

# Demystifying the ESB – What’s real and what’s marketecture?

Ross Altman, CTO

SeeBeyond Technology Corp.

Like pretty much everything that gets caught up in the IT hype cycle, ESBs have taken on mythic capabilities. And, like pretty much everything that becomes the focus of myth, there’s a kernel of truth embedded within all those the superhuman powers.

To begin the process of separating fact from fiction, let’s start with a workable definition of an ESB.

*“An **Enterprise Service Bus** is an integration platform that utilizes Web service standards to support a wide variety of communications patterns over multiple transport protocols and deliver value-added capabilities.”*

Now, let’s pick that apart a bit so we can better understand just what an ESB can really do:

1. An ESB is an integration platform . . .

While an ESB can be used to build a standalone application “from scratch”, it’s not ideal for that purpose. The main competition here is the various component-based application runtime platforms such as the J2EE application servers and the Microsoft .Net framework. These application runtime platforms are designed specifically to support development of new, modular applications where all those components run within the same environment. Also, and perhaps more tellingly, J2EE and .Net applications tend to be more tightly-coupled – meaning that each component is written with specific knowledge of the data and process semantics of the other components in the application.

In contrast to that, ESBs are designed to support the development of composite applications – applications that involve some new code “on top of” lots of existing code from multiple previously-built systems. Unlike the more tightly-coupled architecture of the typical J2EE or .Net application, solutions built on an ESB are best implemented within a loosely-coupled architecture.

2. . . . that utilizes web services . . .

The “service” in Enterprise Service Bus refers to the architectural concept in which a component offers to execute functions on behalf of any other program that calls the component’s published API. There is a presumption in this pattern that the called program may change its internal processing as long as it honors the API – i.e., as long as the API doesn’t change and as long as the visible results obtained through invocation of that API don’t change.

Given that usage, it’s clear that, at least conceptually, an ESB could use any protocol as its normative API. However, as a practical matter, the term ESB has become inextricably

intertwined with Web services. As a result, in the real world of product marketing and vendor-delivered functionality, an ESB uses Web services to deliver its component-to-component connectivity.

3. . . . to support a variety of communications patterns . . .

Saying that an ESB supports Web services protocols, doesn't limit the communications patterns that can be delivered by an ESB. While the predominant Web services-based application pattern is built around synchronous request/reply interaction between service providers and service consumers, an ESB can and must support a much broader variety of one-way and two-way, synchronous and asynchronous program invocation models.

4. . . . over multiple transport protocols . . .

At first glance, it might look like we've got mutually exclusive requirements here. Many would immediately assume that an ESB that "utilizes Web services" couldn't also support communications "over multiple transport protocols". However, a little digging into the most fundamental Web services standards shows that Web services can be implemented on top of a variety of transport protocols. The most common implementation by far is SOAP over HTTP. But, SOAP can also be implemented over SMTP or MQSeries or JMS or . . . well, you get the picture.

5. . . . and deliver value-added capabilities.

We're pretty far along in the process of building a complete array of standards to support enterprise services, however, we're still not very close to the pot of gold at the end of that rainbow. In short, it will be many years before we have a complete set of mature and widely implemented Web services standards that enable robust, secure, performant and highly available communications. In the meantime, developers will look to the ESB vendors for the value-added functionality that the available standards haven't yet addressed.

Now that we've defined an ESB, let's clarify its use. While an ESB can certainly be used to create a brand new, standalone, component-based application, an application server is probably better suited for that sort of project. But, for ESB fans, that's OK, as there are relatively few brand new, standalone, component-based applications in the development queue in most enterprises.

To the contrary, most applications that are being built today are composite applications, consisting of a combination of a relatively small amount of new functionality along with a large amount of existing application logic and data. And, that design target is the sweet spot for ESB use.

So, what's required to build and deploy an ESB? Well, it all starts with the communications infrastructure that we dealt with in our definition – an integration platform that utilizes Web

services to support a variety of communications patterns over multiple transport protocols while delivering value-added capabilities.

In reality, the ESB products in the market provide this to varying degrees. Some only support basic communications capabilities – enabling simple request/reply interaction with basic connection-level security. Other products enable both synchronous request/reply as well as asynchronous send/receive and some also add message-level security along with transaction management, logging and even billing capabilities.

But, regardless of the communications functionality provided by the ESB, that's far short of meeting the complete set of requirements for developing and delivering a composite application. At a minimum, you need three layers of technology to address your integration needs and the communications capabilities that we've just described only provide the middle layer in that model.

