Enterprise Service Bus Technology for Real-World Solutions

Meeting the Challenge of Business Integration

IT executives today routinely rank business integration as their number one priority. While it was sufficient in the past for IT initiatives to focus on individual business functions, the challenge now is to *integrate* those islands of automation across the enterprise. Maximizing operational efficiency in end-to-end business processes, critical to lowering costs, is only one of the reasons. Business integration is also necessary for the business to become more *agile*, able to respond more nimbly to changes in the external environment, launch new products and services more quickly, or incorporate new supply chain or distribution partners more easily. In fact, e-business transformation demands business integration, allowing IT systems to work in concert and respond in real-time.

Service Oriented Architecture (SOA) is the software industry's answer to this challenge, and today most IT shops have accepted SOA as the foundation for their next generation of enterprise software. While SOA is frequently equated to SOAP-based *web services*, it is actually a more general concept. SOA is best described as an application architecture in which individual functions are encapsulated software components exposed externally through a universal implementation-neutral interface. Software functions deployed in this way are called *services*. A hallmark of SOA is that service requests are *transparent*, meaning independent of the service's technical implementation, connection method or protocol used to request it, or even whether it is local or remote to the requester.

SOA allows complex enterprise applications and end-to-end business processes to be *composed* from these services, even when the providers of those services are applications hosted on disparate operating system platforms, written in different programming languages, or based on separate data models. This flexible composition supports the fundamental goals of business integration, which are linking business systems across the enterprise and extending business services to customers and trading partners.

The ability to expose business functions through a service-oriented interface is only half the story. You also need a universal mechanism to *interconnect* all the services required in the composed business solution – without compromising security, reliability, or performance scalability. In SOA, the enterprise infrastructure layer interconnecting service requesters and providers is known as the *Enterprise Service Bus* (ESB). Like SOA itself, ESB is sometimes associated with a particular set of Java and web services standards, but it is more properly viewed as an *integration pattern* or architecture that can accommodate a variety of implementations consistent with the broader definition of SOA.

The purpose of this report is to describe the essential elements of an Enterprise Service Bus in that broader context, and the resulting benefits for business and IT.

What Is an Enterprise Service Bus?

Confusion in the Marketplace

It is useful to acknowledge that ESB has been defined in different ways by various industry analysts and software suppliers. Gartner Group[1] tends to describe it in terms of a low-cost, lightweight alternative to traditional integration middleware. IDC,[2] on the other hand, calls it the open, standards-based connectivity backbone of the future. ZapThink[3] describes it simply as a message bus with service-oriented interfaces.

A number of startup vendors providing lightweight connectivity software based on web services and Java messaging standards have adopted the term ESB as a way to distinguish their offerings from established "heavyweight" messaging middleware. On the other hand, integration middleware leaders such as IBM have shown how their widely installed message bus infrastructure has been used in large-scale SOA implementations and exemplifies the ESB integration pattern. Meta Group[4] reinforces this view by acknowledging that traditional EAI infrastructure may be used to implement SOA, as long as it provides the required universal, transparent connectivity.

A Practical Definition

While descriptors like "low-cost" and "lightweight" are sometimes used to differentiate the offerings of ESB startups from established message bus providers, these attributes are tangential to the real purpose of ESB, which is simply to *provide the connectivity infrastructure required for enterprise-scale business integration using SOA*. In order to deliver the level of reliability, performance, and functionality they need, business-critical SOA solutions based on those lightweight ESB offerings will ultimately have to add that weight and cost somewhere else.

You can deploy ESB as an island, but it has more value when it integrates existing business systems and infrastructure. Unfortunately, most of the software assets than run the business today are not service-oriented, or even standards-based. Moreover, web services standards for critical features like quality of service are still being defined. Thus a practical definition of ESB ought to focus on attributes and capabilities that support SOA today in the real world. Such a definition does not limit ESB to a single product category, but describes it more broadly as an architectural layer or integration pattern, focusing attention on its four essential capabilities.

1. **Universal connectivity of services via XML messaging,** interconnecting requesters and providers across diverse platforms and data models, providing a common backbone for requests, messages, and events.
2. **Vendor-independent communications standards**, such as SOAP and Java Messaging Service (JMS).
3. **Quality of service features**, including reliable delivery, transaction management, and scalable performance.
4. **Service mediation features**, providing loose coupling between requesters and providers.

