Enterprise Service Bus: A Definition

Version: 1.0, Oct 05, 2007

AUTHOR(S):

Anne Thomas Manes
(amanes@burtongroup.com)

TECHNOLOGY THREAD:

SOA and Integration Strategies

Conclusion

An enterprise service bus (ESB) is a middleware solution that enables interoperability among heterogeneous environments using a service-oriented model. Although frequently associated with concepts like “integration” and “mediation,” an ESB also provides a service platform comparable to an application server. In fact, ESBs represent the consolidation of the integration and application server product categories. An ESB's true breakthrough feature is its ability to virtualize services. An ESB's service container abstracts a service and insulates it from its protocols, invocation methods, method exchange patterns, quality of service requirements, and other infrastructure concerns.

BURTON GROUP 7090 Union Park Center · Suite 200 · Midvale, Utah 84047
P 801.566.2880
F 801.566.3611
www.burtongroup.com

Publishing Information

Burton Group is a research and consulting firm specializing in network and applications infrastructure technologies. Burton works to catalyze change and progress in the network computing industry through interaction with leading vendors and users. Publication headquarters, marketing, and sales offices are located at:

Burton Group
7090 Union Park Center, Suite 200
Midvale, Utah USA 84047-4169
Phone: +1.801.566.2880
Fax: +1.801.566.3611
Toll free in the USA: 800.824.9924
Internet: info@burtongroup.com; www.burtongroup.com

Table Of Contents

Synopsis

The enterprise service bus (ESB) product category has matured a great deal since the term was first introduced in 2002. Initially a defensive response by the enterprise application integration (EAI) market to the disruptive influence of web services, ESBs have transitioned into an offensive play to capitalize on the market potential inspired by service oriented architecture (SOA). And pretty much every middleware vendor wants a part of it.

As a result, the term “ESB” has been redefined, overloaded, and diluted to the point where it has no precise meaning. Any two products labeled “ESB” are likely to be very different. But there are some basic commonalities across the products. Fundamentally, an ESB is a service-oriented middleware solution that enables integration of heterogeneous systems by modeling application endpoints as services. In the process, the ESB abstracts and virtualizes the application and enables a much more flexible and adaptable environment.

Although most people associate an ESB with features like integration and mediation, an ESB's primary role in a SOA infrastructure is to act as a service platform—hosting and managing services. In this regard, it is an application server. In fact, an ESB represents the consolidation of the EAI and application server product categories.

Analysis

"Enterprise service bus" (ESB) is the latest rage in the middleware software market. The term was first introduced by Progress Software in 2002 to describe its then-new Extensible Markup Language (XML)-enabled message-oriented middleware (MOM) product, SonicXQ, which has since been renamed Sonic ESB. A few months later, Gartner pronounced that an ESB was a strategic investment, and shortly thereafter, nearly every middleware vendor latched onto the term.

Initially, ESB referred to small group of products that combined the reliable messaging and event-driven processing capabilities of MOM with the multivendor interoperability capabilities of the web services framework (WSF). This new type of product emerged in response to the disruptive innovation engendered by the WSF. The traditional Java Message Service (JMS) and enterprise application integration (EAI) vendors saw their market being eroded by a new, low-cost, standards-compliant middleware. Adding WSF support to their MOM products allowed them to remain competitive.

The ESB market has matured quite a bit since 2002. The market motivation has switched from defensive to offensive. ESB vendors are capitalizing on the industry's enthusiasm for service oriented architecture (SOA). Early ESB products focused much more on "integration" than "service orientation." Recently, the emphasis has switched. Integration is still an important concept in ESBs, but service creation and composition is becoming the primary focus.

With this change of focus, the term "ESB" has been redefined, overloaded, and diluted to the point that it has no precise meaning. Now ESB refers to almost any type of middleware product, and any two products labeled "ESB" are likely to be very different.

One feature that tends to be the same across vendors is market positioning. Most ESB vendors refer to their products as "SOA platforms," and they often position these products as complete solutions that will support all integration requirements and enable "instant SOA." But don't be fooled—there is no silver bullet. Despite the grandiose implications of the name, an ESB product rarely delivers a single solution for enterprise-wide integration, much less a complete SOA solution.

No one product can provide a complete SOA solution. Organizations must use an amalgam of products to implement a SOA infrastructure. The Reference Architecture Technical Position, "SOA Runtime Infrastructure," discusses product alternatives and provides a decision framework for designing a comprehensive runtime infrastructure that supports service-oriented systems.

Although an ESB does not provide a complete "SOA platform," it is a useful component in a SOA infrastructure, and it supports two key functional capabilities: It provides a service platform for hosting and managing services, and it provides mediation services that ensure that messages are delivered properly. Note, though, that some mediation services required by a SOA infrastructure, such as acceleration and security mediation, are not typically supported by ESBs. ESB's role in a SOA infrastructure is discussed in the "Positioning ESB Within a SOA Infrastructure" section of this Technology & Standards document.

The distinction between "SOA platform" and "service platform" is important. "SOA platform" implies a complete SOA infrastructure solution. "Service platform" refers to one of many functional capabilities supplied by a comprehensive SOA infrastructure solution. A service platform is equivalent to an application server that can host services.

One question that Burton Group hears frequently from its clients is, "What ESB should we choose?" Invariably, the answer is, "many." Just as most organizations use multiple application platforms and multiple middleware systems, they are almost certainly going to use multiple ESBs. Each ESB offers different strengths and weaknesses. Some ESBs focus more on service platform capabilities, while others focus more on mediation and integration capabilities. Some ESBs are preconfigured to deeply integrate with particular commercial applications. Each ESB supports a limited set of programming languages, therefore different ESBs are required to support different languages. ESBs also vary in respect to characteristics like ease of use, manageability, performance, scalability, and availability. When choosing an ESB for a project, it's important to identify the project's specific requirements, and choose a product that supports those requirements.

In contradiction to what vendors say, an ESB is not the foundation of a SOA infrastructure. An organization is likely to use multiple service platforms and multiple service mediation systems. No one product will sit in the middle, coordinating the entire environment. In fact, building a SOA infrastructure that relies too heavily on a single product or vendor may be harmful to a SOA initiative. Service-oriented systems are heterogeneous by definition. Business processes invariably reach outside corporate boundaries, and organizations no longer have the option of requiring their partners to use specific products or protocols. A better perspective is to view an ESB as just one of many tools in the toolbox that can be used to implement services and to manage service interactions. It's a useful tool, to be sure—but it has its limits, and another tool might be better for a particular job.

One of the most interesting market dynamics sparked by the rise of the ESB is the way it is impacting the application server market—especially the Java Platform, Enterprise Edition (Java EE) application server market. An ESB is not just an integration system. It is also an application server. It represents a consolidation of the EAI and application server product categories. And this consolidation will be very disruptive to the application server market.

More and more, ESBs are starting to replace Java EE application servers as an application platform. Quite a few ESBs are built on a Java EE application server, and these ESBs have all the capabilities of the underlying application server plus additional integration features. But there is one important difference between an ESB and a Java EE application server: An ESB typically supplies a more abstract container model, and it doesn't require that developers adhere to the complex Java EE component models, such as Java Servlets and Enterprise JavaBeans (EJBs). It supports these models for backward compatibility, but it doesn't require them. ESB vendors appear to be rallying behind a new, language-independent, soon-to-be-standard component model called Service Component Architecture (SCA). Although still early in its development, SCA shows significant potential as a major market force that could disrupt Java EE's dominance in the application server market.

Although most people associate ESBs with SOA, an ESB can be very valuable to organizations that have not yet launched SOA initiatives. An ESB is an application server (at least in most cases), and it provides a simpler, more abstract, more flexible application platform than a traditional Java EE application server.

In 2004, Burton Group published a document stating that Java EE was a standard in jeopardy. Following the release of Java EE 5 (JEE5) in 2006, we updated the document with more dire predictions. See the *Application Platform Strategies* overview, "JEE5: The Beginning of the End of Java EE." The emergence of the ESB as the next generation application platform may be the straw that breaks the camel's back.

What Is an ESB?

More than 40 commercial and open source products are referred to as "ESB." Vendors include the who's who of middleware providers, and then some: BEA Systems, Cape Clear Software, Cast Iron Systems, Fiorano Software, IBM, IONA Technologies, iWay Software, Microsoft, Neudesic, OpenLink Software, Oracle, OrderWare Solutions, Paremus, PolarLake, Progress, Rogue Wave Software, SAP, Skyway Software, SOA Software, Software AG, Sun Microsystems, TIBCO Software, and Vitria Technology. Open source projects include Apache Camel, Apache ServiceMix, Apache Synapse, Apache Tuscany, BEA's fabric3, Blackbird ESB, ChainBuilder ESB, IONA's Fuse, JBossESB, Keystroke ESB.NET, Mirth, Mule, ObjectWeb PETALS, ObjectWeb OpenCeltix, OpenEAI ESB, Sun's Open ESB, and WSO2 ESB. (No doubt there are more.)

