard for anything other than a trivial example. We have found that realistic and long lasting load tests, accurate even to the simulation of client delays, capable of discovering many race conditions in a Web Service, from thread concurrency issues to race conditions between worker modules and housekeeping routines.

Bulka, [Bul00], has good coverage on performance tuning server-side code. Tricks covered include caching objects for a finite period of time, such as caching a `java.util.Date` object for a second, rather than recreating a new one every request. Tricks like this can impact the readability and maintainability of code, but can make a measurable difference.

3.3 Clustering

Clustering is the sequel to multithreading; a cluster of computers acting like one service. Performance of a cluster should scale with the number of boxes. If the system is badly designed, management scales up significantly faster than performance.

- Design the system with no race conditions with other cluster elements

accessing shared data. Databases and directory servers should be the standard repositories.

- Design the service such that state is not kept in a single application instance, for failover and load balancing. If all the requests in a session must be routed to the same box, the service cannot handle failover.
- On a cluster, simple web status pages does not work as a management mechanism; there are too many systems to track this way. Use of application management technologies —JMX, WMI, SNMP— from the outset can avoid this issue.

Although the software must keep session state in a shared location for failover robustness, if the router consistently redirects session requests to the same server, that server can cache content from call to call. Because it should check to see if the persistent data has changed, caching the persistent data is worthless —but caching incidental data is. There is always a trade-off between the costs of data creation versus those of storage; these costs (and the frequency of the need to recreate objects) vary between projects such that there is no clear answer as to best approach.

3.4 Partitioning

This is an extension of clustering. Sometimes there is a hard limit on how many boxes can be in a cluster before contention becomes an issue; all the work put in for load balancing and statelessness imposes overhead which only new independent clusters can avoid.

If clients can be assigned to a single cluster, either statically or dynamically, then the logon process should bind the client to the appropriate cluster for a session: the logon server should be a cluster apart from the rest.

- To enable partitioning, always use GUID identifiers for externally referenced objects, not incrementing integer columns supplied by the database.
- For effective distributed deployment, never rely on a globally accessible shared file store as part of the solution. It does not work over long haul links, Use HTTP URLs that can easily be mapped to files in a shared filestore when the URLs are resolved on a system close to the files.
- A logon process where the caller is allocated a session ID should also return the endpoint for the rest of the conversation.

- If the partitioned systems have to interoperate, and they will be distributed widely, such as co-located on MAE-West, MAE-East, London, and Singapore, consider using loosely coupled Web Service protocols for the communications. Toolkit support for asynchronous messaging and connectivity failures are relevant for such distances. There are even implicit benefits if the protocols do not require all instances to be running the same software version; it becomes possible to perform a rolling overnight update in each timezone.

3.5 Multihoming

This is scaling down: running multiple instances of a service on a single system or cluster. This is a useful configuration to support both at the beginning or end of a product's life, when it is being ramped up or run down. Multihoming provides a different endpoint for each customer using DNS as the routing mechanism; this gives operations the flexibility to redirect customers to different server clusters.

To support multihoming the service must be able to use the logon or request URL to generate an appropriate personality for the caller, accessing the appropriate data sources. This can be done with multiple instances of the same webapp running, or the multihoming can be done on a session by session basis; the latter being lighter weight but more complex, and potentially less secure.

The hard part is for the server to recognize what identity it is running under at any point; this can be derived from the URL of the request. If that is done then the session information —be it in cookie or server side— must track the original 'server' identity. Otherwise, a user could authenticate in one server and rewrite the URL to acquire rights in a different virtual server.

A barrier to effective multihoming can often be the underlying application server and Web Services software. Check the capabilities of these products early, if multihoming is considered important.

Chapter 4

Design Issues

4.1 API Design

What is a good design of a remote API for callers to use? It depends upon the problem and the deployment model: a LAN based system can be far chattier than a WAN solution, where delegated workflows and perhaps asynchronous callbacks or polling the appropriate notification mechanism. A solution designed for LAN operation either explicitly in the requirements or implicitly (in the implementation), does not easily transit to WAN operation. The whole REST versus SOAP debate is another issue, which will take time to resolve. The exact mechanism by which a client talks to a Web Service does not make any different what the development and deployment process should be. It does affect the API, as does the choice of doc/literal SOAP versus RPC SOAP.

Ignoring the SOAP/REST debate, here some tips about API design which should be relevant to either:

- Any document based design should be in XML format with a DTD/Schema to validate it before sending. Providing a simple validation entry point could be useful to client testing: the payload is parsed and validated but not executed. Validation obviously encompasses more than simple schema validation; it can check the arguments for being legitimate as a whole. An XSLT implementation of the basic validation tests is ideal; this can be given to callers as part of the interface specification.
- Exception code data should be machine and user parseable. Add:

1. Unique machine/VM instance ID (can be disguised)
2. Error code number/string/GUID
3. Text for developer at far end
4. Text for operations to read and understand

We have used COM `HRESULT` codes as the error number, as they could encompass legacy applications. We added and documented internal error codes (including "Internal Java Error + exception string", and distinguished some network errors (incomplete calls to dependent system) which were likely to succeed if the request was retried. The only problem was that the union of `HRESULT` + `java.lang.Exception` was a very large set.