---

[1] Gartner Group, "Hype Cycle for Application Integration and Platform Middleware," May 2003
[2] IDC, "The Enterprise Service Bus: Disruptive Technology for Software Infrastructure Solutions," March 2003
[3] ZapThink, "What is the Shape of a Service-Oriented Architecture?" August 2003
[4] Meta Group, "Practical Approaches to Service-Oriented Architecture," November 2003

Such a definition embraces both "heavyweight" messaging middleware and "lightweight" startup offerings, and recognizes that both web services startups and established EAI middleware vendors are ultimately aiming for the same goal. The appropriate ESB software for the job is a function of the application requirements.

Evolution of ESB

From this perspective, ESB can be seen not just an outgrowth of web services, but as an evolution of enterprise messaging and message broker technology:

- **Enterprise messaging** provides universal connectivity and quality of service, with high-speed asynchronous any-to-any communications between application systems – even those that are not service-oriented or standards-based. Its quality of service features such as multihop store-and-forward, reliable once-and-only-once delivery, transaction management and recovery, and scalability to high message volumes are key ESB requirements.
- **Message brokers** provide the service mediation required by ESB, with features such as publish-subscribe integration, content-based routing, and message transformation. Message brokers make the integration of large numbers of resources practical by reducing the number of point-to-point connections. More importantly, mediation is the key to *loosely coupled integration*, in which the sender of a message (service requester) does not need to know who the receiver (service provider) is. As message broker functionality migrates into application servers, those components become a key piece of the ESB puzzle as well.
- **Web services** provide a platform-independent framework for SOA based on open standards such as XML, WSDL, and SOAP, most often on standard Internet transports such as HTTP.

To understand what distinguishes ESB from traditional EAI architectures, we first need to take a closer look at SOA itself.

The Importance of Service Oriented Architecture

SOA is a new distributed application architecture or design pattern that specifically addresses the challenges of end-to-end integration in the e-business era. SOA is the key technology enabler of a new agile business model that allows companies to respond quickly to changing customer requirements, new business opportunities, and emerging competitive threats. SOA enables flexible integration and reuse of existing information assets. It reduces operating costs by automating information flows across disparate business systems. It supports e-business transformation by linking enterprises with the business processes and systems of trading partners. And it enables the end-to-end performance management required for competitive advantage.

Interoperability, Composition, and Reuse

SOA's fundamental principles are flexible interoperability, composition, and reuse of software assets. A *service* in SOA is a unit of work performed by a service provider to achieve a promised end result for a service consumer. It is a business function encapsulated as a software component and exposed for flexible reuse in multiple business processes and applications. In SOA, processes and applications are *composed* – or in the new vocabulary, *choreographed* – from individual services. The location and technical implementation of the function by the service provider is invisible or *transparent* to the requester.

While a service has the appearance of a self-contained function from the service requester's perspective, the provider's implementation may actually be a multi-step process involving multiple systems or even crossing enterprise boundaries. Implementation transparency makes SOA fundamentally different from distributed component architectures of the past, such as object-oriented architectures, in which the structure of the function request (method call) is bound to the component's technical implementation (CORBA, Java, etc.).

Location and implementation transparency are examples of SOA's *loose coupling principle*. Loose coupling is essential to *interoperability* among diverse and changing platforms, object models, and programming languages. SOA provides loose coupling by requiring service interfaces to be *universal* and *descriptive*, based on implementation-neutral formats and protocols rather than APIs. Loose coupling allows either the requester or provider's software to *change* without impacting the other, provided that the descriptive interface remains the same.

SOA and Web Services

Broadly speaking, a software solution can be called "service-oriented" if it supports:

1. **Composition** of business functions (services) provided by encapsulated software components exposed through platform-neutral interfaces.
2. **Universal, standards-based connectivity.** In concrete terms, that means that service interfaces in SOA are based on *XML messages*.
3. **Loose coupling,** in which the technical implementation and location of a service are transparent to the requester.

While they are by no means the only form of SOA, *web services* are certainly the most familiar. Web services represent SOA based on a specific set of Internet standards, which impose two additional constraints on SOA:

- Service request and response messages are delivered over standard Internet transports, such as HTTP, via a specific protocol, SOAP.
- XML message content defined by a particular interface definition language, WSDL. Service interfaces based on the Web Services Description Language (WSDL) describe the service operations, schemas of request and response messages, and a URL that will accept service requests, but do not reference details of the service's technical implementation.