Unfortunately, the term "ESB" means different things to different people and therefore generates confusion and misinterpretation. Two products labeled "ESB" can be remarkably different. Take, for example, Sonic ESB and IONA Artix ESB. Both products support mediated application integration using XML messaging, but the similarity ends there. The two products use completely different architectures. Sonic ESB is MOM-based and relies on a central message service to manage and mediate message traffic. Artix ESB, on the other hand, has no dependency on MOM and uses a distributed model to manage and mediate messages at the service endpoints.

IBM, ever the overachiever, offers three very different ESB products:

- **WebSphere ESB:** First introduced in 2005, this product was designed from the ground up to be an ESB. Based on a centralized model hosted in WebSphere Application Server, this product is different from both Sonic ESB and Artix ESB.

- **WebSphere Message Broker:** Previously known as WebSphere Business Integration (WBI), this product is traditionally referred to as an integration broker. IBM rebranded this product an ESB in 2005—presumably to prevent erosion of its market opportunity by the burgeoning ESB market. It is MOM-based and relies on a centralized message broker to coordinate, manage, and mediate traffic (similar to but more centralized than Sonic ESB). It offers the most extensive support for legacy integration among IBM's ESB product family.
- **WebSphere DataPower Integration Appliance XI50:** IBM is stretching the definition of ESB with this product.This type of hardware appliance is more traditionally referred to as an XML gateway, and it doesn't really qualify as an ESB. Although it supports service mediation capabilities, it doesn't support the kind of application server capabilities typically associated with an ESB.

Ironically, before it had released an ESB product, IBM insisted that ESB was not a product. Instead it defined ESB as "an architectural pattern that offers a comprehensive, flexible, and consistent approach to integration." IBM no longer maintains the same collateral on its website as it did in 2005, but it still uses this "architectural pattern" terminology to explain why it offers three distinct ESB products.[1]

Disparate Definitions

The industry is full of disparate definitions for ESB.

Some people stick with the circa 2002 definition: "ESB" is a middleware system that supports both SOA and event-driven architecture (EDA), or more succinctly, ESB = MOM + WSF. Unfortunately, this definition no longer holds water. A number of ESB products don't support event-driven processing or don't rely on MOM. With the standardization of Web Services Reliable Messaging (WS-ReliableMessaging), systems no long require a MOM and a proprietary protocol to enable reliable message delivery. WS-ReliableMessaging can run over Hypertext Transfer Protocol (HTTP). See the "Java Message Service" and "Web Services Framework" sections of this Technology & Standards document for information about MOM, WSF, and WS-ReliableMessaging.

Some people correlate "ESB" with Java Business Integration (JBI), a Java standard specification that defines an integration framework that supports plug-in component and connector modules. The reasoning goes, "if a product implements JBI, then it is an ESB." This statement is reasonably accurate, but the inverse—"if a product is an ESB, then it implements JBI"—is not. Only a small percentage of ESBs implement JBI. JBI obviously isn't particularly popular among ESB vendors that don't support Java, and even among the Java community, it has limited support. For information about this framework, see the "Java Business Integration" section of this Technology & Standards document.

Some people correlate "ESB" with Business Process Execution Language (BPEL), which is an orchestration language. Although some ESBs support BPEL, many do not. Many ESB vendors view orchestration as an aspect of a business process management suite (BPMS) rather than as a feature of an ESB. For information about BPEL, see the "Business Process Execution Language" section of this Technology & Standards document.

Some people correlate "ESB" with composite application development—the ability to wire service components together to create applications. A growing number of ESB vendors have joined the SCA standardization effort, but support for SCA is definitely not pervasive. Some ESBs support much more basic integration capabilities and don't support composite application development. For information about SCA, see the "Service Component Architecture" section of this Technology & Standards document.

Some people (particularly users) use the term "ESB" to refer to a complete SOA runtime infrastructure rather than to any particular type of product. An ESB is just one of many products that work together in an ecosystem to provide a SOA runtime infrastructure. To avoid overloading terms, Burton Group uses the term "ESB" exclusively to refer to products and refers to a SOA runtime infrastructure as a managed communications infrastructure (MCI). For an overview of the MCI, see the "Positioning ESB Within a SOA Infrastructure" section of this Technology & Standards document.

Related Product Categories

Compounding the confusion over ESB's definition is the fact that the boundaries differentiating ESBs from other products are quite blurry. Related product categories include:

- **XML gateway:** XML gateways are hardware appliances that support service mediation—one of the key features of an ESB. In fact, XML gateways support a number of mediation capabilities that ESBs do not, such as acceleration and security mediation. But XML gateways do not provide a service platform—a feature typically associated with ESBs.
- **XML services management (XSM):** Like XML gateways, XSM systems support service mediation. They also support central monitoring and administration of a SOA environment. But they do not provide a service platform.
- **Integration brokers:** Quite a few traditional integration broker vendors, such as IBM, Microsoft, Software AG, and TIBCO, now refer to their products as ESBs. These products tend to be more capable, more complex, more proprietary, and more expensive than other ESBs, but it's difficult to draw a line between the two product categories.
- **BPMS:** Some ESB vendors view orchestration and automated business process management as required features of an ESB, while others view orchestration as a separate capability—one that belongs to the BPMS category. For example, BEA and IBM do not support orchestration in their base ESB products, but they supply separate BPMS products containing a BPEL engine.
- **Application server:** Most ESBs supply a service platform for building and hosting service agents. In that regard, an ESB is an application server. Most application servers provide containers for hosting services, and they typically support a message processing pipeline that supports message mediation and policy enforcement. And through technologies like Java EE Connector Architecture (JCA), they can support diverse resource adapters and legacy integration. They generally don't support a wide range of protocols, and they provide fewer integration tools, but otherwise the line between ESBs and application servers is particularly vague.

So What Is an ESB?

An ESB is a middleware solution that enables interoperability among heterogeneous environments using a service-oriented model, and it represents the consolidation of the EAI and application server product categories.

This definition is vague and imprecise, but then, so is the ESB market. The specific capabilities, features, and standards supported by ESBs, as well as their product architectures, can vary significantly among product implementations. Yet there is some commonality across ESB products. Most ESBs exhibit the following features:

- **Service-oriented middleware:** ESBs model application endpoints as services.
- **Standards compliance:** ESBs are more standards compliant than previous generations of EAI technology, and in particular, they all support multivendor interoperability using the WSF.
- **Virtualization of service agents:** ESBs provide service containers that virtualize a service and insulate the application code from its protocols, invocation methods, message exchange patterns (MEPs), quality of service (QoS) requirements, and other infrastructure concerns.

Beyond these basic characteristics, ESBs are a remarkably diverse and disparate bunch of products. For a more thorough definition of an ESB, including a discussion of the features and standards supported by ESBs, see the “ ESB Definition” section of this Technology & Standards document.

ESB as the Next Generation Application Platform

Although many people correlate an ESB with integration and mediation, its primary function is actually that of service platform—or more specifically the “virtualization of service agents.” A service agent is the application code that implements service functionality, and virtualization of service agents is the true breakthrough in the ESB product category. This bears repeating: An ESB provides a service container that virtualizes a service and insulates it from its protocols, invocation methods, MEPs, QoS requirements, and many other infrastructure concerns.

The *Application Platform Strategies* overview, "VantagePoint 2007–2008: Build for Today, Architect for Tomorrow," discusses the need to separate application and infrastructure concerns. To whit:

> Burton Group loosely defines "infrastructure" as everything other than business logic. From this perspective, functionality commonly required by enterprise applications (such as storage and database access, identity management, authentication and authorization, logging and auditing, and transaction coordination) is considered infrastructure. Separating infrastructure and business functionality improves the consistency and manageability of the former, and allows application developers to focus on the latter.[2]

Infrastructure refers to nonfunctional aspects of a service, including its protocols, invocation methods, MEPs, and QoS requirements. Middleware technologies evolve at a different rate from application functionality, and therefore it is desirable to separate these concerns. Ten years ago, it was quite reasonable to design a system using Common Object Request Broker Architecture (CORBA), but very few people would consider implementing a system today using CORBA. They would more likely use web services. But 10 years from now, something better will be available, and they probably won't consider using web services. Why tie a service to a particular middleware technology if you don't have to?

Most application frameworks don't provide a layer of abstraction between an application and the middleware that exposes that application to external consumers. This is the real beauty of an ESB. It allows a developer to build a service that is completely independent from the technology that will be used to expose its capabilities. Today the service can be exposed using web services; tomorrow, with a slight adjustment to the service's configuration, it can be exposed using a different protocol. Figure 1 illustrates a next generation application server that enables the separation of application and infrastructure concerns.