- Callbacks are problematic. Polling over HTTP is the only way that works through firewalls. Alternate channels (Instant Messaging, SMTP and SMS) may work, but this vastly complicates the process. Perhaps polling with an email callback to trigger a call is a viable short-term mechanism.
- SOAP with Attachments enables large binary file upload, as can POST or PUT to an intermediate filestore with URL upload. SOAP with Attachments is the most consistent paradigm with a SOAP based marshalling layer; POST is simple to implement server side, and ties in with the REST Web Service model. Use of an indirect jobstore may be useful, but it adds a lot to the service, unless many services can share the same job store. As there is no uniformity of support for binary across different SOAP implementations, large binary files are trouble. S As a fallback, base-64 encoding in the XML body does work, albeit inefficiently.
- Consider including a simple client call that simply echoes back the parameters, returns an uptime or some other low-load response, an endpoint that can act as a ping test from the client application, and even directly from a web browser, which is useful when fielding support calls.

4.2 State and Sessions

State is problematic in Web Services. There is a simple model for web applications: the User Agent stores session cookies that are used to index to state stored server side. A Web Service may have a slightly different notion of what constitutes a session and how long it lasts. Fortunately, it is possible to code some more complex session negotiation into the service process, to deal with partitioning and security

1. After authenticating, the server sends a new endpoint that will be valid for the duration of the conversation. This endpoint could be session specific.

This is actually somewhat hard to describe in WSDL: there is no way to state that the response to a request is new endpoint implementing a known service [W3C02b]. You have to simply return a string which the client must manually use as the endpoint for a new service.

2. The authentication process can return an opaque token to the client; this token is included on the follow-up messages as cookies or a SoapHeader.

3. A load-balancing router that is aware of cookie-based sessions can redirect calls to the favored server for the duration of a session, redirecting to another server only when the original server goes off line. This boosts performance as only core session state needs to be stored in a database; transient data structures can be created and cached on the session specific server. There are two flaws with this approach. First, there is the well-known issue that Level 7 routers cannot inspect the state of HTTPS requests and so redirect based on cookie content. Secondly, it is not clear that cookies are the best method of representing session state in a Web Service API. They work on the web because that is all there is, but Web Services can do better, particularly given that not all SOAP implementations support cookies properly (such as the MS SOAP toolkit 2.0, or anything built atop `java.net.HttpUrlConnection`).

If an alternative to cookies is used to represent session, such as a token passed in as one of the parameters of the request, or in SOAP headers, the current generation of routers will not use this information to redirect the calls to the appropriate server, even if the request is sent in the clear. This can only be resolved by having routers aware of the actual protocol being used, able to parse requests and responses and determine session information.

In the meantime, session-specific endpoints can redirect to a single server, or a small cluster of servers in a larger farm.

4.3 Robustness

Web Services are designed to work over long and unreliable links, something that must be factored into the API design.

- Always set the content-length header.
- Do not trust HTTP without validating that the amount of content received matches that in the content-length header.
- Consider including a checksum header to also validate the data received matches that sent

- What happens if a client disconnects during a request? Does the server handle this gracefully?
- What happens if a hundred valid requests are suddenly disconnected before a response is sent back?
- What happens if the server disconnects during a request? Does the client handle this gracefully?
- What happens if the client is expecting an XML response, and HTML, such as a stack trace, comes back instead?

The HTTP protocol provides no warning when incomplete data was received, it is up to the caller to check the content-length header and then verify that no more or no less was received. This header is optional on HTTP/1.0, mandatory on HTTP/1.1. Server code should always set it, so client code can test it. Client code that supports both protocols should consider reporting an error on a mismatch, but do nothing if no header is set.

Even with content length validation, there is still the issue that the content itself is not validated: a transposition or complete replacement would not get picked up. A checksum is needed here; one can always be sent in the response headers for those clients that know of it.

This exposes a flaw with HTTP; there is no intrinsic checksum validation of responses, except at the TCP level. HTTPS implicitly validates the stream during the encryption process, leaving only the risk that something behind the HTTPS front end server can still corrupt the data.

4.4 Configuration

Being able to control the configuration of a system helps operations keep it running, and make it customizable for end users. At the same time, it leads to big problems when the configuration gets changed and everything breaks.

- Split customization into harmless and harmful areas, such as decoupling: the configuration of skin of the service from the configuration of its behaviour.
- Lock down service configuration. If nobody would possibly want to change it, make it a compile time option and not a value that operations can control.

- Any dynamic configuration options should be changeable in one place and automatically propagated out to all servers. Anything else does not scale and increases the likelihood of configuration differences across systems
- Provide a sensible default configuration that works and is secure.
- Try and support hot-updates of changes
- Provide a mechanism to view the entire configuration of a system.

This is hard to get right. Hard coded URLs and values in ASP/JSP pages are definitely dangerous; even a per-system XML configuration file does not scale across a server farm.

Database storage of the configuration data is the obvious choice when the database is present, and there is some easy means of directing service instances at the service.

A directory service based design is an interesting alternative. Directory servers are readily available, and designed to provide replication and failover, so that a single directory service interface can control multiple partitioned instances of the same service. Furthermore, LDAP support is available across all the Web Service programming languages. Consider this approach for any database-free service implementation.

Chapter 5

Client Design

How can one design client programs to work well with Web Services? If the server and protocol is well designed most of the benefits trickle through into the client. There are still a few items that need explicit support on the client. Obviously, many of these design concepts also apply to a Web Service that calls a dependent service.

Do not rely on the server being local

Assume the server is distant and connected over slower links, multiple routers and through a firewall.

Support the local proxy server configuration

Lots of systems have proxy server access; your software has to work with them. Do not require your application to make the users re-enter what the system knows; In particular, windows does let applications examine the proxy settings for IE: use this data.

Be robust against transient failure

Even if the server is available, transient network failures can fail a request. Be prepared to retry, perhaps using an exponential back-off algorithm. Include

timeouts on requests. The system designers also need to consider how to handle transient failures inside a distributed server installation: do they retry or do they propagate the failure information to the client for them to retry. In our project, we resorted to, propagating the failure back up over multiple requests till it reached the client application, using a special HTTP error code to indicate a transient failure for which a retry was recommended. This enabled only one system to implement retry logic, rather than every part in the chain.

