

# THE SOA HERETIC

---

PUTTING THINGS IN PERSPECTIVE

PRINTED: SUNDAY, AUGUST 13, 2006

---

© HILBERT COMPUTING, INC., 2006



**TABLE OF CONTENTS**

**INTRODUCTION..... 1**

**WEB SERVICES..... 2**

**The Web Services Vision.....2**

**Loosely-Coupled? Really?.....3**

**SERVICE-ORIENTED ARCHITECTURES..... 6**

**Statelessness.....6**

**Synchronicity.....7**

**Reuse Models.....8**

**Diagnostics.....9**

**Performance..... 10**

**Granularity.....12**

**RECOMMENDATION.....13**



## INTRODUCTION

There is a lot of industry discussion on Service-Oriented Architecture (SOA). Architecture by its very nature is abstract and as an architectural approach, SOA is no different. That abstract nature leads to a lot of misunderstanding and misapplication of this technology. Just try to find the single definition of SOA and you will find hundreds.

I like to start with the definition in found on the DM Review website<sup>1</sup>:

*A service-oriented architecture is a collection of services that communicate with each other. The services are self-contained and do not depend on the context or state of the other service. They work within a distributed systems architecture.*

From this definition, we can see that there is an opportunity for autonomous requests for services across a network, even between different enterprises. That creates the potential for a different approach to how software is constructed.

Service-Oriented Architectures are typically oriented toward using Web Services as the primary service abstraction, although most implementations support other connectivity to support past interoperability approaches.

I have been a member of multiple teams that have successfully implemented Web Services within an enterprise and between disparate enterprises. It works well and Web Services was the implementation-of-choice to solve the business problem at hand. However, Web Services and SOA – as with almost everything else in IT – is accompanied by a wave of hyperbole by the industry pundits and the leading vendors. This will discuss some considerations when looking at the applicability of a Service-Oriented Architecture within your enterprise.

---

<sup>1</sup> <http://www.dmreview.com/rg/resources/glossary.cfm?keywordId=S>

## WEB SERVICES

A Service-Oriented Architecture can be implemented using a variety of connectivity technologies. The newest and most widely-advocated connectivity approach is to use Web Services. Before we dive into SOA, it's best to understand the motivation for Web Services.

### THE WEB SERVICES VISION

The World Wide Web changed the way in which people think about information technology. In the beginning of computing, there were autonomous, stand-alone computers. The introduction of client/server technology in the 1980's started to get us thinking about internetworking and cooperative processing, although we mainframer's had been doing remote job entry for quite a while by then.

The Sun Microsystems vision of “the network is the computer” started shifting the thinking toward a collection of computers cooperating to accomplish a goal. Internetworking with TCP/IP already involves a series of cooperating services. The Domain Name Server (DNS) provides name-to-address resolution. Most likely, your desktop computer's network address is determined and served by a DHCP service. Many of the applications developed in the last ten years use database services in which the database engine is responding to requests over a network, although the networking is transparent to the developer.

In the 1990's we saw an emergence of distributed computing for a small subset of enterprise applications. First, RPC<sup>2</sup> enabled procedural programmers to access services on a remote machine. As object-technology gained momentum among developers, the notion of a remote object surfaced. CORBA<sup>3</sup> enabled access to remote objects, much the same way in which RPC allowed access to remote subroutines. With CORBA, programmers could access remote objects (“services”) in a way that was language and operating systems agnostic.

- 
- 2 RPC stands for “Remote Procedure Call”. It enables procedural languages like C to call a subroutine where the actual execution occurs on a remote computer and the results are sent back to the requestor.
  - 3 CORBA stands for the “Common Object Request Broker Architecture”. It is very similar to RPC, but has an object-oriented semantic.

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

But the World Wide Web was fundamentally different. It is genuinely loosely-coupled. There was no prior internetworking relationship that had to be established before computers could communicate with one another. All you needed was a web browser and a URL<sup>4</sup> and you could access global information seamlessly. Once you got connected to a web site, hyperlinks could take you to another computer – perhaps in another country – in a seamless manner. You didn't even have to know the URL, since that was embedded in the web page.