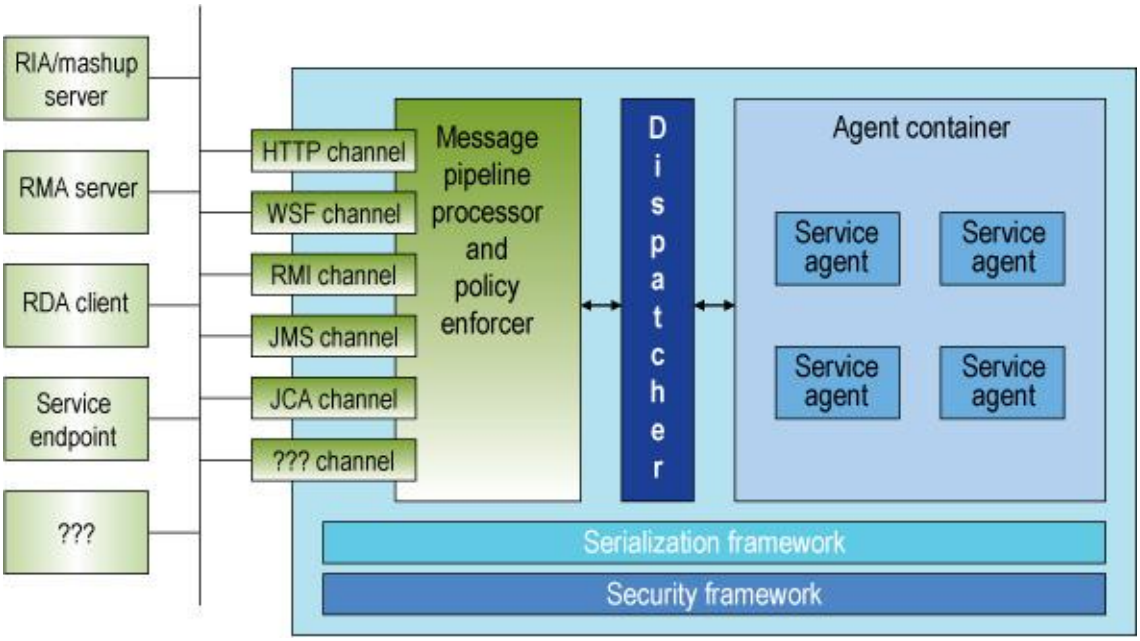

Figure 1: *ESB as the Next-Generation Application Platform*

The service agent running in the platform's agent container is completely separated from the technology used to expose its capabilities to the outside world. The message pipeline processor and policy enforcer can expose the service through any number of communication channels, supporting a wide assortment of client systems, including rich Internet applications (RIAs) and mashups, rich mobile applications (RMAs), rich desktop applications (RDAs), remote service endpoints, and others. The pipeline processor also mediates access to the service agent by enforcing whatever policies apply to the service, such as security, reliability, or transformational policies.

This type of abstract component model, now found in most ESBs, enables the kind of clean separation of concerns between application and infrastructure that Burton Group has been espousing for the past 12 years. It supports the concept of an infrastructure services model (ISM)—in which infrastructure capabilities are externalized from applications and their application platforms and modeled as services that can be applied to service agents and service interactions via declarative policies. The infrastructure itself becomes responsible for ensuring that policies are properly enforced.

Burton Group refers to this next-generation application platform model as the network application platform (NAP)—a virtualized application platform, based on SOA principles, that dissolves current barriers between application platforms. The NAP enables simple, flexible communication and integration across logical and physical boundaries, while maintaining reliability, availability, scalability, serviceability, and security (RAS^3). A diagram representing the NAP is shown in Figure 2.

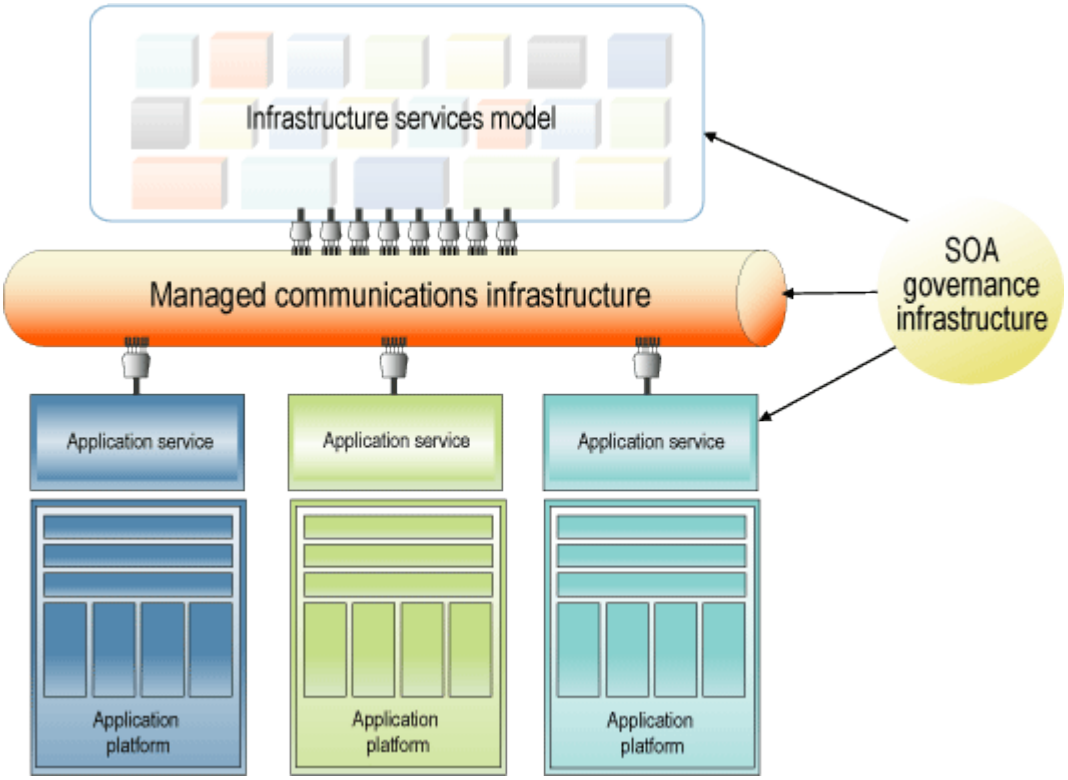

Figure 2: *The Network Application Platform*

The NAP is a conceptual model of an environment that supports SOA initiatives. It shows application services communicating via a managed communications infrastructure (MCI). The MCI relies on infrastructure services within the ISM to ensure the proper delivery of messages between application service endpoints. A SOA governance program provides processes and procedures that ensure that services and service interactions conform to corporate policies, standards, and best practices.

Application services are built on an application platform, which supports the development, deployment, management, security, and execution of the application service. The application platform is also responsible for mediating communications between the application service and the MCI and calling on services from the ISM when dictated by policies.

As a conceptual model, the components in the NAP do not correspond to actual products. In a concrete model, an ESB plays the role of an application platform, and it supports some of the functionality provided by the MCI and the ISM. Some MCI and ISM capabilities may be implemented in an application platform, and some application platform capabilities may be implemented in the MCI or ISM. The purpose of the model is simply to illustrate the need to separate application and infrastructure functionality. In a concrete model, the MCI, ISM, and application platforms work together to create a comprehensive SOA runtime infrastructure. The specific roles that an ESB plays in a SOA infrastructure are discussed in the "Positioning ESB Within a SOA Infrastructure" section of this Technology & Standards document.

ESBs don't yet fully support this model, but they are certainly making progress. Most ESBs support a clean separation between the service agent and the protocols used to expose it. The ability to truly externalize the infrastructure functionality from the application and the application server is still a bit suspect. Very few ESBs fully support a declarative policy enforcement model. Many ESBs rely on proprietary configuration mechanisms that impede centralized policy management and administration and flexible policy enforcement. Nonetheless, ESBs provide a great improvement over traditional application servers. Organizations should consider using ESBs for all application server requirements—not just for services and integration.

For more information about the NAP, see the Reference Architecture Service Oriented Architecture Templates and the *Application Platform Strategies* Root Document, "Turning the Network Into the Computer: The Emerging Network Application Platform."

Evaluating ESBs

When evaluating ESBs, organizations should consider the following criteria:

- Architecture
- Features
- Standards support
- Playing well with others
- Existing deployments
- Licensing terms

Architecture

One of the most fundamental differences among ESB products is the core architecture of their systems. ESBs typically exhibit one of the following five architectures:

- **Extended MOM:** These ESBs correspond to the original definition of ESB. These systems typically distribute multiple nodes across the network and use a MOM infrastructure to support reliable messaging and event-driven processing among the nodes. Although the ESB nodes communicate using a proprietary protocol, service endpoints don't need to be aware of the MOM. All services can be exposed using the WSF or other protocols. These systems tend to implement JBI. Example products include Fiorano ESB, Progress Sonic ESB, Tibco ActiveMatrix Service Grid, and Apache ServiceMix.
- **Extended integration brokers:** Over the last five years, the traditional integration broker vendors have added support for web services and repositioned their products as ESBs. These systems are more standards compliant than they once were, but still tend to be more proprietary than most ESBs. They also tend to provide a very centralized solution in which all messages pass through a centralized broker (or a set of federated broker instances). They typically offer the most extensive integration capabilities. Example products include IBM WebSphere Message Broker, Microsoft BizTalk Server, SAP Process Integration (PI), and Software AG webMethods Fabric.