Validate responses

Match content-length headers and message digests against the received data. Compare the request-ID echoed back with that submitted. Anything on the chain: proxy server, modem, router, load balancer can potentially corrupt the data.

Implement a service probe test

If the server implements a simple status, health or echo method, call it. This provides an easy way to determine if the service is reachable before submitting a complex request.

Provide a debug log facility

This is not for the end users, it is for the end users to email to the Web Service support when they field support calls from the customer.

Implement off-line support

Windows has an off-line mode, so does Java Web Start. Use them and the application software can degrade functionality gracefully when the user is somehow disconnected. or the service off-line.

Chapter 6

Interoperability

The whole reason to use an XML marshalling mechanism rather than a binary one, such as RMI or DCOM tunneled over TCP port eighty, is for interoperability. Clearly, interoperability between other implementations of the Web Services protocol stack is imperative for the Service to be useful. Systems which implement their own XML-based protocol leave it is up to the clients to implement their side from the ground up, working off sample code or specifications. Precisely because this can be over-complex, using a standard protocol such as XML-RPC or SOAP should make it easier for people to talk to your service. XML-RPC, by virtue of its simplicity [Win00], is very easy to interoperate with: there are so few datatypes that implementing XML-RPC support from the ground up is trivial.

SOAP is a different matter. SOAP version 1.1 has its own set of datatypes; the infamous "Section 5" datatypes [W3C01]. SOAP implementations support the majority of these, and a large (but seemingly never complete) selection of XML Schema (WXS) datatypes, as does the WSDL language for specifying service interfaces. Ultimately, XML Schema is the only representation that makes sense. If the application code can read and write data to XSD defined XML files, then it is independent of the particular wire protocol underneath.

While that is a good ultimate goal, the SOAP datatypes have the benefit of being much less complex than XML Schema, and more tuned with the basic datatypes that applications support: strings, integers, booleans, floats, enumerations and arrays. Equally importantly, most SOAP implementations are likely to have the basic SOAP datatypes thoroughly implemented and tested. This makes the SOAP representations a good format for basic interoperability, even though that is still not guaranteed. Note that to handle the fact that any of the Section

5 datatypes can be "nil", JAX-RPC maps the datatypes not to any primitive language counterparts such as `int` and `float` to their `Object` counterparts, `Integer` and `Float`, which may have consequences in performance and code complexity.

Long term, the Section 5 datatypes will be supplanted by XML Schema, which raises the bar on interoperability [Rub02]. XML Schema can describe many complex datatypes, including many that do not map easily to those in popular programming languages. For example, it is easy to restrict the scope of a type, creating datatypes such as a floating point number that is only valid from -180 to +180, yet restrictive subclassing is not explicitly supported in most languages. For arbitrary callers to be able talk to a service, it must not rely on such restrictions.

Even seemingly innocuous types such as `xs:unsignedInteger` have interop issues. They only work well with languages like C++ and C++ that support unsigned integers; Java does not and so SOAP implementations have a harder time knowing what to do with it. The JAX-RPC specification ignores those datatypes completely [Mic02b].

Moving beyond the simple datatypes, the real problem with SOAP interoperability comes with complex datatypes and serialized objects. Most SOAP implementations generate WSDL specifications directly from the code, promising remote service access without having to spend time with an interface declaration language and IDL compilers. Effectively WSDL is the new IDL, this time XML based rather than derived from C and C++, and dynamically generated by default. This dynamic generation does simplify development, but it is a dangerous simplification. It is no longer obvious when developers have accidentally changed the interface such that callers need to be rebuilt, and it is no longer obvious how the system is going to marshal content over the wire. When writing an IDL file developers had to think about what they would send over, how to represent data in way that the callers could work with it, and essentially be forced to worry about interoperability. When writing a Java or C# class that is automatically turned into a Web Service by the framework, they may just use a standard framework object for that language, such as a `Hashtable`, and be surprised when the system at the far end of the wire cannot work with it. There is a `System.Collections.Hashtable` in C#, and a `java.util.Hashtable` in Java, but nowhere in the SOAP, WXS or WSDL specifications are there such datatypes, and nowhere does it say that such language specific collections are exchangeable.

It is this combination: modern languages with intrinsic collection frameworks, and a service development process driven off the implementation code, rather than off any interface specification, that creates interoperability problems. There are no silver bullets here, other than being rigorous, even if the tools do not

mandate it.

- Create the WSDL file first and implement your service against this file, rather than the other way around.
- Use XML Schema datatypes, not the SOAP section 5 datatypes, and avoid use of facets to restrict ranges.
- Avoid `xs:positiveInteger`, `xs:negativeInteger`, `xs:nonPositiveInteger`, `xs:normalizedString` and the unsigned types. For interop with any Java systems, these datatypes cannot be used.
- Test for interoperability, from the outset of the project.

  To test for interoperability, build clients in other languages —this is not as hard as it seems. Ant1.5 has a `<WsdlToDotnet>` task to convert a WSDL file into C# or VB.NET; if the WSDL cannot be processed that target will fail with an error. Likewise, Apache Axis has an Ant task to generate Java code from a WSDL file; C# service developers can use this to verify that their service goes the other way [HL03].

  Writing these interop tests in the popular languages and SOAP toolkits is currently the only way to reliably validate interoperability. If running the tests can be automated, they will become an essential feature of every deployment cycle.

Chapter 7

Versioning

How are versions of a service to be handled?

