Why ESB and SOA?

Version: 0.3

Date: 27/01/06

1. What is SOA?

Service Oriented Architecture (SOA) represents a popular architectural paradigm[1] for applications, with Web Services as probably the most visible way of achieving an SOA[2]. Web Services implement capabilities that are available to other applications (or even other Web Services) via industry standard network and application interfaces and protocols. SOA advocates an approach in which a software component provides its functionality as a service that can be leveraged by other software components. Components (or services) represent reusable software building blocks.

SOA allows the integration of existing systems, applications and users into a flexible architecture that can easily accommodate changing needs. Integrated design, reuse of existing IT investments and above all, industry standards are the elements needed to create a robust SOA.

As enterprises slowly emerge from the mad rush of cost reduction into a more stable period of cost management, many of them find themselves in unfamiliar territory. Prior to the economic slow down, most firms understood the options they had for IT investment. Many embarked on major package implementations (e.g., Siebel, Peoplesoft and so on), while others built on the legacy systems they have trusted for years. Either way, most firms recognized the return promised and made the investment. Today, the appetite for such large investment is gone.

However, enterprises still need to make forward progress and keep ahead of the competition. SOA (and typically Web Services as a concrete implementation of those principles) make this possible. The result is dramatic improvements in collaboration between users, applications and technology components, generating significant value for any business creating competitive advantage.

Imagine a company that has existing software from a variety of different vendors, e.g., SAP, PeopleSoft. Some of these software packages may be useful to conduct business with other companies (customers, suppliers, etc.) and therefore what the company would like to do is to take those existing systems and make them available to other companies, by exposing them as services. A service here is some software component with a stable, published interface that can be invoked by clients (other software components). So, requesting and executing services involves software components owned by one company talking to components owned by another company, i.e., *business-to-business (B2B) transactions*.

Conventional distributed system infrastructures (middleware) are not sufficient for these cross-organizational exchanges. For instance

- You would need agreement between the parties involved on the middleware platform
- There is an implicit (and sometimes explicit) lack of trust between the parties involved.
- Business data is confidential and should only to be seen by the intended recipient.
- Many assumptions of conventional middleware are invalid in cross-organizational interactions. Transactions, for instance, last longer - possibly for hours or days so conventional transaction protocols such as two phase commit are not applicable.

So, in B2B exchanges the lack of standardization across middleware platforms makes point-to-point solutions costly to realize in practice. The Internet alleviated some of these problems by providing standard interaction protocols (HTTP) and data formats (XML) but by themselves these standards are not enough to support application integration. They don't define interface definition languages, name and directory services, transaction protocols, etc,. It is the gap between what the Web provides and what application integration requires that Web services are trying to fill.

However, whilst the challenge and ultimate goal of SOA is inter-company interactions, services do not need to be accessed through the Internet. They can be made available to clients residing on a local

---

[1]The principles behind SOA have been around for many years, but Web Services have popularised it.

[2]It is possible to build non-SOA applications using Web Services, so it is wrong to assume that Web Services imply SOA.

LAN. Indeed, at this current moment in time, many Web services are being used in this context - intra-company integration rather than inter-company exchanges.

An example of how Web services can connect applications both intra-company and inter-company can be understood by considering a stand-alone inventory system. If you don't connect it to anything else, it's not as valuable as it could be. The system can track inventory, but not much more. Inventory information may have to be entered separately in the accounting and customer relationship management systems. The inventory system may be unable to automatically place orders to suppliers. The benefits of such an inventory system are diminished by high overhead costs.

However, if you connect your inventory system to your accounting system with XML, it gets more interesting. Now, whenever you buy or sell something, the implications for your inventory and your cash flow can be tracked in one step. If you go further, and connect your warehouse management system, customer ordering system, supplier ordering systems, and your shipping company with XML, suddenly that inventory management system is worth a lot. You can do end-to-end management of your business while dealing with each transaction only once, instead of once for every system it affects. A lot less work and a lot less opportunity for errors. These connections can be made easily using Web services.