- **Extended application server:** A number of ESB vendors use a Java EE application server as the basis for their ESB products. These products are typically stronger in terms of service creation and composition than they are in legacy integration. They tend to be rather centralized, although they do support distributed nodes. These systems often implement SCA. Example products include BEA AquaLogic Service Bus, Cape Clear ESB, and IBM WebSphere ESB.
- **Endpoint-based plug-in channels:** A few ESB vendors support an extremely distributed model that implements service mediation at the service endpoint and supports heterogeneous communications using a channel plug-in architecture. Example products include IONA Artix ESB and Microsoft Windows Communication Foundation (WCF).
- **Mediation agents:** Technically, these products don't qualify as ESBs because they don't provide a service platform, but more than one vendor has been known to label this type of product an ESB. These systems support service mediation. Mediation agents can be centralized or distributed. Example products include Apache Camel, Apache Synapse, and SOA Software Network Director, as well as all XSM and XML gateway products.

Features

Each ESB supports a different set of features and capabilities. Products may excel in a wide variety of areas, such as performance, ease of use, legacy connectivity, and management. The “ESB Product Features” section of this Technology & Standards document describes the breadth and scope of ESB features. When choosing an ESB, a project team should carefully identify its project requirements and select an ESB that matches those requirements. ESB features should be evaluated based on five areas:

- Quality of service
- Mediation
- Containers
- Connectors
- Tooling

Quality of Service

Can the ESB support the project's nonfunctional requirements? What are the project's performance, scalability, and latency requirements? How much downtime is acceptable per month or year? Can the ESB support clustering, load balancing, and other features to support future capacity or availability requirements? How quickly can the ESB recover from a failure? What surety level is required? What aspects of the project have to be secured? What mechanisms are available to support those security requirements? Will the project require recoverable transactions? What mechanisms are available to support transactional integrity?

Mediation

Integration projects typically require various types of mediation capabilities. All ESBs support basic mediation capabilities such as routing, protocol bridging, and message transformations. But in some cases, a project may require more sophisticated mediation services. For example, the ESB may need to mediate between service endpoints that use different message exchange patterns or that support different quality of service (QoS) capabilities. Or perhaps the ESB will need to mediate between different security domains via automatic credential mapping. Very few ESBs support these advanced mediation capabilities. (Organizations with these requirements should also consider using an XSM or XML gateway.)

Another aspect to evaluate is the options available to specify mediation policies. Policy specification options may include declarative, programmatic, code annotations, or rules. How easy is it to define the policies, configure them for runtime enforcement, verify their enforcement, manage them, and change them? Does the ESB support centralized management and configuration of the policies?

Containers

ESBs typically provide one or more containers for hosting service endpoints and managing their lifecycle. Each container typically supports a single programming language and container model. Potential language options include Java, .NET languages (e.g., C# and Visual Basic), C++, COBOL, BPEL, PHP, and Ruby. Does the ESB support a standard component model? What languages does it support? What protocol bindings does it support? What message exchange patterns (MEPs) does it support? What is the level of abstraction provided by the component model? Is the application logic suitably separated from the bindings and MEPs? Does the ESB support composition or orchestration? Does the project need these features?

Connectors

ESBs typically supply various resource adapters that enable access to legacy systems. Most ESBs include a few basic adapters that enable integration via the WSF, JMS, JCA, and database access methods such as Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). Many ESBs vendors also supply a host of additional resource adapters (often sold separately) that provide access to numerous commercial application systems, domain-specific business protocols, and legacy middleware systems. What adapters will be required by the project, and are those adapters available? What frameworks are available for implementing custom connectors?

Tooling

What tooling does the ESB supply? Does it supply service modeling tools? Programming frameworks? Graphical tools for defining message transformations and mediation policies? Search, query, and reporting tools? How intuitive and productive are the tools? Does the ESB provide a repository for managing service metadata? Is the repository pre-populated with models and schemas? If it doesn't include a repository, can it integrate with a third-party repository? How well does the tooling integrate with the organization's preferred tools and frameworks? What kind of administrative and management tools are provided? How well do these tools integrate with the organization's preferred administrative tools?

Standards Support

Standards play a key role in ensuring interoperability and portability of services and in enabling an ecosystem of development talent, tooling, and third-party applications. As such, it's important that ESBs support standards. Six groups of standards apply to ESBs:

- HTTP and XML
- WSF
- SCA
- BPEL
- JBI
- JMS

HTTP and XML

HTTP is a standard, pervasive application protocol. It provides the fundamental foundation that enables the World Wide Web, and it supports a core set of architectural principles known as Representational State Transfer (REST). Applications that adhere to the REST principles typically exhibit a number of beneficial characteristics, such as flexibility and scalability. Note, though, that use of HTTP does not guarantee RESTful benefits. An application is RESTful only if it adheres to the REST architectural principles.

XML is a standard, pervasive markup meta-language that can be used to define data vocabularies and formats. It has become the industry's favorite medium for exchanging documents and structured information. When used together, HTTP and XML can be remarkably effective as a universal application and data integration system.

When transferred by HTTP, XML data can be sent in its native form wrapped only in the HTTP protocol, or it can be wrapped in a structured format known as an XML protocol. Popular XML protocols include the Simple Object Access Protocol (SOAP) and the Atom Publishing Protocol (APP). XML data sent in its native form is known as "plain old XML" (POX).

ESBs invariably support communications using HTTP. All ESBs support communications via SOAP, and most systems support POX and other XML protocols. In response to the growing interest in REST, many ESB vendors have started investing in new features that will enable the creation of RESTful services.

For an overview of these standards, see the "HTTP and REST" section of this Technology & Standards document. For further information on HTTP and REST, see the *Application Platform Strategies* overview, "The World Wide Web: An Introduction." See also the slides and recording from the *Application Platform Strategies* TeleBriefing, "REST: Is it the Next Big Thing?" and the slides from a half-day 2007 Catalyst Conference North America workshop on REST, "REST Easy." For information on XML, see the *Application Platform Strategies* report, "Extensible Markup Language (XML) 2005: Core XML Standards and Their Impact on Business Development."

WSF

The WSF is a standards-based service-oriented middleware framework that supports heterogeneous interoperability based on the SOAP XML protocol. As a rule, all ESBs support the WSF, but the WSF is a very complex framework comprising more than 50 specifications. The exact set of WSF specifications supported by an ESB can vary dramatically. The "Web Services Framework" section of this Technology & Standards document provides an overview of the WSF and describes the WSF specifications that are most pertinent to an ESB.

At a minimum, an ESB should comply with the Web Services Interoperability Organization (WS-I)'s interoperability profiles. The WSF specifications frequently support flexibility (i.e., multiple options) in the way certain features or functions can be specified or implemented. Such flexibility can sometimes create interoperability challenges. The WS-I profiles clarify ambiguities in the specifications, limit the number of permitted options, and provide guidance for building web services in an interoperable way.

In particular, an ESB should support the WS-I Basic Profile (BP) and the WS-I Basic Security Profile (BSP). The BP provides guidance when building basic web services that don't use advanced operational semantics, such as security, reliability, transactions, and notifications. The BSP provides guidance when building a web service that requires authentication, message integrity, and message confidentiality. WS-I is in the process of developing the WS-I Reliable Secure Profile (RSP), which will provide guidance for building services that require both security and reliable message delivery.

Most ESBs support the BP and BSP. Many ESBs support advanced operational semantics using nonstandard mechanisms rather than the WSF extension standards. Organizations should evaluate which of these advanced capabilities are needed by their projects and decide how important standards compliance is to their projects. Systems that will need to interoperate outside of their domains are likely to have a stronger requirement for standards compliance.

SCA

The SCA family of specifications is just emerging as a popular service component and composition model standard. Unlike most other programming models, SCA is language and protocol independent. It uses an annotated-code framework that effectively insulates a service's business logic from many infrastructure concerns. For an overview of the model, see the "Service Component Architecture" section of this Technology & Standards document.

SCA has certainly gained broad vendor endorsement, but its effectiveness is yet to be proven. The model is potentially convenient and highly productive, but only if it succeeds in addressing a number of potentially dangerous issues:

- **Code-centric development:** SCA uses an annotated code-driven development methodology, automatically generating interfaces from application code. A code-centric approach leads developers to generate numerous incompatible schemas that hinder interoperability efforts.
- **Proprietary annotations:** SCA defines a number of standard annotations, but there's nothing to prevent vendors from creating additional proprietary annotations that could derail interoperability and portability.
- **Interoperability:** SCA leaves a number of features as an exercise for the implementers. When a feature is not fully specified, interoperability issues usually emerge. This issue is compounded by the fact that SCA is designed to support multiple languages. Interoperability between implementations has yet to be tested.
- **State management:** State management is always a tricky aspect of a component model. SCA specifies that the container must provide automatic state management, and it defines multiple session scopes. It's concerning, though, that it doesn't define singletons.

These issues aside, SCA is forging new ground by defining a simple, abstract component and composition model that can be implemented in any language. Language implementations have been specified for Java, C, C++, COBOL, BPEL, PHP, and Ruby. Many ESB vendors (with the notable exception of Microsoft) have endorsed SCA, and a number of vendors have delivered support for SCA in their products.

BPEL

For the past five years, BPEL has represented the Promised Land for rapid development of automated business processes. The vision enables a business person to sit down with a simple modeling tool and quickly design a business process by wiring together a set of existing service components with a few rules and workflows.

Unfortunately, this vision is a bit disconnected from reality. BPEL modeling tools are not quite simple enough for the average business person to use. But BPEL can be an effective development tool in the hands of a professional developer. For an overview of BPEL, see the “Business Process Execution Language” section of this Technology & Standards document.