Once a service is officially deployed, its behaviour is effectively frozen: the endpoint, the interface and the semantics of that interface.

To be able to move endpoints, either require callers to locate the service using a directory service, such as a UDDI registry or simply define a new hostname for each version of the endpoint, and move them around as you need. For example a service could have its first version implemented on http://v1.example.com/api; a second edition could use a new hostname in its endpoint, such as http://v2.example.com/api.

Changing the exported functions of an existing SOAP endpoint is difficult. Although one can extend a service with new actions, changing the arguments or return value of an existing action can break calling code. The best way to do it is to use the doc/literal model, rather than SOAP's RPC model, so that the XML body of the request can be extended within the limits of the schema. To support options not permitted in the original schema, a new schema can be defined; this approach obviates the need to have version attributes in the XML itself, and offers much more extensibility.

What is hard to maintain is consistent versioning of service semantics, both explicit "this request updates the database" and implicit "this update takes less than 12 seconds". A good development team can maintain the explicit semantics, especially with a good regression test suite, but implicit semantics are often a function of the deployment configuration itself, and are hard to control.

Another potential trouble point for SOAP-based services is that SOAP is evolving, an evolution that SOAP toolkits try to track. This will create SOAP versioning issues at some point in the future. If the SOAP implementation is provided by the application server, or even the underlying platform, then when these system components are upgraded the service may be forcibly upgraded to a later version of SOAP. If this is the case, the only way to maintain SOAP stack compatibility with previous versions will be to freeze the entire server configuration. This is a potential problem with the .NET server SOAP stack and some of the J2EE application server stacks. The solution for Apache Axis and other drop-in SOAP toolkits is of course to keep the toolkit in the software configuration management repository.

Consumers of Web Services need to be aware that versioning will be an issue, and should create a set of tests to verify the behaviour of each aspect of the service they use. The client code will act as a test itself, but having a separate test suite makes identifying faults easier. Which is simpler: fielding a support call that the client is not working, or getting a JUnit generated HTML report of where an updated server is failing your acceptance tests. It may seem excessive, proposing that Web Service consumers write and run their own regression tests against a service implementation, but if you are a consumer of a service, who else is going to look after your interests?

To close on the subject of versioning, it is inevitable that this will become an issue over time. If there is one thing the computing industry is consistently bad at it is is versioning. Despite debacles such as "DLL-hell", new environments, such as Java, replicate the problem. The .NET framework is unique here in that programs are forever bound to the original version of libraries, but Web Services offers to undo that gain by raising the complexity of the problem. It will be irrelevant that an application is bound to the same version of a library it was built with, if that library talks to a web service that now acts differently.

Chapter 8

Securing a Service

A Web Service needs to defend itself from malicious attacks, which can take many forms. It also needs to be trustworthy to callers, which means that it should be resistant to man-in-the-middle attacks and to someone impersonating the server. HTTPS or other public key systems are good for authenticating both servers and clients.

A security problem with web applications and services is that any security breach, anywhere in the system, can compromise the entire machine. Put bluntly, it does not matter if a service is bulletproof if the Code Red patches are not installed on the server. Security holes are different from classic software defects, where a defect in one path of the program prevents that path from being followed. An application can be 99.5% bug free and still be workable; a service that is 99.5% secure is still unsafe to deploy.

It is our untested hypotheses that a Web Service may be intrinsically more vulnerable to security attacks than a web service. This premise is based on the simple observation that many security attacks go through active code, such as badly written CGI-BIN shopping cart programs, rather than fundamental bugs in the web server itself. A Web Service is really nothing more than a large number of programmatic entry points to custom code, along with a system-generated WSDL description to make understanding the entry points easier, and perhaps even a WSIF description to list those same endpoints [B+01]. These same features which make finding and binding to a Web Service easy for trusted callers, may also make automated malicious attacks easier.

This does not mean that there is any validity in Scheier's claim that SOAP is as big a security disaster as ActiveX [Sch02]. Callers of SOAP services are

not exposing themselves to any security risk unless they support some means of callbacks. Without the callbacks, there is no way for a program behind a firewall to receive and process a SOAP message, and thus no security risk. The SOAP server is a different issue, of course.

Here are some basic security guidelines; none are significantly different from those of a normal web site, except that because XML data often exchanged as the payload of an RPC call, the application needs to sanitize that data more carefully.

Have no limits on string buffer size. This avoids buffer overflows. Actually systems can have limits, provided they are explicit and they test for content being in range on receiving requests and reject oversized content. Safest is to avoid languages that are prone to buffer overflows. Even such languages as Java and C# are vulnerable to buffer overflow attacks wherever data is handed to the underlying operating system via helper libraries. Which means, in the absence of a pure .NET or Java platform, that buffer overflow attacks are still possible.

Treat all incoming data as a threat. Assume all received data is a threat unless proven otherwise, regardless of originating IP address. Sanitize it, sterilize it and never use it raw. Perl is very good at this; Java and C# do not implement 'tainted' data out the box, and though a tainted strings class could be implemented, it would not be as rigorous as one built in to the basic class.

For XML processing, URLs inside XML document need to be filtered so that `file:`, `classpath:` and other protocols that provide access to server side resources are intercepted. Catching `xlink:href` and `xsd:include` references is also imperative. Note that simple string matching does not filter these strings, unless the XML has been canonicalized.

Filter on caller. If operations have a finite list of allowed callers, they can hard code that into the router. If the access control is only to some parts of the server, use http filters to validate access, rather than tests in individual classes and JSP/ASPX pages. It is too easy to forget those.

It also means that any early validation "is this a legitimate request" can resist DoS attacks, as the amount of computation expended is less the earlier you reject a request.