Some vendors of lightweight ESB equate SOA to web services, and assume SOAP and WSDL in their connectivity architecture. But SOA is much broader than web services, and practical realities require ESB to go beyond web services standards. In fact, an effective baseline description of an ESB might be *any communications infrastructure capable of requesting and delivering services via XML messages*.

ESB in Theory and Practice

In SOA, ESB serves as the connectivity fabric linking service requesters and providers. It adds *quality of service* (QoS) features, such as security and reliable delivery, and *mediation services*, such as message routing and data transformation, to an enterprise messaging backbone. First-generation ESB software has been able to provide these features in a "lightweight" and "low-cost" form by leveraging industry standards and the built-in capabilities of J2EE applications servers, notably JMS and web services.

Limitations of "Lightweight" ESB

ESB vendors proudly claim to have architected their software from the ground up take advantage of the new standards, in contrast to established integration middleware, which they position as "proprietary" and "out-of-date." Nevertheless, actual deployment of business solutions based on this software has been limited to relatively small "greenfield" applications built from web services and JMS-aware components, not large-scale business-critical applications that reuse and compose existing software assets. There are several good reasons for this:

1. Existing resources not standards-based or decomposable

One reason is fairly obvious. The critical applications that currently run the business are neither standards-based nor WSDL- or JMS-aware. Many are not even decomposable into independent services without a complete rewrite. This is not a flaw in the SOA vision. It's just the reality. Moving to SOA is by necessity an *incremental process*.

2. Immature standards limit interoperability

Less widely understood is the fact that the standards underpinning these "lightweight" ESB offerings are themselves immature and do not yet provide true interoperability.

- **Web services** support is native to application servers, but incompatibilities still divide vendor implementations. True web services interoperability is still being hammered out in standards committees, including WS-ReliableMessaging and WS-Trust, among others.
- **JMS** as a standard has focused on the API. However, many aspects of the underlying messaging service implementation, such as the wire protocol, vary from vendor to vendor. As a result, JMS implementations from different vendors typically do not interoperate.
- **J2EE Connection Architecture (JCA)** provides a standard framework for pluggable resource adapters that can invoke APIs of popular packaged applications. But it, too, is limited. JCA 1.0, for example, does not support requests or events *from* packaged applications, only requests *to* those applications, and does not standardize the metadata required to *introspect* available operations in order to configure service requests. These advanced functions are provided in vendor-proprietary extensions.

Even the underlying messaging backbone of the ESB may be limited in its interoperability. For example, it may not support the operating system or programming language required to integrate some existing IT assets needed in the end-to-end solution.

3. Performance limitations

A third limitation is performance. For example, while lightweight ESB vendors tout "wrapper" tools as a way to provide service-oriented interfaces for any application or middleware that exposes APIs, wrappered interfaces may degrade performance. Also, layering QoS features like reliable delivery on top of standards-based communications can limit performance scalability.

4. Functional limitations

In many ESB offerings, "low cost" is directly related to limited functionality. For example, mainstream business integration middleware suppliers have added *business process management* – tools to compose and execute end-to-end processes based on service

choreography – and *performance management* capabilities – complementary tools to monitor service levels and optimize key business metrics. Lightweight ESB vendors shun such capabilities as bloat and overkill, and for greenfield proof-of-concept applications they probably are. But for SOA designed to really run the business, they certainly are not.

ESB for "Real World" SOA

The point is not that ESB is a bad idea, but that it is defined by the connectivity requirements of "real world" SOA. By necessity, SOA adoption – in the business-critical applications where it really counts – is occurring incrementally. Refactoring, wrappering, or replacing legacy applications with new standards-aware equivalents is going to be a slow process. The web services and Java standards that promise out-of-the-box interoperability are still being tweaked. Thus lightweight ESB technology that simply *assumes* interoperable WSDL-based business functions is going to see limited use in mission-critical solutions today.