A number of vendors include an orchestration engine with their ESBs, but not all of them do. Some vendors don't provide an orchestration engine at all, while others sell their orchestration engines as separate products. When selecting an ESB, an organization should first assess whether an orchestration engine is required, and if so, determine whether that orchestration engine needs to be tightly integrated with the ESB.

As a general rule, orchestration engines support BPEL. Even though the language has a number of limitations, the siren call of a standard orchestration language (plus overwhelming vendor support) has given BPEL a decisive victory in the battle of the orchestration languages.

BPEL defines a standard orchestration syntax for specifying workflows that invoke web services in a predefined sequence. BPEL is useful for describing relatively simple automated business processes, but it has two significant limitations. First, a BPEL process can only invoke web services. If an application isn't exposed via a web service, it cannot be incorporated into the process. And second, BPEL does not currently support human participation in a workflow. (An effort is in process to address this limitation.) Another concern when using BPEL is that many vendors have implemented proprietary extensions to address some of BPEL's limitations. In fact, BEA and IBM have proposed a way to extend BPEL with Java (BPELJ).

Although most orchestration vendors support BPEL, BPEL compliance is a fuzzy thing. Some vendors have built their orchestration engines from the ground up to support BPEL, and their orchestration engines process BPEL natively. Other vendors adapted their pre-existing orchestration engines, and these products tend to support BPEL only through import and export. The BPEL is transformed into the engine's native processing language, which may have different capabilities than BPEL. If a process uses a feature that is not supported by BPEL, the import and export won't work. In that situation, the project team must determine whether portability of BPEL processes is a system requirement.

For more information about the strengths and weaknesses of BPEL, see the *Application Platform Strategies* report, "Crossing the Divide: The Mechanics of Process Execution."

JBI

JBI defines a standard integration framework for Java-based middleware systems. JBI defines a standard plug-in model that enables containers and connectors to connect to a centralized message router. Conceivably, a JBI-compliant container or connector could work with any JBI-compliant message router. Note, though, that JBI does not define any application-level programming models. It only applies to developers building system-level connectors and containers. For more information about JBI, see the "Java Business Integration" section of this Technology & Standards document.

The standard plug-in model has an appealing ring to it because it could enable an ecosystem of after-market and open source JBI plug-ins. This has certainly been the case with similar Java standards, such as JCA. But a healthy ecosystem requires pervasive market buy-in. Unfortunately, JBI has not managed to win the endorsement of the Java superplatform vendors.

JBI has a number of characteristics that dim its luster:

- JBI is based on a message router, which means that JBI is predisposed to support the extended MOM architecture. Obviously JBI appeals to MOM vendors looking to extend their products, but this architectural preference reduces its appeal to non-MOM vendors.
- JBI requires all messages to pass through the message router, which involves a transformation on both entry and exit, even when both endpoints use the same formats and protocols. Many view this overhead as unnecessary and burdensome.
- JBI defines its plug-in interfaces only in Java. This language dependency contradicts the current trend in the ESB market to embrace multiple languages.

The fact that JBI does not define any application-level programming models reduces its relevance as an ESB selection criterion. A developer implementing a service never touches the JBI application programming interfaces (APIs), and therefore doesn't care if it is there. JBI's value is dependent on the availability of JBI-compliant connector plug-ins. If the JBI connector plug-in ecosystem doesn't take off, then JBI provides no added value.

JMS

JMS is the standard Java API used to interface with MOM systems. Any MOM vendor that supplies a Java programming API invariably supports JMS. The Java EE specification also requires support for JMS. JMS is one of the most pervasive APIs in the industry—at least in the Java community.

Although JMS is widely implemented and used, it has three significant shortcomings:

- **No multivendor interoperability:** Although JMS provides a common API to multiple MOM products, it does not support multivendor interoperability. Each MOM product communicates using a proprietary protocol, and two applications must use the same protocol to communicate.
- **Low-level programming model:** The low-level nature of the API tightly couples an application to the JMS programming model and MOM protocols, reducing the flexibility and agility of the application's architecture.
- **Restricted to Java:** JMS is only available to Java applications, and no comparable standard API exists for other languages, such as C++, C#, Visual Basic, Python, Ruby, and COBOL.

Most ESBs include support for JMS, and in doing so, they add a layer of abstraction to JMS that addresses these shortcomings. ESBs typically support a variety of MOM protocols and perform automatic protocol bridging that enables interoperability between multivendor systems. Some ESBs use a particular MOM product as their primary communications system between ESB nodes, but service consumers are not required to use that MOM to interface with services hosted in or exposed by the ESB.

ESBs typically provide a service-oriented component model that allows developers to build services that are independent from the middleware used to expose them. For example, most ESBs allow developers to implement services as "plain old Java objects" (POJOs). The service container hosting the service automatically manages the interactions between the POJO service and its external interfaces, such as JMS.

And because the service container manages all communications on behalf of the service, the ESB can expose the service functionality through multiple interfaces and protocols, such as JMS, SOAP, or HTTP, thereby making the service accessible to any application regardless of programming language or platform. Some ESBs also provide multiple containers that support different programming languages.

Playing Well with Others

A SOA infrastructure is an ecosystem of cooperating products that together enable services to interact properly and effectively. An ESB is just one product in this ecosystem. Therefore, it is essential that it plays well with the other products in the ecosystem.

The Reference Architecture Technical Position, "SOA Runtime Infrastructure," outlines the functional requirements of a SOA infrastructure and identifies alternatives that can be used to support those requirements. The "Positioning ESB Within a SOA Infrastructure" section of this Technology & Standards document discusses the functional requirements that an ESB can support. But other products will be necessary.

In many cases, the organization has a number of existing middleware, platform, management, and mediation systems deployed. As a SOA initiative progresses, the organization is likely to deploy additional platform, management, and mediation systems, as well as governance systems, such as registries and repositories, XSM systems, and XML gateways. How easily does the ESB integrate and work with these other systems?

Some vendors offer a comprehensive suite of SOA infrastructure products, including ESBs, orchestration engines, security systems, registry/repositories, management suites, and development tools. In some cases these suites use proprietary interfaces to enable tighter cohesion among the suite components. In that case, how easy is it to swap out one component and replace it with a third-party offering? Is single-vendor lock-in an acceptable risk?

Existing Deployments

Very few organizations have the luxury of starting with a green field. Invariably, the organization has existing deployments of various application platforms and middleware systems. Many middleware vendors now provide an upgrade path from their legacy products (e.g., MOMs, integration brokers, BPMS, and application servers) to their new ESB products. These existing deployments may predispose an organization to select an ESB that is compatible with the installed system. For example, if an organization has extensive deployments of WebSphere MQ, it might be predisposed to upgrade to WebSphere Message Broker. Likewise, if an organization has extensive deployments of BEA WebLogic Server, it might be predisposed to upgrade to BEA AquaLogic Service Bus.

Licensing Terms

Although software licenses typically constitute only a tiny fraction of the total cost of ownership of a SOA initiative, unexpected license or subscription fees can be disruptive to a project. When evaluating an ESB, estimate the total licensing terms for the product both at initial deployment and at various stages throughout the SOA initiative. What are the fees for the base product? What are the fees for add-on products, such as tools, agents, management suites, resource adapters, and security systems? What fees will be imposed when additional ESB nodes need to be deployed?

Recommendations

The following recommendations provide guidance related to the positioning and selection of ESB products.

Recognize the Value and Limitations of an ESB

Many ESB vendors position their products as universal integration environments that provide a fast path to SOA, but such lofty goals are a bit of an exaggeration. Despite the grandiose name and despite what the vendors and some other analysts say, an ESB is a tactical investment. An ESB is a useful tool, but it is not a universal integration system, nor does it provide a complete SOA infrastructure.

Understand ESB's Role in a SOA Infrastructure

A SOA infrastructure is an ecosystem of products and services that work together to enable service oriented systems. An ESB can be a valuable member of that ecosystem, but it is just one of many products that must work together. (Playing well with others is a critical characteristic.) An ESB is no more important than any other product in the ecosystem. No one product will sit in the center and coordinate the environment. Most organizations will use multiple ESBs in their SOA infrastructure, just as they currently use multiple application languages, platforms, and middleware systems.

An ESB's primary role in a SOA infrastructure is to act as a service platform. It hosts service agents, and it exposes legacy resources as services. It can also support mediation services, but this role is secondary to its role as a platform. The ESB's true value is in the way that it virtualizes service agents.

For a discussion of SOA infrastructure product alternatives and a decision framework for designing a comprehensive infrastructure, see the Reference Architecture Technical Position, "SOA Runtime Infrastructure." For an examination of an ESB's role in that infrastructure, see the "Positioning ESB Within a SOA Infrastructure" section of this Technology & Standards document.