Businesses are waking up to the benefits of SOA. These include:

- opening the door to new business opportunities by making it easy to connect with partners;
- saving time and money by cutting software development time and consuming a service created by others;
- increasing revenue streams by easily making your own services available.

1.1 Why SOA?

The problem space can be categorized by past IT investments in the area of eProcurement, eSourcing, Supply Chain Management, Customer Relationship Management (CRM) and Internet computing in general. All of these investments were made in a silo. Along with the incremental growth in these systems to meet short-term (tactical) requirements, the decisions made in this space hurt the long-term viability of the applications and infrastructure.

The three key drivers for implementing an SOA approach are:

1) Cost Reduction: Achieved by the ways services talk to each other. The direct cost effect is delivered through enhanced operations productivity, effective sourcing options, and a significantly enhanced ability to shift ongoing costs to a variable model.
2) Delivering IT solutions faster and smarter: A standards based approach will allow organizations to connect and share information and business processes much faster and easier than before. IT delivery productivity is markedly improved through simplification of the developer's role by providing standard frameworks and interfaces. Delivery timescales have been drastically reduced by easing the integration load of individual functionality, and applying accelerated delivery techniques within the environment.
3) Maximizing return on investment: Web Services opens the way for new business opportunities by enabling new business models. Web Services present the ability to measure value and discrete return much differently than traditional functional-benefit methods. Typical Total Cost of Ownership (TCO) models do not take into account the lifetime value generated by historical investment. This cost centric view destroys many opportunities to exploit these past investments and most enterprises end up building redundancy into their architecture, not out of necessity, but of perceived need. These same organizations focus the value proposition of their IT investment on a portfolio of applications, balanced by the overhead of infrastructure. An approach based on Web Services takes into account the lifetime contribution of legacy IT investment and promotes an evolution of these investments rather than a planned replacement.

SOA/Web Services fundamentally changes the way enterprise software is developed and deployed. SOA has evolved where new applications will not be developed using monolithic approaches, but instead become a virtualized on-demand execution model that breaks the current economic and technological bottleneck caused by traditional approaches.

Software as a service has become pervasive as a model for forward looking enterprises to streamline operations, lower cost of ownership and provides competitive differentiation in the marketplace. Web

Services offers a viable opportunity for enterprises to drive significant costs out of software acquisitions, react to rapidly changing market conditions and conduct transactions with business partners at will. Loosely coupled, standards-based architectures are one approach to distributed computing that will allow software resources available on the network to be leveraged. Applications that separate business processes, presentation rules, business rules and data access into separate loosely coupled layers will not only assist in the construction of better software but also make it more adaptable to future change.

SOA will allow for combining existing functions with new development efforts, allowing the creation of composite applications. Leveraging what works lowers the risks in software development projects. By reusing existing functions, it leads to faster deliverables and better delivery quality.

Loose coupling helps preserve the future by allowing parts to change at their own pace without the risks linked to costly migrations using monolithic approaches. SOA allows business users to focus on business problems at hand without worrying about technical constraints. For the individuals who develop solutions, SOA helps in the following manner:

- Business analysts focus on higher order responsibilities in the development lifecycle while increasing their own knowledge of the business domain.
- Separating functionality into component-based services that can be tackled by multiple teams enables parallel development.
- Quality assurance and unit testing become more efficient; errors can be detected earlier in the development lifecycle
- Development teams can deviate from initial requirements without incurring additional risk
- Components within architecture can aid in becoming reusable assets in order to avoid reinventing the wheel
- Functional decomposition of services and their underlying components with respect to the business process helps preserve the flexibility, future maintainability and eases integration efforts
- Security rules are implemented at the service level and can solve many security considerations within the enterprise

1.2 Basics of SOA

Traditional distributed computing environments have been tightly coupled in that they do not deal with a changing environment well. For instance, if an application is interacting with another application, how do they handle data types or data encoding if data types in one system change? How are incompatible data-types handled?

The service-oriented architecture (SOA) consists of three roles: requester, provider, and broker.