Never use clear text authentication. Basic HTTP authentication is only secure over HTTPS links; even that sends a shared secret over an encrypted

wire.

Digest authentication not only avoids sending the password over the wire, so can be used to authenticate over HTTP channels, it effectively signs the request, ensuring that the far end can verify that no changes or errors have occurred. This is ideal for Web Service requests where a simple transposition of digits could have adverse consequences.

Do not store sensitive information in cookies. This should be obvious to experienced web server developers, but merits repetition. Cookies are stored client side. Expiry dates and times are hints, not mandatory commands, so some other expiry check should also be performed server side. User ID and other sensitive values must be hidden and made resistant to tampering. Session theft is a common attack on web applications [AtS02] —there is no need to replicate this vulnerability in Web Services.

Do not trust IP addresses or reverse DNS alone. A compromised system can use false IP addresses when attacking another. The system needs a better authorization mechanism than simply filtering by IP address. Digest authentication is good. Using a session handle returned after the logon request that is included in the follow-on requests, helps the service endpoints do quick caller validation. The session handle should be an encrypted index to the real session data, and contain a timestamp and some strongly random digits at the beginning to make stream cipher cracking harder, something like the following

```
Handle= Encrypt(random[4]+timeout[4]+index[4]+sessionIP[4])
```

Decoding this can include verification that the session time had not expired and that the IP address was still the same [1].

An advanced technique is to require the caller to do some NP-complete per-request processing (perhaps using the session handle and a monotonically increasing request number), which can be verified easily at the far end. This forces the caller to invest CPU time per request, which makes DDoS attacks harder by reducing the possible request rate of each zombie in the attack,

Filter by URL. If the system can have a separate URL, or better still hostname, per customer, then the router and DNS can be used to direct and filter requests. For example, a file storage service for external web sites could have different virtual hosts for each customer site: site1.example.com, site2.example.com?

[1] Major ISPs (AOL, MSN) can route callers requests through different proxy servers, so IP address validation does not work if the service must support callers from such locations.

etc. Usually these could all point to the same system, but if one site1 comes under a DDoS attack, operations can direct it away so that site2 is unaffected.

Never print supplied string parameters inside an HTML page This is the standard trick used to enable cross-site scripting attacks. Even pages which try to be clever and filter out all tags that contain script, are vulnerable, unless they filter out all tags which can possibly contain DHTML commands. It is simpler and safer to use the `\<IFRAME\>` tag to display untrusted content with the scripting inside that frame disabled, or escape all HTML tags completely, though browser support is an issue there.

Run the service in a sandbox. Both Java and MS.NET provide isolation for untrusted and semi-trusted applications. Run the service in such a sandbox, granting it only the rights it needs to perform its tasks. If somehow the service is subverted, the damage it can cause is then limited to the dimensions of the sandbox. Never run a service with super user/administrator rights.

Defend in depth. Design the cluster such that subversion of a single box does not compromise the entire system

Imagine if someone does gain control of a server so can execute arbitrary commands on it. Should this give it unrestricted access to the rest of the Web Service? It shouldn't. If back end servers also validate user identities and rights then the damages a compromised front end can do is limited. Also, use some penetration detection tools and have a plan in place for a compromise.

Lock down the server. Have nothing on the server is not needed, disable all extraneous bits of IIS and other front-end applications, rename all system tools such as `netstat` and `telnet`. This is actually an area where Win2K and WinXP's "system recovery" feature gets in the way: they view deleting a program such as `netstat` as an error and recover it from the original OS image, whereas operations view deleting such programs and any other surplus bits of the system as good housekeeping. The trick is to keep the install disk inaccessible to the OS after deleting these files to prevent it restoring them.

Trust the staff, but not completely. Although security breaches by staff are seemingly quite low, the damage can be significant. One option here is to be utterly paranoid, but that would require auditing not only all the code written by the team, but all the unsigned Java and .NET libraries used too. It is

too easy in these languages to decompile code, add an extra class or method, creating a back door in an XML Processing Instruction or some other piece of system code. One tactic is to decide where security really matters, and isolate those components and audit them rigorously, while the rest can be treated less cautiously. Signing modules after auditing prevents them from being subverted later. Of course, audited components can no longer trust the unaudited elements.

Get a security expert to test it. To secure a Web Service the underlying server needs to be secure, which means operations need to be up to date with OS and web server security patches, and the operations tests need to probe for those patches. It is rare for servers to regress security patches, but it is possible. More likely is that a new node in a cluster is brought up and some of the patches left out by accident; tests can detect this.

The Web Service itself needs to be tested for security. Here are some tests to consider:

- Can any URL gain access to the source of the service?
- What gets printed/sent back on an error? A stack trace?
- Are directory listings possible?
- What does the client side do if some proxy redirects it to a malicious site? Can it tell if this has happened?
- What happens if someone posts an near-infinitely long request? What happens they post a hundred such requests ?
- What happens to any string validating process if data is sent in Unicode?
- Can one import a text file into an XML payload using XML "&entities;"? What about XML paths?
- What happens when an RPC call is broken before server side processing is completed? What happens if this happens simultaneously on a hundred inbound requests?
- If the validation code searches for certain strings in the XML, what does it do when those strings are escaped using XML's "@" escaping mechanism?

Chapter 9

When Things Go Wrong

No matter how well designed the system is, at some point it will stop working. Be ready for this, preferably as early as you can, as it will ease development as well as deployment.