Recognize That ESBs Are Not Just for SOA

Although everyone correlates them with SOA, ESBs offer tremendous value to organizations that have not embarked on a SOA initiative. ESBs represent a consolidation of the EAI and application server product segments. ESBs offer a simpler, more intuitive, and less expensive alternative than previous generations of EAI products. At the same time, they provide a simpler, more abstract, and more powerful application server than traditional Java EE application servers. Most ESBs offer the same type of reliability, availability, and scalability features typically associated with application servers, but without the complexity inherent to Java EE. These characteristics are particularly true of the ESBs that implement the extended application server architecture, but any ESB that supports service agent virtualization delivers benefit. Organizations should consider using ESBs for all application server requirements—not just for service enablement and integration.

Don't Waste Time Trying to Pick One "Universal" ESB

An ESB is not a universal integration solution. No one product is going to address all of an organization's integration requirements. Each ESB offers different features and benefits. Just as most organizations use multiple languages, platforms, and middleware, they are also going to use multiple ESBs.

Pick the Right Tool for the Job

Before selecting an ESB, it is important to understand a project's requirements and its strategic goals. Then evaluate ESB options based on those requirements. Keep in mind that some projects may not need an ESB.

Consider Upgrading Older Systems to ESB Equivalents

Most organizations have a variety of older middleware systems, such as application servers, CORBA systems, MOMs, and integration brokers. The MOMs and integration brokers use proprietary protocols that propagate vendor lock-in strategies, and all of these older systems involve a tight binding between application code and the middleware. Although the legacy applications based on these older systems aren't going away any time soon, organizations should think twice before investing further in these older middleware systems.

Most middleware vendors provide a new and improved ESB offering that extends and enhances their older middleware systems. These ESBs can provide a legacy encapsulation environment that paves the way for gradual migration away from a tight integration model to a more flexible and loosely coupled environment with clean separation of application and infrastructure concerns. A good strategy is to upgrade existing middleware systems to their ESB equivalents in order to leverage and extend the current investments.

Beware of ESB Vendor Lock-In Strategies

One of the hidden “gotchas” in today's ESB market is the potential for vendor lock-in. This is especially true of vendors that use proprietary interfaces to enable better cohesion among the product components in their integrated SOA suites. When evaluating an ESB, pay particular attention to how well it works with third-party products.

The Details

This section defines enterprise service bus (ESB), describes the range of ESB features and standards, and positions ESB within the context of a service oriented architecture (SOA) infrastructure.

ESB Definition

An ESB is a middleware solution that enables interoperability among heterogeneous environments using a service-oriented model. An ESB models an application endpoint as a service. The ESB may host the service agent locally, or the service may execute remotely. In both cases, the ESB provides an abstraction layer that virtualizes the service and separates it from infrastructure concerns. The ESB makes the service accessible to other applications via one or more middleware protocols. As a general rule, one of the protocols that an ESB supports is Simple Object Access Protocol (SOAP), but it doesn't require all services to communicate via SOAP. The ESB mediates interactions between service endpoints and enables dissimilar systems to interoperate.

Latest Generation of EAI

ESB products have evolved from and build on the legacy integration capabilities of traditional enterprise application integration (EAI) products, such as integration brokers. ESBs enable integration among environments such as .NET, Java, Common Object Request Broker Architecture (CORBA), Customer Information Control System (CICS), Information Management System (IMS), Tuxedo, and numerous packaged applications. ESBs support communications using Hypertext Transfer Protocol (HTTP), SOAP, Internet InterORB Protocol (IIOP), various proprietary message-oriented middleware (MOM) protocols, as well as numerous domain-specific business-to-business protocols, such as ACORD, Electronic Data Interchange (EDI), Financial Information eXchange (FIX), and Society for Worldwide Interbank Financial Telecommunication (SWIFT).

The distinction between ESBs and previous generation EAI products, such as integration brokers and business process management suites (BPMSs), is often blurry. In fact, a number of integration broker vendors simply attached the "ESB" moniker to their products after adding support for SOAP and the web services framework (WSF). As a general rule, ESBs support pervasive and vendor-independent protocols, such as HTTP and SOAP, although a number of ESBs maintain a dependency on proprietary MOM systems. ESBs generally prefer to work with Extensible Markup Language (XML) data formats, although many ESBs can work with other formats. ESBs typically support multiple message exchange patterns (MEPs), such as request/response, one-way messaging, and event-driven patterns.

ESB Product Features

ESBs represent the consolidation of the EAI and application server product categories. They support integration and mediation capabilities as well as application hosting and lifecycle management capabilities.

At a minimum, an ESB supports the following service mediation features:

- **Routing:** An ESB acts a broker, routing messages to service endpoints based on a variety of factors, such as system load, time of day, identity, message attributes, or message content. An ESB provides a virtualization layer that enables dynamic service selection, versioning, location, and binding.
- **Protocol bridging**: An ESB supports automatic mediation across multiple protocols, such as SOAP, HTTP, CORBA, Java Remote Method Invocation (RMI), and various proprietary and domain-specific messaging protocols.
- **Message transformation:** An ESB provides XML data processing capabilities, supporting validation, aggregation, filtering, and transformation of XML message content. Some ESBs also support data processing for non-XML formats.

ESBs also support the following service platform features:

- **Service hosting:** An ESB provides one or more containers for hosting services and managing service lifecycle.
- **Service component model:** An ESB supports an abstracted component model that developers can use to implement services. The component model coordinates the interactions between the container and the service agent and enables a service to expose its interface through a variety of protocols and MEPs.

Many ESBs support other features, such as:

- **Resource adapters**: Many ESB vendors provide resource adapters that developers can use to implement connections to various legacy applications and data sources, and then expose these connections as services. Resource adapters are often sold separately.
- **Composition**: Many ESB products include tooling and frameworks that enable developers to wire services together to create a simple composite application.
- **Orchestration:** Some ESB products support the development of composite services using an orchestration model. An orchestrated service is a service that calls other services in a predefined execution pattern or workflow. An orchestration engine coordinates the execution of the pattern at runtime.
- **Reliable messaging**: Many ESB products support reliable message delivery semantics, including best effort, persistent queuing, at least once, at most once, exactly once, and ordered messaging.
- **Event processing**: Many ESB products support an event-driven interaction pattern via publish and subscribe capabilities.
- **Transactions:** Some ESB products support transactional integrity. The persistent queuing systems that enable reliable messaging and event processing typically operate as transactional data resources, and these queuing systems can participate in heterogeneous transactions. In addition, an ESB product may supply a distributed transaction manager that can coordinate a distributed transaction across heterogeneous data resources using a two-phase commit (2PC) protocol or compensating transactions.
- **MEP and capability mediation:** In some cases, the ESB can automatically mediate the interaction between two applications that communicate using different MEPs or that support different capabilities, such as reliability and transactions.
- **Security mediation:** In a few rare cases, the ESB can mediate an interaction that crosses security domains by mapping credentials from one domain to credentials that are acceptable by the second domain. (All ESBs can control access to services through authentication, but only a few products support federation across security domains.)
- **Tooling:** An ESB typically provides tooling for design, development, configuration, deployment, operation, and management of services. Tooling may be model-driven, graphical, or invoked using a batch command. Some ESBs come pre-populated with models and metadata related to specific commercial application or domain-specific models and schemas.

Positioning ESB Within a SOA Infrastructure

While an ESB can provide valuable features that support the development and integration of services, it by itself does not constitute a complete SOA infrastructure. The Reference Architecture Technical Position, "SOA Runtime Infrastructure," provides a decision framework for designing a runtime infrastructure that supports service-oriented systems.

The Technical Position describes a SOA runtime infrastructure as a managed communications infrastructure (MCI). This infrastructure is managed in the same sense that a managed code environment is managed. The infrastructure is responsible for managing communications between service endpoints and ensuring that messages are delivered properly in accordance with defined policies.

The Technical Position approaches a SOA runtime infrastructure from a functional perspective. It identifies six core functional capabilities that a SOA runtime infrastructure must support:

- **Middleware:** Provides the means for service endpoints to communicate

- **Service platform:** Hosts and manages the lifecycle of service endpoints
- **Service mediation:** Manages intermediaries that mediate dissimilarities between service endpoints
- **Service administration:** Configures and controls service endpoints, service intermediaries, and infrastructure components
- **Service monitoring:** Monitors service endpoints, service intermediaries, infrastructure components, and message traffic
- **System of record:** Maintains an index of service information (stored in metadata and policy repositories) and enables information exchange among infrastructure components

Figure 3 shows a functional model of the MCI.