- *Service Provider*: A service provider allows access to services, creates a description of a service and publishes it to the service broker.
- *Service Requestor*: A service requester is responsible for discovering a service by searching through the service descriptions given by the service broker. A requester is also responsible for binding to services provided by the service provider.
- *Service Broker*: A service broker hosts a registry of service descriptions. It is responsible for linking a requestor to a service provider.

1.3 Advantages of SOA

SOA provide several significant benefits for distributed enterprise systems. Some of the most notable benefits include: interoperability, efficiency, and standardization. We will briefly explore each of these in this section.

1.3.1 Interoperability

Interoperability is the ability of software on different systems to communicate by sharing data and functionality. SOA/Web Services are as much about interoperability as they are about the Web and Internet scale computing. Most companies will have numerous business partners throughout the life of the company. Instead of writing a new addition to your applications every time you gain a new partner, you can write one interface using Web service technologies like SOAP. So now your partners can

dynamically find the services they need using UDDI and bind to them using SOAP. You can also extend the interoperability of your systems by implementing Web services within your corporate intranet. With the addition of Web services to your intranet systems and to your extranet, you can reduce the cost integration, increase communication and increase your customer base.

It is also important to note that the industry has even established the Web Services Interoperability Organization.

"The Web Services Interoperability Organization is an open industry effort chartered to promote Web Services interoperability across platforms, applications, and programming languages. The organization brings together a diverse community of Web services leaders to respond to customer needs by providing guidance, recommended practices, and supporting resources for developing interoperable Web services." (www.ws-i.org)

The WS-I will actually determine whether a Web service conforms to WS-I standards as well as industry standards. In order to establish integrity and acceptance, companies will seek to build their Web services in compliance with the WS-I standards.

1.3.2 Efficiency

SOA will enable you to reuse your existing applications. Instead of creating totally new applications, you can create them using various combinations of services exposed by your existing applications. Developers can be more efficient because they can focus on learning industry standard technology. They will not have to spend a lot of time learning every new technology that arises. For a manager this means a reduction in the cost of buying new software and having to hire new developers with new skill sets. This approach will allow developers to meet changing business requirements and reduce the length of development cycles for projects. Overall, SOA provides for an increase in efficiency by allowing applications to be reused, decreasing the learning curve for developers and speeding up the total development process.

1.3.3 Standardization

For something to be a true standard, it must be accepted and used by the majority of the industry. One vendor or small group of vendors must not control the evolution of the technology or specification. Most if not all of the industry leaders are involved in the development of Web service specifications. Almost all businesses use the Internet and World Wide Web in one form or another. The underlying protocol for the WWW is of course HTTP. The foundation of Web services is built upon HTTP and XML. Although SOA does not mandate a particular implementation framework, interoperability is important and SOAP is one of the few protocols that all good SOA implementations can agree on.

2. What is the right paradigm for SOA?

It is often said that Web Services are reinventing the wheel of distributed systems without learning from the past. Web Services started off simple, with SOAP and WSDL and have rapidly evolved in complexity with a range of specifications and standards being developed across the industry. For many observers, there appears to be little difference between Web Services and previous distributed systems incarnations such as CORBA, J2EE and DCOM. However, that is an unfair and inaccurate comparison, based on a lack of understanding of the roots of Web Services and the hype surrounding them.

Whilst it is true that there is very little new technology in Web Services, it is worth understanding what has given this approach the potential for a level of success unheard of in over twenty years of wide spread use of distributed systems:

- XML
- wide spread adoption of the technology, based in no small way on the fact that it leverages the Web (HTTP).

However, this does not take into account the architectural difference between SOA and previous distributed system architectures. To illustrate this, consider that prior to Web Services and the wider-spread adoption of SOA that it brought, probably the most success distributed system architecture was based on *distributed objects* (e.g., CORBA and DCE/DCOM). Therefore, if it were merely a case of taking XML, HTTP and adding distributed objects, why did technologies such as WebObjects and W3Objects not have the same level of success? There are several reasons for this, including the fact that SOA encourages loosely coupled applications by further separating service implementation from

their interfaces, the separation of contract definitions from service endpoints, heterogeneity of implementations and lack of overall control within the network.