Nevertheless, as we have seen, service-oriented solutions can be built today from existing IT assets by applying the principles of composition, XML message-based interfaces, and loose coupling. These solutions represent "real world" SOA. The Enterprise Service Bus they use may not be the lightweight, standards-based variety, but an integration pattern leveraging established messaging middleware such as IBM MQ, IBM WebSphere Business Integration Message Broker, and associated integration adapters. Here are three examples:

Customer Example: Standard Life

Standard Life,[5] based in the UK, is one of the world's largest mutual financial services companies. The majority of its business is conducted through Independent Financial Advisors (IFAs). In the late 1990s, the need to lower operating costs while improving service to IFAs and customers was straining Standard Life's IT infrastructure, primarily mainframes running COBOL-based IMS and DB2 applications. For example, IFAs needed to be able to compare prices across various products and aggregate all of a customer's holdings in a single view.

In the new integration architecture, flexibility and reuse would be critical. According to Standard Life's core technology design manager, "We needed to maintain independence of underlying infrastructure, allowing us to change the infrastructure without rewriting the applications." The key elements of the new architecture were determined to be an intelligent messaging hub linking application services, XML as the common language of integration messages, and Java as the platform for new application services running on a variety of operating systems. While it was radical at the time, today we recognize Standard Life's new architecture as an early example of the ESB integration pattern.

Proposals from IBM, Microsoft, and Oracle all included a messaging middleware, message brokers, and application server technology, but Standard Life chose IBM largely because of the proven performance scalability of its WebSphere MQ messaging backbone and brokers at large financial institutions. Figure 1 illustrates the resulting architecture.

The WebSphere Business Integration Message Broker (formerly known as MQSeries Integrator) and messaging backbone interconnects WebSphere Application Server, IMS mainframes, other application servers, and web portals, all through XML messages. Message-driven beans on WebSphere Application Server interface the messages to Java. On the mainframe, Standard Life created its own messaging interface, called Core Systems Access, which exposes IMS and CICS transactions to MQ messages.

[5] Case study, D. Marshak, Patricia Seybold Group, www.psgroup.com

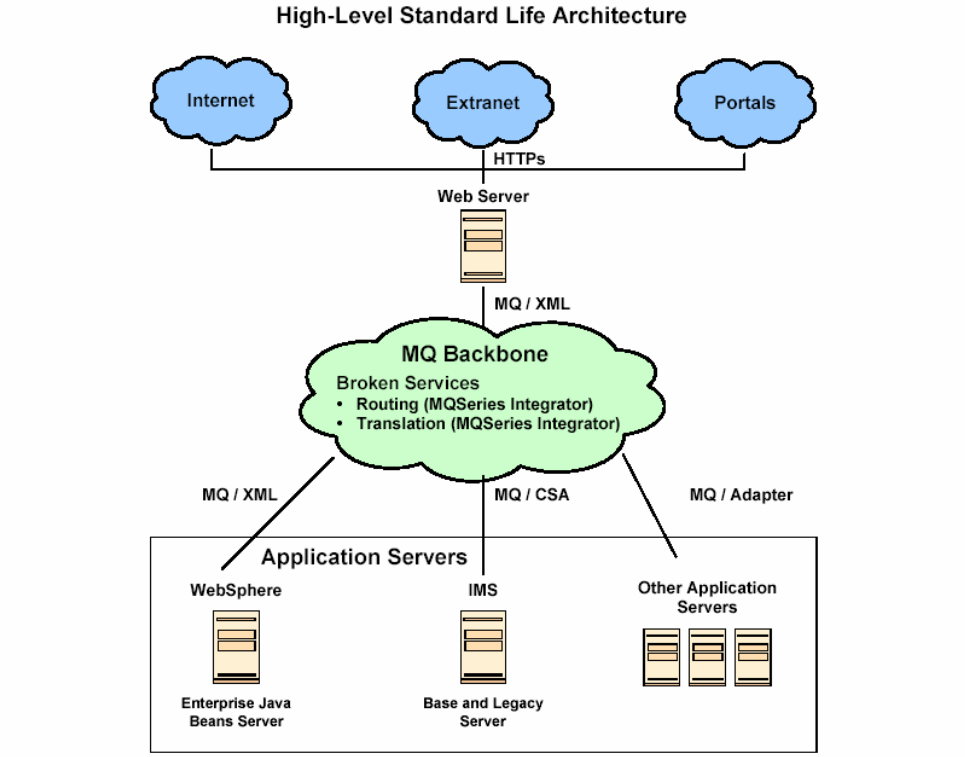

Figure 1. Standard Life's ESB uses XML messaging over MQ. Source: Patricia Seybold Group

The new integration architecture became the required platform for all new e-business projects at Standard Life. Less than a year after initial deployment of the platform, the company had twelve applications running on the IFA extranet, including commissions, quotes, individual client and policy service, group service, and group new business. Now IFAs, while talking to customers, can go to a single site and get quotes from multiple financial institutions and aggregate them in real time. IFAs can also submit new business requests to Standard Life as XML messages.