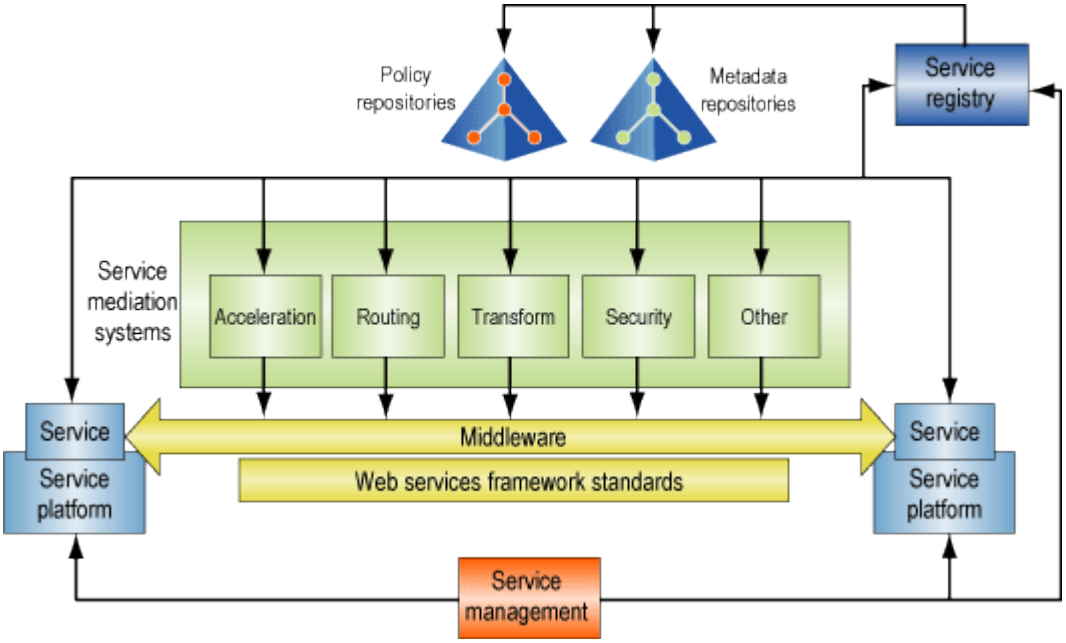

Figure 3: *Functional Model of an MCI*

Service endpoints reside on a service platform. They exchange messages using some type of middleware. The message exchange is managed by mediation systems, which can be deployed anywhere along the message path (including within the service endpoint). The rules governing the message exchange are defined by policies, and it is the responsibility of the infrastructure to ensure that the policies are properly enforced. A system of record maintains information about the entire environment, and provides pointers to metadata and policies (typically stored in various repositories). The entire environment should be monitored to enable proactive management of issues and incidents. A service administration console enables centralized management and configuration.

The Technical Position describes each functional component and identifies the various product alternatives that can be used to implement the functionality. Unfortunately, product categories don't particularly align with these functional capabilities. Many products address aspects of multiple functional components, but in many cases they address only a portion of the required capability. Multiple products are typically required to support all functional requirements.

Table 1 describes ESB capabilities in relation to the MCI functional requirements.

Function	ESB capability	Comments

Middleware	Full support	ESBs typically support multiple middleware technologies with a variety of MEPs and configurable quality of service (QoS). Even though an ESB provides full support for middleware capabilities, most organizations will use multiple middleware products.
Service platform	Full support	ESBs typically provide containers to host service endpoints, resource adapters to expose legacy systems as services, and orchestration engines. Even though an ESB provides full support for service platform capabilities, most organizations require multiple service platforms to support different languages and middleware systems.
Service mediation	Partial support	ESBs support routing, protocol bridging, and message transformation. Some ESBs support MEP and capability mediation. Very few ESBs support security mediation. Additional products, such as XML gateways and XML services management (XSM) systems are typically required to implement full support for mediation.
Service administration	Limited support	ESBs typically supply tools for administering services that reside in the ESB, but they don't provide a central administration tool for the entire SOA environment. Additional products such as XSM systems are typically required to enable centralized administration.
Service monitoring	No support	ESBs do not provide operational dashboards for monitoring and managing the SOA environment. Additional products, such as XSM or SOA extensions to an enterprise systems management solution, are typically required to support service monitoring.
System of record	Varies	Some ESBs include an integrated registry and repository that manages and enables sharing of service metadata, policies, and artifacts. In most cases, a registry and repository is sold separately.

Table 1: *ESB Capabilities Within the Context of the MCI Functional Model*

Standards That Apply to ESBs

As a general rule, ESBs are more standards compliant than previous generations of EAI technologies. Given that the industry has not established a concrete definition for ESB, though, it should come as no surprise that the industry has also not established definitive standards for ESBs.

The wonderful thing about standards is that there are so many of them. In fact, quite a few standards apply to ESBs—although not all ESBs support all of them. In some cases, the standards complement each other. In other cases, they overlap. The most prevalent standards supported by ESBs include the following:

- HTTP and REST
- Java Message Service
- Web services framework
- Business Process Execution Language
- Service Component Architecture
- Java Business Integration

HTTP and REST

HTTP is the standard, pervasive application protocol that enables the World Wide Web. HTTP is a stateless, connection-oriented, client/server protocol that implements the architecture of the Web, as specified by Roy Fielding in his dissertation on Representational State Transfer (REST).

REST is an architectural style of system development, not a middleware technology. REST is similar to and compatible with SOA, although it imposes additional constraints on service interactions. The first and most fundamental tenet of REST is that applications interact with resources, and that all interactions with a resource involve the exchange of representations of the resource. One can think of REST as a resource oriented architecture (ROA) rather than a service oriented one.

A resource is a constrained type of service. It must have a name (a Uniform Resource Identifier [URI]), and it must expose its capabilities using a uniform interface. REST is closely associated with HTTP because HTTP provides this type of uniform interface—all resource interactions use simple, generic methods: GET, POST, PUT, and DELETE. Other important constraints of the REST style include clean separation of interface from implementation, stateless communications, cache-ability of resource representations, and the use of hypermedia (e.g., hyperlinks) to reference and manage state. These constraints bestow predictable benefits to applications that adhere to them, including extreme scalability and flexibility. The World Wide Web owes its scalability and flexibility to the REST architecture.

Java Message Service

Many ESBs rely on a MOM foundation for reliable messaging and event-driven processing. The Java Message Service (JMS) is the standard Java messaging application programming interface (API) for interacting with MOM systems. JMS 1.1 was approved by the Java Community Process (JCP) in December 2003. All Java-based MOM products, such as IBM WebSphere MQ, Progress Software SonicMQ, and TIBCO Rendezvous, support JMS. JMS is also a required feature in the Java Platform, Enterprise Edition (Java EE); therefore, all Java EE-compatible application servers include support for JMS.

JMS provides a common messaging API that can be used with any MOM system. MOM vendors are responsible for mapping the JMS API to the messaging capabilities supplied in their products. In some cases, the JMS API may not provide access to all features in the product. Some MOM products—especially those that predate the JMS specification—also supply a proprietary API that provides access to the middleware's full functionality.

When using JMS, an application interacts with a MOM product at a relatively low level. The application uses JMS to establish a connection with the MOM's messaging service. It then sends and receives messages to and from message destinations (queues or topics) managed by the messaging service. The application can specify a set of delivery policies indicating the QoS desired, including at least once, at most once, or exactly once. If the application is sending multiple messages, it can specify that it wants the messages delivered in order. The messaging service ensures that the messages are delivered to the appropriate target applications according to those delivery policies. MOM products can guarantee reliable delivery by persisting messages in a durable message queue. JMS supports asynchronous, intermediated communications by default, disconnecting the receive operation from the send operation. An application does not receive a message until it explicitly retrieves the message from the messaging service.

JMS also supports event-driven processing and publish and subscribe (pub/sub) interaction patterns. In this scenario, an application subscribes to a particular topic and receives a notification whenever a message relating to the topic is published to the topic queue. Any number of applications may subscribe to the same topic, supporting one-to-many message delivery. For more information about JMS, see the *Application Platform Strategies* overview, "JEE5: The Beginning of the End of Java EE."

Web Services Framework

The web services framework (WSF) provides a language- and platform-independent set of middleware and communication technologies that enable service-oriented design and standards-based interoperability. The framework is based on an extensive set of composable standards and specifications, often referred to as WS-*, which are in various stages of development and readiness. For an in-depth analysis of the WSF standards and their states of readiness, see the Reference Architecture Template, "Web Services Framework Standards."

The WSF consists of a set of functional components that work together to supply a complete service-oriented framework. Each functional component comprises a set of specifications that enable the functional capability. Each specification addresses a particular capability in the framework, and developers can configure a service to use only the specific capabilities required by the service. The architecture is organized into four logical groups, as highlighted in Figure 4.