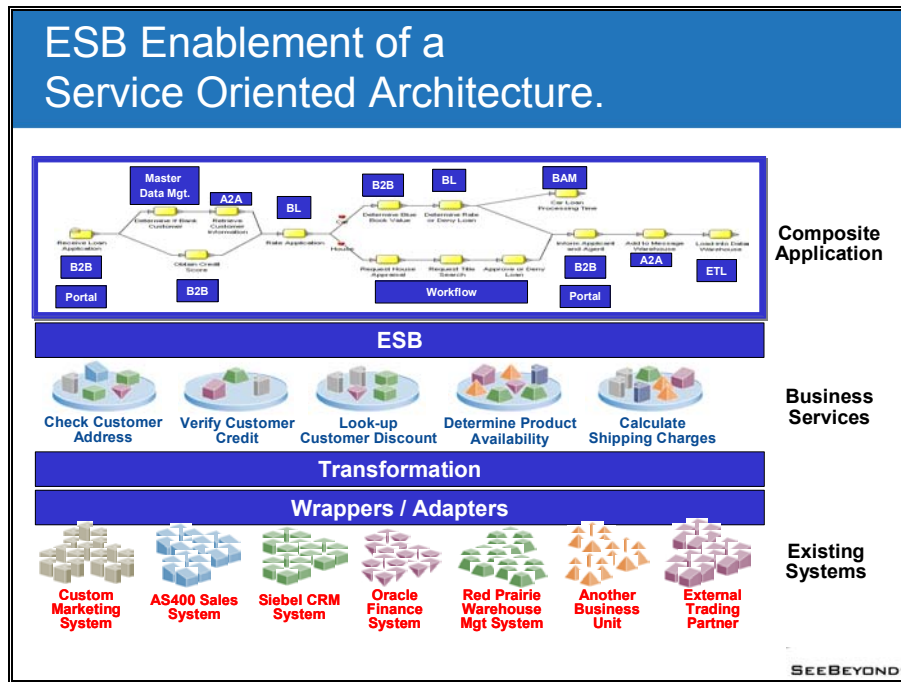


Figure 1: An ESB-based Service Oriented Architecture

The bottom layer in our three-tier model provides the ability to link ESB communications to the wide variety of applications and data that need to be leveraged in the typical composite application. This layer includes tools or components that are sometimes referred to as adapters or wrappers, but that only describes part of what's needed here. You start with the ability to wrap or adapt a legacy program or data API to present a Web services interface, but you also need to be able to readily create and maintain the data transformation logic that's required to merge data from multiple applications into one new application.

You also have to be able to create APIs where none exist today – i.e., you need to be able to screen scrape those legacy programs that were never designed to be called by another program.

In the end, the functions needed at this bottom layer of our model can be met by products that have been traditionally referred to as EAI tools or integration brokers. There are also vendors who specialize in providing full function adapters (i.e., adapters that include transformation capabilities) that can be used with a range of integration tools from other companies. Regardless of where they come from, you can't really build an ESB-based application without these adapters, as there just aren't many legacy and packaged applications in the typical enterprise that natively supported Web services interfaces.

In addition to this adapter layer, we need a layer in our model that sits on top of the ESB's communications functions. Once data flows from your legacy and packaged applications through adapters and transformation logic and into your ESB, you've created a basic Service Oriented Architecture. But, that's a bit like having a Ferrari on cinder blocks. To take your SOA on the road, you need an application layer, including such capabilities as Business Process Management, Business Activity Monitoring, workflow, B2B connectivity, Graphical User Interface functionality, portal-based authentication, authorization and personalization and master data management. In short, you need a comprehensive application development toolkit on top of your ESB and adapters/transformation tools.

Now, you could get an ESB from one vendor (i.e., an "ESB-only" vendor) and these development tools from various other vendors. Or, you can get all this stuff from a single vendor (i.e., an "ESB ecosystem" vendor). Which works out best for you?

The ESB ecosystem vendors have several short and long term advantages over the ESB-only vendors.

1. You can buy everything from one vendor, which generally results in a shorter and simpler product evaluation and acquisition process, lower license and maintenance costs and "one throat to choke" when issues come up around support, professional services and customer care.
2. Basic ESB functionality is going to become commoditized, which will endanger the long-term viability of ESB-only vendors, who will find it increasingly difficult to differentiate their products.
3. As the Web services standards mature, both the perceived and the real value of an ESB will erode.

These advantages for the ESB ecosystem vendors call into question the long term viability of the ESB-only vendors. They will either have to become ESB ecosystem vendors or sell out to one.

By the way, there are also issues that have to be evaluated when buying a solution from an ESB ecosystem vendor. If the vendor has simply pulled together the components of the ecosystem through the "late binding" of separately developed development tools or (more commonly)

through merger and acquisition activity, then there will be no synergy in the product suite except, perhaps, the “one throat to choke” benefits previously noted.

On the other hand, if the various components and tools of the ESB ecosystem are actually integrated, then there are further benefits that can be realized. For example, a fully integrated set of ESB-based tools could share a single repository for development and a single runtime environment for deployment. This would allow developers to better implement reuse strategies by making it easier to find previously built components. Also, it would be easier to deploy and manage the solution at runtime if there were a single runtime console, a single debugger, a single runtime management environment, a single security framework and a single transaction management engine.

Given all the hype that has surrounded the ESB marketplace, it's understandable that some of these issues have been lost in the turmoil. But, in the end, myths have nothing to do with getting your applications into the hands of users. It's the performance of your tools and platform software that's critical.