As a result of their pioneering ESB, Standard Life has been able to reduce costs, while making themselves more attractive to IFAs through superior business integration and customer service. Within the IT shop, the major benefit of a common business integration platform has been savings in resource planning and allocation. According to the development manager, "We are definitely seeing improved speed to market – directly attributable to the new architecture and the way we are designing the applications."

Customer Example: Charles Schwab

Charles Schwab[6] invented the full-service discount brokerage model thirty years ago, and today handles over 8 million active accounts and almost $800 billion in customer assets through 422 US offices, four regional call centers, and automated web and telephonic channels. As internet trading has eroded commissions, Schwab has evolved toward a business model based on trusted relationships with customers, focusing on accurate independent advice for investors. Advice to customers is provided through a hierarchy of channels, ranging from web self-service to face-to-face interaction with a Schwab investment consultant.

[6] Case study, D. Marshak, Patricia Seybold Group, www.psgroup.com

Schwab's IT architecture is shaped by three primary business goals:

- Consistency of client experience across all channels
- Responsiveness to new business needs by adding new customer types and channels without affecting back-end services, and adding new services without affecting channels
- Efficiency through maximum reuse of existing infrastructure, new development, and IT resources

Like other large financial services companies, Schwab is a huge COBOL/CICS mainframe shop. However, the need for consistency, responsiveness, and efficiency pushed Schwab to turn its hierarchical mainframe environment into a peer-to-peer service-oriented architecture. The first step was to take the backend applications and make them channel-independent services, beginning years ago with the introduction of voice response units in the call center. Step two was to create a middleware bridge between the channels and the channel-independent services. The third step, ongoing today, is to evolve that middleware away from tightly-coupled proprietary interfaces to loosely-coupled, standards-based interactions – an Enterprise Service Bus.

In Schwab's bus, backend services are exposed to channels as web services, i.e., with WSDL interfaces. According to the company's VP of Engineering in charge of the project, "WSDL is very important to enable different invocations of a service, and the Bus is essentially an enabling technology that supports these multiple services and transports."

Like Standard Life, Schwab chose ESB technology from IBM largely for its strength in integrating the mainframe assets at the core of Schwab's backend services. Phase 1 of the rollout supports asynchronous messaging based on MQ, WebSphere Business Integration Message Broker, and JMS for distributed connections. Message Brokers running on the mainframe provide data transformation between COBOL and XML. Phase 2 adds synchronous service invocation built on the Web Services Gateway component of the WebSphere Application Server.

In parallel with the ESB deployment, Schwab is converting its core backend functions into managed sets of reusable components called *domains*, each controlled by an explicit "owner" with subject matter expertise and responsibility for long-term maintenance. In addition, IT is creating a new structured Java development environment for reusable service components, intended to free designers to focus on business logic without worrying about the underlying implementation. All of these efforts contribute essential elements to Schwab's Service Oriented Architecture.

Customer Example: Raiffeisen Group

Raiffeisen is the third largest banking group in Switzerland, with 2 million customers and $55 billion in deposits. Since Raiffeisen is structured as a network of independent institutions, its IT environment is by necessity distributed and heterogeneous, but it faces increasing demands for business integration. For example, distributed systems in the branches need to be integrated with centralized securities trading. To achieve this, the company created a "banking bus," a central platform providing a standardized and transparent interface linking heterogeneous applications across the bank – in other words, an Enterprise Service Bus.

Raiffeisen, like Schwab and Standard Life, based its ESB on WebSphere MQ and WebSphere Business Integration Message Broker. Raiffeisen required a mainframe solution because of the heavy transaction volume, and was the first company to deploy the

WebSphere Business Integration Message Broker on an IBM e-server zSeries mainframe. Today, stock trades and position management data are automatically sent over the bus to a variety of decentralized applications, streamlining trading operations and allowing numerous manual processes to be automated and simplified. Over 800 employees have been freed from these manual tasks to spend more time on advising customers. In the next phase, the ESB will allow customers to check their accounts at any office and perform transactions directly online.