The vision of web services grew from that realization. With RPC and CORBA, programmers had to establish a semantic relationship. That is, we had to know where the remote service was, the inputs to the service and how to interpret the results of that service. If it was possible for software to establish the same kind of ad-hoc, seamless relationship as people running a web browser, it would be a revolutionary view of information technology. Software could produce and consume information in an automated fashion from computers around the world. It would be a huge productivity boost for businesses and end-users.

### LOOSELY-COUPLED? REALLY?

Every time we see a presentation on Web Services or SOA, the term “loosely-coupled” appears within the first few slides. The notion of software coupling isn't a binary thing. That is, we can't view it as being coupled or uncoupled. It is a “shades-of-gray” thing. Essentially, coupling describes how much one piece of software has to know about another piece of software before it can be effectively utilized. To put it another way, it is a level of abstraction between software elements.

On one end of the spectrum – the close-to-ideal end – is the web browser. We don't have to know hardly anything to interact effectively with a remote software systems. We can start with a URL and “surf” among a global network of computers in a seamless fashion.

On the other end of the spectrum is the monolithic software systems typified by COBOL applications. When software is developed with a monolithic application architecture, every part of the application has to have a complete understanding of the data that comprises the software implementation of the business process. Most COBOL applications have a global data area that is updated by multiple algorithms that model business processes. A change to the global data structures usually requires a wholesale change in the application. That's very expensive.

---

4 URL stands for Uniform Resource Locator. It looks like: <http://www.google.com>

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

Object-oriented languages were developed to provide programmers with language semantics that enabled the introduction of abstractions that decoupled the need for one part of a software system to know about another part of a software system. That enables each object to provide a discrete, encapsulated function, largely independent of the larger application in which it is used. Object-oriented languages introduced the notion of an “interface” that is essentially a contract to software that invokes algorithms, called methods, such that the implementation of that algorithm is unknown to the caller. That abstraction allows for a significantly greater resilience to change. Resilience to change means a shorter time to market as software systems adjust to the ever-increasing pace of change in the way our businesses have to operate.

Object-oriented systems are more loosely-coupled than the procedural software systems typified by COBOL applications, but there is still a significant coupling. We write software to the contract represented by the interface. We have to know specific data types – is it an integer, a decimal number or a character string – and, more significantly, we have to understand the *semantics* of the information.

Semantics are important. Just ask NASA. In 1999, the Mars Climate Orbiter spacecraft dramatically crashed on the surface of the red planet. The reason was that one of the contractors, Lockheed-Martin, used the metric system of measurement and other NASA software components used the English system of measurement. A number is not a number. It has semantics. A length of “9” is different if you mean “meters” rather than “yards.”

Where does Web Services fit in all of this? Well, the vision of Web Services is that programs would benefit from being as decoupled as web browsing. There are multiple problems to overcome for Web Services to have this level of decoupling. The first is semantics. Assume I am going to use a software agent to book a plane flight from Kansas City to Rome. The software agent used to call on the services has to know the name and data types of every Web Service out there that can give me a price quote. A theoretical Web Service from Travelocity might have a method called “quoteFlight” that takes a source and destination, a date and a class-of-service and gives me the lowest cost. A theoretical, similar Web Service from Expedia might have a method called “getPriceOfFlights” that also has a source, destination and class-of-service, but returns multiple price quotes sorted from least-to-most expensive.

First, how do I discover these Web Services in the first place? The Web Services stack of specifications defined UDDI<sup>5</sup> to solve that problem. UDDI is essentially a directory service that operates conceptually the same way in which we use phone books. There is a definition of a “white pages” type service and a “yellow pages” type service that would

---

5 UDDI stands for “Universal Description Discovery and Integration.” Sounds impressive.

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

allow us to dynamically discover the Web Services that seemed to fulfill the service need that our software agent had. I think it is safe to say that UDDI has failed. IBM and Microsoft hosted free UDDI services to spur development in Web Services technology and both services have been terminated. Why? Well, there's that semantic thing.