Web Services, which can be considered a concrete representation of SOA, are about interoperability *and* internet-scale applications. So, what do we mean by internet scale applications and why did previous architectures not work well in solving these problems? There are several definitions of what constitutes an internet-scale application, but there are some common requirements:

1) they should scale from several to hundreds and thousands of participants/services.
2) they should be loosely coupled, so that changes of service implementation at either end of an interaction can occur in relative isolation without breaking the system.
3) they need to be highly available.
4) they need to be able to cope with interactions that span the globe and have connectivity characteristics like the traditional Web (i.e., poor).
5) asynchronous (request-request) invocations should be as natural as synchronous request-response.

Scalability and availability are possible with other technologies, such as CORBA. Although 2) and 4) can certainly be catered for in those technologies as well, the default paradigm is one based on an implementation choice: objects. Objects have well defined interfaces and although they can change, the languages used to implement them typically place restrictions on the type of changes that can occur. Although it is true that certain implementations, such as CORBA, allow for a loosely coupled, weakly typed interaction pattern (e.g., DII/DSI in the case of CORBA), this is not typically the way in which applications are constructed and hence tool support in this area is poor. This tends to drive developers back to the closely coupled style of application development.

So what is the best paradigm in which to consider an SOA that allows it to address all 5 points above?

2.1 Messages versus events?

One way of thinking about SOAs is that they are only about sending and receiving messages (documents) of arbitrary types. Everything else, such as the interpretation of those documents, is hidden behind the service endpoint and is hence implementation specific. Although this is a technically accurate description, the problem with this is that at some level *all* distributed systems work this way. For example, in the case of CORBA, the message/document is encoded in an IIOP packet and there are well defined ways for that document to be interpreted. Therefore, from an abstract architectural level it would seem like CORBA is an appropriate SOA infrastructure.

The SOA purist says that it is inherently impossible for a CORBA user (or a user of any distributed OO system) to ignore the fundamental implementation choices that are made for them by the architecture: everything is based on objects and methods. In SOA those choices are not architecture but implementation. The problem, however, is that this is an arbitrary distinction; someone else could draw the line at a different level in the stack and come up with different conclusions. Precisely because no new languages have been developed to construct SOA applications, ultimately they will all use objects and method invocations and an architecture built purely on messages has to eventually say something concerning how those messages are mapped to higher-level business logic.

Therefore, no architecture for a distributed system can be complete if it simply works in terms of the message. However, in order to address the 5 points mentioned earlier and maintain the benefits of SOA, it is appropriate to break down the *global architecture* into two related components:

- *The message architecture*: this is concerned solely with the message interaction pattern and although it acknowledges higher-level business processes, these are left to the other architectural component.
- *The implementation architecture*: this describes how messages are mapped onto physical components and objects and is not concerned with how those messages are conveyed. A suitable concrete form of this component can still be responsible for maintaining SOA principles, such as loose coupling.

Furthermore, fundamentally SOA is not about message exchange patterns based on request-response, request-request, asynchrony etc. but about *events*. The fundamental SOA is a unitary event bus which is triggered by receipt of a message: a service registers with this bus to be informed when messages arrive. Next up the chain is a demultiplexing event handler (dispatcher), which allows for sub-services

(sub-components) to register for sub-documents (sub-messages) that may be logically or physically embedded in the initially received message. This is an entirely recursive architecture that abstracts away from implementation details, allowing large-scale SOAs to be designed without recourse to a specific implementation.

2.2 The right level of abstraction

There is message passing in any distributed system. CORBA messages are IIOP encoded, for example. DCOM has its own implementation choices. But underneath they are all based on the flow of bytes (ultimately even bits) across the wire. However, these bytes are received by TCP/IP and ultimately reformed into a packet that is then given to a server stub that knows about the object that is being invoked. In the 1980's with Sun RPC, the same thing happened, but the stub (dispatcher) knew about registered procedures. The important thing is that the user (developer) of applications didn't see any of this, but was presented with the ability to explicitly register procedures (as in Sun RPC) or objects and hence implicitly a range of methods (as in CORBA). They don't see what happens under the covers and they don't need to. In some ways, Sun RPC was a very early ESB.