Essential Elements of the ESB Integration Pattern

From these examples we've seen that traditional heavyweight messaging middleware like WebSphere MQ and WebSphere Business Integration Message Broker can be leveraged effectively in the ESB integration pattern. Those software products do not *require* messages to be XML and WSDL-based, but they *allow* it. The examples also demonstrate that ESB is not a single software product but an *architectural layer* that creates a service-oriented infrastructure out of multiple middleware components, and tailors it to the particular needs of the enterprise.

It is also clear that linking business functions through a transparent, service-oriented interface does not require the ESB internals to be purely standards-based. In fact, in industries like financial services, where either the transaction volume or established backend infrastructure means a heavy dose of mainframe technology, performance scalability and mainframe integration features are more important today than open standards. In those cases, MQ messaging may be a critical ESB capability.

On the other hand, not all implementations of MQ or other traditional EAI middleware should be considered ESBs. ESB actually represents a particular integration style, whether the bus is lightweight and standards-based or built on established "heavyweight" middleware. With this in mind, let's examine its essential elements:

Baseline Capabilities

1. Standards-Based Communications

A fundamental ESB requirement is support for XML communications that is "standards-based." Today in practice that means support for asynchronous store-and-forward messaging using JMS and synchronous request-reply using HTTP. Although JMS is a standard messaging API, it allows a variety of message bus implementations to act as JMS providers, including widely deployed middleware such as IBM WebSphere MQ. While JMS supports reliable delivery and other quality of service features demanded by real-world SOA, web services QoS standards are still in development, so these features must be layered on top of the standard by the ESB.

2. Universal Connectivity

The ESB must provide connectivity to service providers based on all types of software assets, including:

- Packaged applications, such as SAP and Siebel
- Legacy applications, such as CICS and IMS
- Databases
- EJBs and Java components
- COM and CORBA components

- Web services

For components that are not natively "message-aware," connectivity is delivered in the form of *integration adapters.* These adapters provide the service interface by translating platform-neutral XML message content into the native APIs of the component. Ideally, the ESB should support pre-built adapters for all of these components.

Even components that are message-aware – in particular, legacy mainframe applications – are frequently unable to receive XML messages directly via JMS or HTTP. However, most support traditional messaging middleware such as MQ, with the XML transformed into proprietary formats. Thus universal connectivity implies that if an ESB cannot support MQ as a native messaging backbone, it must at least connect to it through an integration adapter and support the necessary data transformations.

3. Service Mediation

Location independence in SOA means that the requester of a service does not need to know the "address" of the service provider. Instead, the service request message is sent to an intermediary called a *broker*, part of the ESB. The broker provides a number of functions in support of SOA's loose coupling principle, and must be able to compose these functions in user-defined message flows:

- **Content-based routing**. The broker routes the request to a specific service location or *endpoint* based on data in the message itself, as prescribed by user-defined rules.
- **Publish-subscribe integration**. The broker routes the request to all services that have registered a *subscription* to the request.
- **Message transformation**. The broker transforms message content into the schema required by the requested service. For XML messages, this means the broker should include an XSLT or XQuery transformation engine.
- **Message validation and authentication**. The broker authenticates the message sender and ensures that the message content conforms to its predefined schema.

Enabling Real-World SOA

Most of these baseline capabilities are provided by both lightweight ESB offerings and heavyweight business integration middleware, although offerings in the latter category, such as IBM WebSphere Business Integration, offer more in the way of packaged integration adapters and connectivity to legacy software assets than do the lightweight ESBs. But large-scale business-critical SOA solutions require more than this baseline functionality. They require robustness and performance as well. Real-world ESB technology therefore must also provide:

1. Scalability

The ESB needs to be able to process peak message volumes while maintaining service level agreements. For applications like brokerage settlement, these volumes are measured in tens of thousands of transactions per hour. Mature message bus technology like WebSphere MQ has proven its ability to handle this; lightweight ESBs have not. Performance scalability must be designed deeply into the ESB architecture, and may not be easily layered on top of internet standards. Although some lightweight ESBs claim scalability, the imposition of reliable messaging and mediation functionality on them can seriously compromise performance.