- Have logging code built into the system, with per package and level control of log data generation. Apache's Log4J is one example, the Apache Avalon project's Logkit another; Sun's Java 1.4 logging API a third if Java1.4 can be guaranteed. Jakarta-Commons/logging provides an abstraction that lets your code integrate with any of these, though it has a bias towards Log4J.
- Don't compile out the debug level logging, you may need on a run time system; just turn it off. Provide a way to turn it on without restarting the server; JMX is the obvious choice.
- Keep line numbers in release packages, for the same reason. Full symbol information is too much of a security risk; it makes reverse engineering easier.
- The logging system should log enough information after an error for the problem to be replicated. This should include session ID and request parameter details.
- Provide a means of controlling the logging level dynamically through an MBean interface. This lets you debug a running system and then throttle logging back afterwards.
- When reporting errors via an RPC response, include a (possibly disguised) machine/instance identifier to help track down configuration bugs.

Example: one server JVM stopped being able to resolve hostnames, but to the caller it looked like a general intermittent failure of one request in eight, rather than a permanent failure of one system. If the error had included a machine and JVM ID, they would have been able to tell us "system 3 keeps failing" instead of "there is an intermittent failure".

- Include email notification for when things go very wrong.

  Example: when an incoming request created a fatal error in a native application, our service emailed the event and the data that caused it to the development email alias, so we could file a complete bug report ourselves. This enabled us to start fixing a problem before the end users reported it, and so narrow the apparent interval between their bug report and the availability of a fix.

- Even if you can find a bug a minute after receiving the report, and upload an fix immediately later, always have a full test and staging and deploy process, forcing 24 hours of lag between report and fix. This not only ensures that the fix is tested, it sets people's expectations as to what the minimum response time is. Once you fix a trivial bug in twenty minutes, management expects all bugs to be fixed in twenty minutes, and you end up distracted from the process of finding and fixing problems.

- Don't send software exception strings to the log and expect operations people to be able to understand them.

A log message of `java.io.NoRouteToHostException` means to a developer that networking problems are keeping a remote system unreachable. To operations, the word "java" at the front meant "software problems", which meant waking the development team.

At the very least, common exceptions need to be noted in the deployment bug tracking database, along with their real meaning and what could be done to fix them.

Chapter 10

Components of the Future

If the Web Services vision is to be realised, then the software components used to construct such products will need to evolve. The areas raised as issues here: deployment, configuration, management, testing and logging all need to be addressed, not just by those software packages a team writes themselves, but by those third party products which are used in the overall system.

The most successful server-side component models in current use are probably COM+ and EJB; .NET is the newcomer.

10.1 COM+

COM+ components are almost a representative "how not to address deployment and availability" design: deployment is usually a matter of hand-filling of dialog boxes on a system by system basis, creating user accounts in different management console windows, and the sole logging standard, the Windows NT Event Log, is slow and inflexible The WMI interface for remote management of components is an integral part of Windows, yet a WMI interface is not a standard feature of COM+ objects. Testing is an abstract concept: there is no real framework for automated testing of COM+ objects on a par with JUnit [Obj01].

Although COM+ components on Windows.NET server can be exported as SOAP endpoints, moving to an XML transport does not address configuration or scalability issues. The SOAP API a service exports should be designed for use with SOAP, not mapped from an existing COM+ API, to address inter-

operability and wide area connectivity.

10.2 Enterprise Java Beans

Is Enterprise Java Beans (EJB) any better than COM+? EJB provides a conceptual model for describing the domain model as a collection of objects, bound to a database. Although it was intended to create a market of re-usable beans from third party developers, with an explicit model of bean producers, bean assemblers, system administrators and assemblers. Ignoring the controversial issue as to whether Container Managed Persistence is a viable approach, the EJB model does at least show developers how to write components which can be used as part of a transaction, be accessed remotely and be tightly bound to a database. The whole process of building and deploying EJB components is somewhat unwieldy, but emergent Ant tasks, such as XDoclet, are simplifying the process. From the perspective of high availability Web Services, it has a number of weaknesses:

1. There is no explicit "tester" role defined, or a standard test API. Cactus is emerging to fill the gap by extending JUnit to server side testing; the complexity of the process shows how testing is clearly an afterthought
2. There is no standardized logging model. Components all had to choose their own logging API. Log4J has emerged as the standard, although there are others, including the Java1.4 logging package. A single API needs to available for re-usable components, an API which can be bound to the application server's single logging mechanism underneath. However, given that the EJB rules forbid direct writing to the file system, or opening of sockets, there is no clear "legal" means to log inside an EJB.
3. There is no configuration model other than pure database state. This is appropriate for the EJB beans themselves, but not for all the other elements of an enterprise Java application, of which a Web Service is one instance. All that is standardized are configuration files in the WAR/EAR archives, which are inadequate.
4. There is no comprehensive notion of management: EJB beans do not come as 'manageable' or instrumented by default.
5. EJB is hard to develop for and incomplete. To achieve scalability and ACID transatctions, many common functions of an application are forbidden: such as threading, file system I/O, and non-EJB networking. Any large application will contain much other than the business logic functionality encapsulated in session and enterprise beans, yet there is no model for these aspects of application development.

6. The RPC mechanism exposes the fact that the implementation is built from EJB; clients need to include the EJB client jars and are tied to the version of the code that they are built with.

One of the fundamental failings of EJB has to be that final one: that all the direct clients need to know that the implementation is fronted by an EJB session bean. This is actually addressed by the Web Services model: if you can export your beans as Web Services, then XML can act as the transport, and WSDL can be used to build up the client side proxy classes. From this perspective, mapping EJB objects to SOAP endpoints is not only a good match, it is an ideal way to hide implementation details from the caller.