Recursively, the implementer of the stub generation code (in Sun) or the ultimate dispatcher (in CORBA) knew about the higher levels, but was working in terms of packets received from TCP/IP. Their architecture really was based on constrained messages. But because of the higher level requirements, some of this was also constrained by that architecture (e.g., objects and method invocations). But what this meant is that this code in the case of distributed objects, is tailored specifically for the end receiving object: change that object interface and the dispatching code has to be changed.

Now, let's look at how to make this flexible so that the dispatching code doesn't need to be changed if the end receiver does. This has precedent in the CORBA world, with DSI/DII at the low level, and something like the Activity Service (also in J2EE) at the higher level. The way that is accomplished is via an interface to ultimate receivers that does not expose implementation details and is entirely abstract. If we look at the Activity Service, the participants are all implementations of the CORBA Action interface that has a single method, processSignal. The parameter to processSignal is a Signal, which is essentially a CORBA any: anything can be encoded within it of arbitrary complexity or simplicity. The Action participants can change without affecting the sender code directly.

But how does this affect the ultimate application? Since it is working in terms of received Signals which have any information encoded within them, it is now very similar to the original low-level TCP/IP receiver/dispatcher code: although the low-level infrastructure does not change if the Action implementations change, the application developer must become responsible for encoding and decoding messages received and acting on them in the same way as before, i.e., dispatching to methods, procedures or whatever based on the content. At the low-level, messages (Signals) can carry any data, but higher up the stack the application developer is constraining the messages by imposing syntactic and semantic meaning on them (based on the contract that exists between sender and receiver). Therefore, at the developer's level, changes to the implementation (the contract, the object implementation etc.) *do* affect the developer again: this can never be avoided. The message-driven pattern simply moves the level affected by change up the stack, closer to the developer.

So does this give us anything then? If the developer (at either sender or receiver) makes changes at the right level, these changes will affect the corresponding level(s) at receiver or sender. The answer is that yes, it does help because it allows us to separate out different types of developer and different types of contract. The level of restriction is right for the application and not imposed by the architecture. We shouldn't be restricting the things that can be done at the lower level by forcing architectural/implementation choices down the stack.

At the moment we are concentrating on the low level details of SOA: the message passing level. This is because it is here that the simplicity and utility of SOA arise. XML and SOA allow for loose coupling. Nothing should be added at this level that will restrict what can be done at the higher level. If someone wants to implement objects and method invocation over Web Services then that is a higher level choice and shouldn't impact the developer who wants to interact with a pure messaging service (e.g., a JMS). For example, there should be nothing explicit about object methods (op codes, interface definitions, inheritance relationships etc.) at the low level message passing/receiving level: those can be encoded within the message. To do otherwise would be the same as CORBA requiring the format of all lower-level TCP/IP messages to have some aspects of the IIOP packet structure on them. The level of TCP/IP

is obviously not the right place for this to go because it impacts other things. Imagine if TCP/IP worked not in terms of host/port pairs but in terms of host/port/object key/method op code/interface name etc. Now email *could* be implemented to conform, and so could NNTP, FTP, gopher etc. But that is not a natural fit.

The fact is that there are different users at different levels with different contracts and expectations. All that SOA/WS are saying is that we need to give the flexibility in our next generation distribution infrastructures to make all of these levels available without impacting on each other, i.e., one level should logically build on another, but lower levels shouldn't need to know about higher levels and likewise for users. The right level of abstraction must be maintained between users and levels. For example, ESB and JMS are all about message passing at one level, but at a higher level users still work in terms of objects; it's just that the lower level message passing doesn't have these higher level object semantics imposed on it.

As we've seen, the implementation details of SOA/WS are secondary to the conceptual layered model/architecture. For example, one "large" object with a generic doWork operation that works in terms of abstract messages can be used to achieve these goals; that was the basis of the CORBA Activity Service after all.