2. Platform and Language Independence

While lightweight ESB vendors like to talk about standards, they don't claim to provide a client API for all the platforms and programming languages required by service consumers and providers – the applications that currently run the business. Java and J2EE by itself is rarely enough. The ESB may need to support C and COBOL, and run on platforms from Windows to mainframes. As seen by the earlier examples, some customers even require core ESB components such as brokers to run on a mainframe to handle the transaction volumes.

3. Reliable Delivery

Reliable integration via messaging is more complicated than with synchronous procedure calls. Messages can get lost due to server or network failures. The first message sent might not be the first one received. If a message is re-sent because a reply is not received in time, two copies of the original message may be received. The ESB must provide whatever is missing in the underlying transport protocol to assure that messages are delivered reliably, once and only once, and in the right order. Depending on the quality of service required, this means the ESB must store messages until delivery is confirmed, and be able to manage messages as transactions, supporting commit and rollback.

4. Security

Security is critical. Authentication using LDAP or J2EE security is required at both the user and service level. Role-based access control to ESB administration must be enforced. The ESB must be able to support transport-level security such as SSL and handle encrypted messages, and should support non-repudiation through secure document tracking and audit trails.

5. High Availability

Beyond reliable message delivery, business-critical systems often must provide high availability, including nonstop operation despite component failures. ESBs for these environments should support clustering, redundant hardware, and other features of high-availability configurations.

6. Monitoring and Management

Business-critical solutions demand that service levels be continuously monitored, and ESB components managed, from a centralized administrative console. In practice that means the ESB must make its metrics accessible to enterprise-class system and network management tools such as Tivoli, OpenView, or UniCenter.

7. Graphical Design Tools

Mediation functions such as authentication, validation, transformation, routing, and exception handling are composed in user-defined message flows. The ESB should provide graphical tools to design, maintain, and reuse these flows with a minimum of skill and effort.

Delivering Business Value Today

ESB is more than just a next-generation technology architecture. It can deliver real business value today.

For line-of-business managers, ESB is the key to:

- **Cross-functional business integration**. While leading companies are sweeping away organizational barriers to cross-functional business process integration,

technical integration barriers remain. Ripping and replacing existing stovepipe solutions is not always feasible, and never can be done overnight. Real-world ESB bridges the barriers between the old and the new, linking legacy mainframe, packaged client-server, and new web-centric components in end-to-end business process solutions. The ability to manage cross-functional business processes as a whole means improved transaction speed, better service to customers, and operational cost savings.

- **E-business transformation**. The internet has created new business models linking companies electronically with their trading partners, and new business processes spanning the entire "extended enterprise." XML, standards-based communications, and SOA make e-business possible. Global competition makes it a practical necessity.
- **Agility**. The only constant today is change. Companies must be able to respond rapidly to new customer demands, competitive threats, and emerging opportunities. Bringing new products and services to market quickly requires IT infrastructure designed for change. Because SOA is based on composition and standards-based interfaces, new solutions can be delivered more quickly than in the past, and ultimately at lower cost.

ESB has practical benefits for IT managers as well:

- **Reuse of existing assets**. Real-world ESB allows existing software assets to be reused in new service-oriented solutions, saving time and money. While some work is required to provide service-oriented interfaces for legacy applications, the cost is far less than replacing them entirely, and the work may be reused in multiple applications and processes.
- **Responsiveness**. Traditional software development cycles that take a year or more to complete are under fire. The business demands more responsiveness from IT in order to meet its own agility goals. SOA solutions based on composition of existing assets make IT more responsive.
- **Flexibility**. While platform standardization is always an IT goal, managers know it's an illusion. As business solutions grow larger and increasingly cross organizational boundaries, dealing with a multitude of OS platforms, programming languages, and object models is a fact of life. SOA and an Enterprise Service Bus provide the only way out.

In order to deliver business value today, SOA cannot depend on web services and Internet standards alone. The standards are incomplete, immature, and unproven in high-volume business-critical solutions. On the other hand, enterprise messaging middleware like WebSphere MQ and WebSphere Business Integration Message Broker is both production-proven and capable of providing the needed ESB functionality. Besides, it is likely already deployed in the organization. While lightweight ESB products are just getting off the ground, over 80% of businesses that use messaging middleware have MQ installed.

IT managers need to understand they can begin deploying ESB today by leveraging this existing middleware, applying the principles of service-oriented architecture: platform-neutral XML messaging at the integration hub and implementation-specific adapters for individual services.

Bruce Silver