When people are surfing the web, they are pretty good at sorting out the semantic meaning of the content on the page, assuming the content is in a language they understand. Software isn't so good. If we go back to our example of the software agent that books travel, we know that we need a character string for the source and destination. However, we don't know the semantics. Is my source location "MCI" or "Kansas City International Airport?" When I get my price quote back, is it in Euros or U.S. dollars? Semantics are important. Just ask NASA.

So what do you think? Are Web Services loosely-coupled? Well, in a way they are. They provide an abstract interface for some business functionality implemented in software. The abstraction hides the implementation, the programming language and the operating system on which the service is hosted. That's all good. However, the data types and semantics of the data are tightly-coupled. Compared to "surfing the net", Web Services are significantly more tightly-coupled.

Compared to previous implementations of distributed computing, Web Services have essentially the same level of coupling. Both RPC and CORBA abstracted the implementation programming language and the operating system on which the service was deployed. Without the realization of UDDI in any practical sense, Web Services offers little benefit over these 20 year-old technologies. In fairness, the implementations of Web Services has placed a greater emphasis on interoperability than the predecessors. CORBA and RPC tended to work only with the same vendors, whereas, Web Services interoperates between different vendor implementations. Java on Unix really can interoperate with Web Services using .NET on Windows.

## **SERVICE-ORIENTED ARCHITECTURES**

A Service-Oriented Architecture is an architectural approach that creates an information technology landscape that sees a collection of autonomous, reusable services. What we would now call an “application” becomes a collection of services required to accomplish a set of business goals.

There are a lot of great concepts embodied in SOA. I have always viewed distributed computing as a collection of peers. The whole notion of a “client” and a “server” seemed artificial to me. It's good to see SOA take that same approach.

To effectively implement a Service-Oriented Architecture, the underlying software services must be largely decoupled from one another. Depending on the maturity of the software development culture, this can be a significant challenge.

## **STATELESSNESS**

As noted previously, decoupling increases the resilience to change in software systems. Another aspect of decoupling is the notion of “state.” We see this concept in our everyday lives. An automobile has state. It can be in park, reverse or drive. When it is in reverse or drive, it can be stopped or moving. Therefore, the state of the automobile can be described as “moving in reverse” or “stopped and in park.” As we move from one state to another, we go through state transitions. It is not possible to move from all states to all other states. Transitioning from “moving in reverse” to “moving in drive” would be a very bad state transition. The two states “stopped in reverse” and “stopped in drive” (i.e. shifting the transmission) need to occur in the transition path between those two states.

What does state have to do with SOA? In software systems, state is essentially a residual memory of what occurred previously. Web Services in an SOA typically need to be stateless. One of the key advantages to Web Services is that they have the capability to be used by multiple existing systems concurrently and they have the potential to be integrated into future systems not yet envisioned. To accomplish these goals, Web Services can not have any lingering information about any past interactions with systems. To do so would create a coupling that would reduce the effectiveness of an SOA.

Stateless programming adds a considerable additional complexity to software development. One way to remove state from a software service that inherently has state is for the state to be managed by the caller. When a service is invoked, the client is responsi-

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

ble for passing the parameters needed for that specific function, as all Web Services examples will show. However, it is responsible for also passing the state of the client to the Web Service so that the Web Service implementation can understand the context in which the service is to operate.

For many software systems, part of that state is going to be the security credentials that are used to determine the authority of the caller to invoke certain services or get certain information back. Extra care must be taken to securely ship security-related information across a network.

I have been a developer on stateless, inter-networked systems for over ten years. In that time, I have acquired the necessary set of skills to be effective, but solutions to stateless programming aren't intuitive. Your development staff is going to undergo a non-trivial learning curve.

Object-oriented programming outside of Web Services is significantly simpler. Most objects implement the algorithms to operate on data (the “methods”) and contain data internally to manage the state of the object (the “instance variables”). Each time we create a new instance of an object, we get an independent set of variable in which to change and maintain state within the implementation of our methods.