10.3 .NET

The .NET framework is still new; its weaknesses will take time to become apparent. Its persistence model does not stir up as much controversy as EJB, because it has no equivalent model; this also means there is less instruction as to how developers should construct their applications. The management side of .NET is clearly incomplete: it is easy to write a new management entry point, but you then have to write a Microsoft Management Console (MMC) snap-in to work with these objects. A .NET aware management infrastructure that used class introspection and metadata to provide a dynamic management UI is the obvious improvement.

All this may change in future: one may hope that the 'blessed' design patterns do address the needs of a high availability Web Services. There is an obvious risk that the focus of .NET evangelization will be on the speed of development and execution over Java, rather than the ability to write quality code in the framework. Given the slow takeup of the NAnt and NUnit tools in the .NET community, and the lack of encouragement from Microsoft for the use of these tools to raise the bar on build and deploy processes, there is some risk that .NET Web Service development processes will be behind that of Java.

This potential gulf could be addressed by taking the best aspects of the leading edge Java build tools and processes:

- Universal JUnit based testing
- Universal Ant based build processes integrating building, testing and deployment
- Continual Integration servers such as CruiseControl, the Gump and Anthill [FF00, Rub01, urb01].

- Autogeneration of JUnit test cases from WSDL files in Axis [Axi01, HL03]
- Seamless integration of Ant and Junit into the IDE.

The first step would be to create a test framework as powerful as JUnit, integrating it into Visual Studio. The IDE's wizards should automatically create tests for components it creates, web services it imports, and web sites it develops. These tests would be runnable from the IDE, and from an automated continuious integration server that could be built atop the IDE, or a separate component.

Similarly, a better logging API is needed, and again the Visual Studio wizards should make it easy to include this logging in an application or component, and easy to configure.

One could point to the .NET ports of Java tools: NUnit, NAnt, Log4net, and say that the basis for this solution exists. That may be so, but without support and integration from Microsoft, these tools are likely to remain on the sidelines.

10.4 Improving the component models

Could it be done differently? What would an ideal component world for Web Services resemble?

Imagine being able to buy or re-use components which can be glued together to build server side components. Each component is represented as objects that can be created directly, or as some object whose lifecycle the application server manages, and which must be located or created via some factory.

The component is configurable, supporting a configuration mechanism that determined by the application server: the server could extract the configuration data from a database, a directory service or simply an XML file; the component does not have to care. All it needs to know is that it gets 'configured'.

Each component would have a management interface. This would be a JMX or a WMI service interface depending upon the underlying platform. To implement such a service, all the component had to do was state in the metadata associated with methods that a method was part of the management interface; the build tools or the runtime would extract this metadata and bind those methods to a management interface.

The same model would apply to logging: a component would log events at a

debug level of verbosity, and leave to the application server to control the actual log.

The application server would also be instrumented, so that you could turn on logging for a particular SOAP endpoint, which would give a trace of inbound messages, perhaps even performance statistics on computation time. The instrumentation provided in the server would be tightly integrated with the client code: all could be saved to a file, streamed to a remote client or viewed in a local GUI.

The final element of the vision would be implicity scalability: up, down, and out. It may be that components designed for a grid architecture will deliver the best outward scalability, but the components are easy to develop and deploy on a single system, developer adoption will be limited.

Is this an unrealistic vision? Perhaps. But unless we start to improve the current state of affairs, we will never get there.

In the Java world we have three competing logging APIs: Jakarta Log4J, Jakarta Avalon Logkit, and now the Java 1.4 logging API; if an application uses the Jakarta commons-logging logging API they can actually be written to be independent of all three APIs, let log through any of them. This gives independences at the expense of another layer of indirection.

The closest we have to XML/database/LDAP bindings is probably the Castor work from Exolab. These tools will bind Java objects to XML Schema described XML files, LDAP servers or databases via JDO —but the classes do have to know what they are binding to: they have to pick one of the three. If someone were to write an LDAP wrapper around XML and another to a database table or two, then LDAP configuration should work for objects regardless of the underlying source. This would seem to be a fruitful area for prototyping.

Chapter 11

A component model for the federation of Web Services

If the federation of services that will comprise the set of publicly accessible Web Services are viewed merely as a set of software components, what should their component model be?

One obvious question is *should they share the same component model of the implementation?*. The COM+ to SOAP bridge of Microsoft .NET Server does let one export a COM+ API as a SOAP API, while similar bridges for EJB can turn a session bean into a SOAP entry point.

Many of the issues covered in this paper: interoperability, security, state and versioning, and the whole challenge of working over long haul connections strongly argue against doing so. Federated services on the scale of the Internet are so different from the piece used to build them, that one cannot safely extend LAN-scoped objects and endpoints to the scale needed.

> *Should Web Service protocols be used as the glue between elements inside a web service?*

Individual elements in a Web Service, or indeed, any other large software system, can be implemented as Web Services in their own right.

This provides two benefits, one immediate, one potential. The immediate benefit is that the Web Service protocols can be used to glue together disparate systems, written in different languages or running on different technologies. For example,

a Java Web Service could use a "legacy" Win32 Web Service running on the local system, exporting a SOAP interface to low level system calls. We have used this technique in the opposite direction, with ASP pages calling a Java Web Service for authentication.

The other potential benefit is that the exported interface can be re-used in other applications, or perhaps underlying implementations of individual components changed without the rest of the system needing modification. This is in marked contrast to using interconnection technologies such as COM+, EJB/RMI-IIOP or .NET's `tcp:` remoting, as these technologies effectively tie a system's implementation to their underlying technology for the life of that system.

Using an HTTP based API incorporates the front end load balancing components of the system, such as the router, to balance intra-stack requests and handle failover.