## SYNCHRONICITY

All programmers are familiar with synchronous programming. That's how we typically think of our software. We follow a prescribed set of actions, in some order. We iterate over sets of information and we conditionally execute parts of the code based on where the data and algorithms need to go to accomplish their function. We view the software as actively pursuing its algorithmic goal. Asynchronous programming is different.

Asynchronous programming turns that notion completely upside down. Instead of our code actively pursuing its goal, it is passively waiting for something to do. That “something to do” can be viewed as a discrete functional service, although the implementation certainly doesn't have to be a Web Service or even a networked service. It is just an object within the same programming that performs a discrete unit of work.

The caller to asynchronous services don't get the answer immediately as they do with a synchronous method call. They ask for a service to be invoked on their behalf, but they don't wait for the reply. They are free to work on other duties. At some point in the future, the response to their earlier request will be enqueued to them and they can continue

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

processing with that new resultant information. This programming model will be familiar with programmers who have used message-oriented middleware products such as IBM's MQSeries or Java Message Service (JMS). Most programmers have not dealt with the programming challenges that arise when their request for software services from another component doesn't arrive immediately.

The most common implementation of a Service-Oriented Architecture is to use an Enterprise Message Bus (ESB). The message bus manages the interconnectivity and routing of requests between Web Services and other distributed services implemented with prior technologies, such as message-oriented middleware. Once an ESB is introduced as part of the architectural infrastructure, your programmers are most likely going to require some familiarity with asynchronous programming.

I am a big fan of asynchronous programming models. I have been using them effectively for about fifteen years. They scale very well under rapidly variable workloads. The asynchronicity requires the developer to decouple parts of the application from one another, which leads to a more resilient software base. If your software development environment isn't used to asynchronous programming models, SOA and Web Services probably isn't the best place to start, given the added complexity that it brings.

## REUSE MODELS

One of the benefits that is often cited for Web Services is the ability to reuse code easily and instantly. Once a Web Service is deployed, it can be used by any other peer software without having to repackage and redeploy the software embodied in it. If there is a change to the implementation of the service, then it only has to be redeployed once and all of the software that uses it gets the immediate benefit of that new implementation.

That sounds like a tremendous benefit in operationally managing the enterprise software asset. While it does have some benefits, there are some drawbacks to this approach. First, if we have a software service that is available through a network as opposed to being available within the same program, we have additional points of failure, specifically, the network, the system on which it is running and the Web Services container<sup>6</sup> in which the service is hosted. We can reduce these single points-of-failure by introducing a clustered environment. A cluster allows us to deploy the service on multiple machines and

---

<sup>6</sup> The container is additional server software that supports the Web Service software. It might be Microsoft's IIS server in a .NET environment or a servlet container running something like Apache Axis in a Java environment.

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

possibly multiple network segments to reduce or eliminate a single point-of-failure. Clustering also increases the operational complexity of our enterprise infrastructure.

We also have to look at how we manage the quality assurance process for our software. Since the Web Service will be deployed in a single place<sup>7</sup>, we have, in effect, changed all of the applications that use it. Ideally, we should regression test all of the applications that use the shared service before we redeploy it. However, the complexity of wholesale integrated regression testing of multiple applications is going to significantly increase the time-to-market for those service updates.

Conversely, if the software services is implemented using plain-old objects, we can roll out the updated functionality one application at a time. The scope of the regression testing effort is correspondingly simpler. When we implement plain-objects bound within an application, we also eliminate the points-of-failure within the networks and the distributed systems. That makes diagnostics of a software failure much simpler.

## DIAGNOSTICS

When I started in the business in the early 1980s, the bulk of the enterprise software was running on IBM mainframe systems such as MVS. Interactive systems most typically were running under a transaction processing system such as CICS. Because both MVS and CICS were so rich in diagnostic capabilities and *especially* since there was rarely internetworking by the software components<sup>8</sup>, diagnostics were fairly straightforward in most cases. The software infrastructure of the operating system and the underlying transaction systems were responsible for most of the diagnostics. That burden typically was not shouldered by the application programmer.

When the industry shifted to client/server systems and then to e-Business systems, the underlying operating systems and infrastructure software shifted as well. At the same time, the level of sophistication expected from software systems by the business community rose as well. This increased complexity of software, the increased deployment complexity due to the introduction of internetworking to the software systems and the relative immaturity of the software “containers” supporting the applications led to a perfect storm of diagnostic complexity.

---

<sup>7</sup> Even in a clustered environment, it is a single conceptual location for the service.

<sup>8</sup> Of course, there was a network between the software and the terminal, but there was no peer-to-peer internetworking.

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

The IT staff needed to respond to this increased complexity by raising the sophistication of the internal diagnostic capabilities within their application. That continues to not happen as much as it should, since the decision makers tend to be focused on getting business functionality to market. The non-functional requirements embodied in the infrastructure software tend to get the short shrift. Infrastructure software includes things like application security, diagnostic capabilities and response time metrics for managing service level agreements and capacity planning efforts. Modern software systems can not be maintained in a cost-effective manner without rich infrastructure services.

A typical enterprise application has a web server that handles the initial interaction with a browser. That network stream is passed to an application server, such as a servlet container<sup>9</sup>, in which the application logic runs. That application will likely make calls to a networked database engine to manage the persistent data associated with the application.

An operational person responding to a report of a problem with the application needs to determine which of the primary components – the web server, the application server or the database engine – is the primary failing component. Then, he has to determine the cause of the failure. Is it the operating system, the underlying software service or the network that is causing the problem? That is a more complex diagnostic path. In most cases, the application software isn't very helpful to the operational staff in diagnosing the failing component.

Now envision a system of orchestrated Web Services in an SOA environment. Our single application is now distributed across potentially dozens of services across multiple systems. Each one of those has an internetworking element, an operating system and a Web Services container as well as the application itself. Diagnostics just became an order of magnitude more complex. Before an enterprise considers moving to a Service-Oriented Architecture, they better have rich operational diagnostic capabilities in their existing code base and expect to enrich it even more within the Web Services they develop.

## PERFORMANCE

The first rule of performance tuning is “Don't tune until you have to.” Performance issues only matter when they matter. If 500 milliseconds is fast enough for a unit of work, then it doesn't matter if it could run in 5 milliseconds. That said, Web Services is going

---

<sup>9</sup> Examples include IBM WebSphere, BEA WebLogic and the open-source Apache Tomcat containers.

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

to add a substantial overhead for CPU, memory and network bandwidth. Whether that overhead is *too much* depends on the volume of processing within your environment.

When a compiled object calls a method on another object to invoke a discrete unit of work, the action occurs within the same memory space on the same operating system. The overhead memory costs for the binding between the two objects are essentially zero, given that the objects are already in memory. The call invocation is sub-millisecond.

When an object calls a method on a distributed object using Web Services, a lot more has to take place to bind the two software elements. The parameters have to be converted from their native representation into a text string and wrapped with XML tags that contain information about the data<sup>10</sup>. As noted previously, if we are calling a stateless distributed object, we probably have to include additional information in the call as compared to the requirements of a local method call. This contextual information that describes our state also has to be marshalled into text strings and wrapped with the appropriate XML metadata tags. The CPU and memory for this binding is non-trivial, but we aren't done. We now have to send our XML across the network. The Web Service on the other end has to perform the inverse operation. It has to unmarshall the textual XML information back into native data types and traverse through the call stack to find the appropriate implementation of the Web Services code. While this might not be the case for all Web Service implementations, the container environment I used had to create a new copy of the Web Service for *every invocation*. That is a non-trivial amount of overhead in itself.