There is still a configuration problem; either the endpoints need to coded into the system configuration files, or a UDDI server has to be set up and run in the stack. The Mir prototype extension for Apache Axis uses multicast IP to locate URI-identified endpoints on a LAN, so can be used to find available services without any need for a central registry or advance configuration [Lou02]. This could be a flexible solution for a registry-free boot up process. The first service to locate may of course be a UDDI registy; multicast service location simply aids in finding this registry.

There are strong arguments against using the current SOAP protocol stack everywhere in a Web Service cluster. Firstly, it lacks transactions and authentication, key features of the existing technologies. If these are needed, then SOAP is not viable. Secondly, it may be extra engineering effort, which, as the XP programming philosophy espouses, is only justifiable if the results are immediately tangible. Thirdly, such an exported interface would needs to worry about interoperability and security, just as if it were a public API.

Finally, it makes the system more complex. The routers and the servers need to clearly distinguish public endpoints from private endpoints, and the routed-through-the-Internet tests need to verify that the private endpoints are not accessible. Using Web Service protocols to bridge disparate systems in a single server stack may seem admirable, but having such heterogeneity inside a single stack increases development, management and maintenance costs. There is a lot to be said for the simplicity of a single language, single platform implementation.

Overall then, it is hard to come out completely in favor of blindly using SOAP for intra-stack communications. It has a place, but it needs to be used carefully. As the Web Service protocols evolve to include missing features, and integration with platforms more seamless, it may be usable more widely. Security and

configuration issues still force caution, regardless of how much the underlying techologies improve.

Chapter 12

Summary

Writing a Web Service is like writing a normal application except for the need to implement near 100% availability, be secure and deliver robust distributed software functionality on web timescales. The key to this is to adopt a deployment centric process that involves operations from the outset, in communicating their requirements and deploying early builds of the system. All the foundational techniques of modern software development processes: use cases or stories to focus development, defect tracking and testing can be applied to the operations and deployment process. By doing so, the delivery process of Web Services can be eased.

Issues that have to be addressed in the process cover security, scalability, manageability and configuration; problems that other server designs have long encountered. New to Web Services is the API design itself: a good API for a LAN is rarely a good API for a WAN and vice versa. There are still many problems in this area, from interoperability to versioning —problems that do not all have solutions.

Finally, there are ways we can improve the component model of application frameworks, ways that are independent of the uses the components are put too. At the very least, developers need all components supporting configuration, management, testing and logging consistently with all other components in the system. We can achieve this if the organisations that produce the component models enable and evangelize this process, or the development community itself adopts the methodology and tools to realise the vision. Although Apache and JUnit have demonstrated that community developed tools can lead the way in development processes, they have yet to make significant headway into the design patterns of applications themselves.

The new Web Services ecosystem is a new environment, one that presents an opportunity to apply the lessons from the past into the new world, and guide developers into writing stable, scalable, manageable, web applications out of components which exhibit the same characteristics.

Acknowlegements

Acknowlegements to Charlie Amacher, Mike Bialek, Debi Braught, Otto Gygax, Sally Kaneshiro, Debi Johnson, Larry Mull, Mark Newsome, and Fred Taft.

Bibliography

[Ant02] Ant Developer Team. *Ant 1.5*, 2002. http://jakarta.apache.org/ant/.

[AtS02] AtStake. All applications are not created equal. Technical report, AtStake, 2002. http://www.atstake.com/research/reports/.

[Axi01] Axis. *Apache Axis*, 2001.

[B+01] Keith Ballinger et al. Web services inspection language. Technical report, 2001.

[Blo01] Joshua Bloch. *Effective Java*. Addison-Wesley, 2001.

[Bul00] Dov Bulka. *Java Performance and Scalability*. Addison-Wesley, 2000.

[FF00] Martin Fowler and Matthew Foemmel. Continuous integration. Technical report, 2000. http://www.martinfowler.com/articles/continuousIntegration.html.

[FGM+] R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. Technical report, IETF.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. dissertation, University of California, 2000.

[HL03] Erik Hatcher and Steve Loughran. *Java Development With Ant*. Manning press, 2003. http://manning.com/antbook.

[Kru00] Philippe Krutchen. *The Rational Unified Process*. Addison-Wesley, 2000.

[Lou02] Steve Loughran. Mir: Multicast endpoint resolution. 2002.

[Mic02a] Microsoft. Global XML Web Services Architecture. Technical report, Microsoft, 2002.

[Mic02b] Sun Microsystems. *JAX-RPC Specification*, 1.0 edition, 2002. http://java.sun.com/xml/downloads/jaxrpc.html.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[Obj01] Object Mentor. *JUnit*, 2001.

[Par74] D. L. Parnas. Use of abstract interfaces in the development of software for embedded computer systems. 1974.

[Rub01] Sam Ruby. The Gump, 2001.

[Rub02] Sam Ruby. To infinity and beyond — the quest for SOAP interoperability, 2002.

[Sch02] Bruce Schneier. Crypto-gram newsletter, June 2002. http://www.counterpane.com/crypto-gram-0006.html.

[Syr01] Mark Syrett. private communication. email discussion, 2001.

[urb01] urbancode. *Anthill*, 2001.

[W3C01] W3C. SOAP version 1.1. Technical report, W3C, 2001. http://www.w3.org/TR/SOAP/.

[W3C02a] W3C. SOAP version 1.2 part 0: Primer. Technical report, W3C, 2002. http://www.w3.org/TR/soap12-part0/.

[W3C02b] W3C. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2002. http://www.w3.org/TR/wsdl.

[Win00] Dave Winer. XML-RPC, 2000. http://www.xmlrpc.com/spec.

[XDo01] XDoclet. *XDoclet*, 2001.