Once the algorithms embodied in the Web Service have completed, the results of the service need to be marshalled, sent across the network and unmarshalled by the caller. The invocation time for a remote method call using Web Services is thousands of times longer than the invocation time for a local object method call. The network traffic goes from zero to potentially substantial. In what was most likely a worst-case scenario, a four-byte integer took over *four-thousand* bytes on the network. The bulk of the overhead was the need to pass security credentials between enterprises. That wouldn't be the case for all Web Services calls, but there is still a significant increase in the size of the network data. The use of XML for the content is significantly less network-efficient than previous distributed computing technologies such as RPC and CORBA.

When we create distributed systems, it is imperative that we implement the appropriate level of granularity between the distributed objects.

---

<sup>10</sup> termed "metadata"

## GRANULARITY

Granularity refers to the amount of work that is accomplished by a discrete piece of software. Since the overhead in a distributed method call is substantial, we want to accomplish a significant amount of actual application functionality for each call. That is, we want our Web Services to be fairly coarse-grained.

The greatest amount of reuse is accomplished by implementing a large number of small-grained object and composing them in larger grouping to form components that in turn form applications. As we compose objects in larger components, we introduce greater specificity in what the component does. That is, as the objects increase in specific functional behavior, they also decrease their ability to be reused.

We can increase reuse opportunities within our localized objects by increasing the levels of abstraction that are facilitated by object-oriented concepts such as inheritance hierarchies and the effective use of design patterns. These techniques facilitate software reuse through decoupling. These fine-grained design patterns and inheritance are not as effective with distributed objects because of the inherent need to implement coarse-grained functionality.

In effect, a move to granular Web Services is impractical because of the invocation overhead and the increase in diagnostic complexity. A Service-Oriented Architecture as the basis for *all* application development is similarly impractical.

## **RECOMMENDATION**

I've painted a fairly negative picture of Web Services and Service-Oriented Architectures. Given that this is called "The SOA Heretic", you might assume that I don't think these technologies are worthwhile for enterprises. That's not the case. There is so much positive literature on these technologies, and all I wanted to do was present the other side. For some specific business needs, Web Services are the implementation of choice. I have been involved in successful implementations of Web Services.

Web Services is the implementation of choice for inter-enterprise applications that need to exchange information in real time. One of the projects in which I was involved automated the scheduling of temporary help. The customer in need would request specific skills from multiple business partners of which we were one. When the request came in, we would search a database of matching, available skills and present them as a choice for the requesting customer. This significantly increased the productivity over the e-mail and phone based system it supplements.

From the topics in this paper, I want to make a couple of key points. The first is that those advocating a wholesale implementation of SOA and Web Services for all internal applications are misguided. There are other ways of packaging code that are just as effective for reuse opportunities, but take significantly less computer resources and are significantly less difficult to diagnose.

For inter-enterprise interaction, Web Services make a lot of sense. However, this tends to be a set of well-defined, coarsely-grained services that don't require the level of sophistication that SOA brings to the enterprise. If there is a need to create interoperability between existing systems that were developed in information silos, then the connectivity that a Service-Oriented Architecture brings may help in the exchange of information between disparate systems, although the binding layers will likely be older technologies like MQSeries rather than the newer Web Services.

The second main point is that bringing in software that implements an SOA, like an Enterprise Service Bus, is most likely not a drop-in solution to your enterprise needs. The solution is a lot of hard work in creating an IT culture that appreciates architectural issues and knows how to create loosely-coupled systems with appropriate layers of abstraction. If your IT staff isn't doing that with object technology, then there is little chance that your staff will be successful with the added complexity inherent in SOA and distributed systems. That's not only true of the programming skills, but in the infrastructure code and the staff that supports the computing infrastructure.

---

# THE SOA HERETIC

## PUTTING THINGS IN PERSPECTIVE

---

Decision-makers must avoid the “silver bullet” mentality that most vendor sales representatives pitch. As Information Technology becomes more complex with increasing demands for complexity with a shorter time-to-market, the only way to create an effective IT environment is to appreciate the complexity of architectural issues and find the appropriate funding and focus for infrastructure issues. Once that is in place, the enterprise is positioned to use new technologies such as Web Services and SOA in a manner that is appropriate for their enterprise.