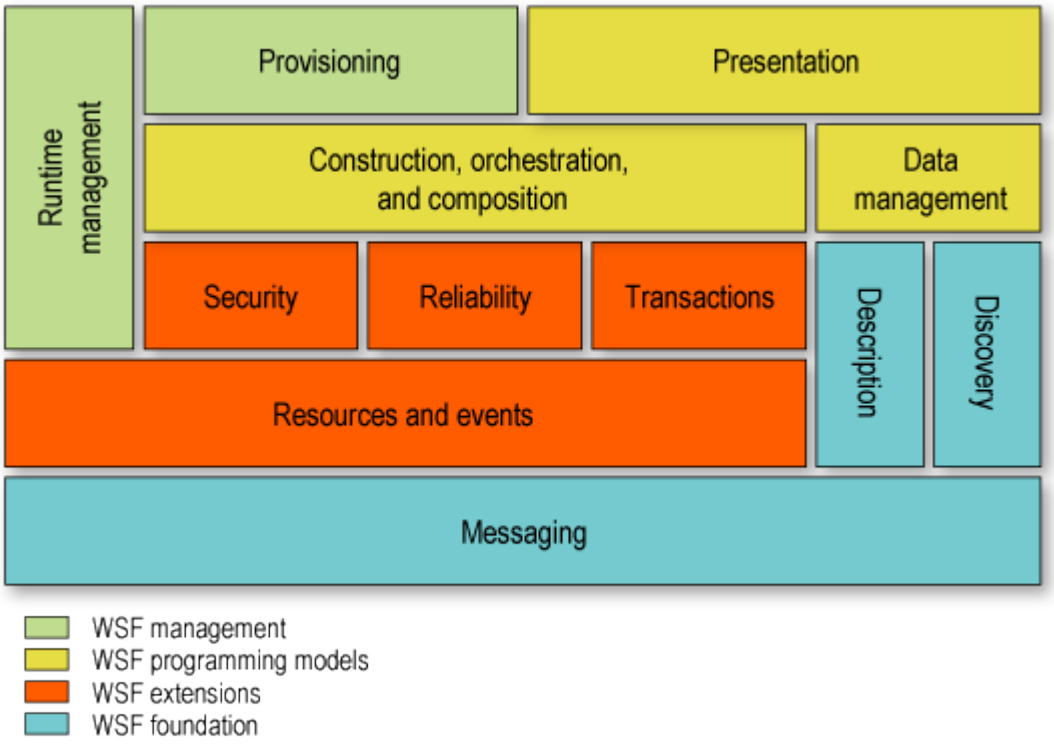

Figure 4: *The WSF Composable Architecture*

The four logical groups are:

- **WSF foundation:** Colored blue in Figure 4, the WSF foundation comprises architectural components that support basic service interaction requirements, including messaging, description, and discovery.
- **WSF extensions:** Colored orange in Figure 4, the WSF extensions comprise architectural components that support advanced operational semantics, such as state management, security, reliability, and transactions.
- **WSF programming models:** Colored yellow in Figure 4, the WSF programming models comprise architectural components that support development activities, including service construction, service orchestration, application composition, data management, and presentation requirements.
- **WSF management:** Colored green in Figure 4, WSF management comprises architectural components that support runtime management and provisioning requirements.

The capabilities of an ESB correspond to these logical groups. All ESBs support the basic service interaction capabilities as defined in the WSF foundation. ESBs typically support many of the advanced operational semantics defined in the WSF extensions. Many ESBs also support various programming models. An ESB may also support management interfaces for monitoring and control.

WS-* comprises more that 50 specifications. The following sections discuss only the most pertinent specification families that apply to ESBs.

WSF Foundation

The WSF foundation specifications support messaging, description, and discovery. The most pertinent WSF foundation specifications include:

- **SOAP:** SOAP is a language-independent XML messaging protocol that defines a standard message format, standard message-exchange patterns, standard message-processing rules, and an extensibility framework. SOAP 1.1 was published in May 2000. It was never ratified, but it has become a de facto standard. SOAP 1.2 and a family of related specifications were ratified by the World Wide Web Consortium (W3C) in June 2003 and updated in April 2007.
- **MTOM:** The SOAP Message Transmission Optimization Mechanism (MTOM) enables the attachment and optimized transfer of binary data with a SOAP message. MTOM can work with both SOAP 1.1 and SOAP 1.2. MTOM was ratified by W3C in January 2005.
- **WS-Addressing:** Web Services Addressing (WS-Addressing) supports asynchronous communication and message routing. WS-Addressing also defines a standard syntax for referencing a web service endpoint. WS-Addressing 1.0 was ratified by W3C in May 2006.
- **XML Schema:** XML Schema is a family of specifications that define a syntax for describing the structure, content, and semantics of XML documents, such as the documents transferred using SOAP. XML Schema 1.0 was ratified by W3C in October 2004. Standardization of XML Schema 1.1 is in progress at W3C. A Working Draft was published in August 2006.
- **WSDL:** Web Services Description Language (WSDL) is an XML-based machine-readable service-interface definition language that describes a web service, and middleware frameworks can generate interface code from a WSDL description. A WSDL description specifies the abstract interface of the service, bindings that map the abstract interface to concrete protocols, and the location of the service's endpoints. WSDL uses XML Schema 1.0 to describe the structure of XML message formats. WSDL does not pre-suppose the use of SOAP, (a binding can describe other protocols), but SOAP and WSDL are typically used together.WSDL 1.1 was published in March 2001. It was never ratified, but it has become a de facto standard. WSDL 2.0 was ratified by W3C in June 2007.
- **WS-Policy:** The Web Services Policy Framework (WS-Policy) and its related specifications provide a means to declaratively specify the constraints, capabilities, and QoS expectations associated with a service. For example, developers can specify routing, security, reliability, and transaction requirements using WS-Policy. Policies are specified using a domain-specific policy assertion language (PAL). Many WSF extension specification families include a PAL for expressing declarative policies. WS-Policy 1.5 was ratified by W3C in September 2007.
- **UDDI:** Universal Description, Discovery and Integration (UDDI) defines a registry data model, and it defines standard protocols for registering a service and querying the registry. UDDI 3.0.2 was ratified by the Organization for the Advancement of Structured Information Standards (OASIS) in February 2005.
- **WS-MetadataExchange:** Web Services Metadata Exchange (WS-MetadataExchange) defines a standard protocol for querying a service or service avatar to retrieve the service's metadata, including XML Schema, WSDL, and WS-Policy descriptions. The last revision (written by BEA Systems, CA, IBM, Microsoft, SAP, Sun Microsystems, and webMethods [now part of Software AG]) was published in August 2006. The authors have requested a public review of the specification prior to submitting it to a standards body.

All ESBs support SOAP 1.1, WSDL 1.1, and XML Schema 1.0. A growing number of ESBs support SOAP 1.2, MTOM, and WS-Addressing. Very few products support WSDL 2.0. A few ESB vendors use WS-Policy to specify a service's security policies, but they rarely use WS-Policy for other service constraints and capabilities. Most ESBs support integration with registries via UDDI, but very few support WS-MetadataExchange.

For information about these and other WSF foundation standards, see the Reference Architecture Template, "Web Services Framework Standards." For more information on SOAP and WSDL, see the *Application Platform Strategies* overview, "The Advent of the Network Platform: Web Services Move into the IT Fabric." For a more detailed discussion of WS-Policy, see the *Application Platform Strategies* overview, "VantagePoint 2005-2006 SOA Reality Check." For an in-depth discussion of UDDI, see the *Application Platform Strategies* Technology & Standards document, "Registry Services: The Foundation for SOA Governance."

WSF Extensions

The WSF extension specifications support advanced operational semantics, such as security, reliability, and transactions. The most pertinent WSF extension specifications include:

- **Web Services Security (WS-Security)** and its family of related standards provides the means to attach security information to a SOAP message, enabling authentication, authorization, auditing, message confidentiality, and message integrity. Related standards define security token formats, a domain-specific PAL (WS-SecurityPolicy), a protocol for obtaining security tokens (WS-Trust), and a system for establishing secure sessions (WS-SecureConversation). WS-Security 1.1 and the various security token profiles were ratified by OASIS in February 2006. WS-Trust 1.3 and WS-SecureConversation 1.3 were ratified by OASIS in March 2007.WS-SecurityPolicy 1.2 was ratified by OASIS in July 2007.
- **Web Services Reliable Messaging(WS-ReliableMessaging)** and its associated PAL (WS-RM Policy) support guaranteed message delivery according to a specified assurance level (AtMostOnce, AtLeastOnce, ExactlyOnce, and InOrder). WS-ReliableMessaging 1.1 was ratified by OASIS in June 2007.
- **Web Services Transaction (WS-Transaction)** is a family of specifications that supports distributed transaction integrity using either 2PC protocols or compensating transactions. The WS-Transaction 1.1 specifications were ratified by OASIS in April 2007.
- **Web Services Notification (WS-Notification)** is a family of specifications that supports brokered and non-brokered notifications and pub/sub patterns using a hierarchical tree-based topic space. The WS-Notification 1.3 specifications were ratified by OASIS in October 2006.
- **Web Services Eventing (WS-Eventing)** supports event-driven notifications using a lightweight, extensible pub/sub pattern. It supports a subset of the capabilities of WS-Notification. WS-Eventing was developed by BEA, CA, IBM, Microsoft, and TIBCO Software. It was submitted to W3C in March 2006, although as of September 2007 W3C had not yet launched a working group to manage the specification.

All ESB vendors support WS-Security and many support WS-Security Policy. Very few support WS-Trust or WS-SecureConversation. Even fewer ESBs can mediate a secure session established using WS-SecureConversation. A few ESB vendors have implemented support for WS-ReliableMessaging, and many others plan to add support for it in late 2007 or early 2008. Currently, most ESBs support reliable messaging using proprietary methods based on MOM. Many ESB vendors have committed to adding support for WS-Transaction in the next release of their products, although very few support it currently. Unfortunately, notification standards are still in flux. Although WS-Notification has been ratified, few products support it. The vendors appear to be committed to WS-Eventing instead. But until W3C launches a working group to manage it, WS-Eventing has no chance of being ratified.

For information about these and other WSF extension standards, see the Reference Architecture Template, "Web Services Framework Standards." For a more detailed discussion of the WS-Security family of specifications, see the *Security and Risk Management Strategies* overview, "Web Services Security Standards 2006: Where Are We Now?."

WSF Programming Models

WSF programming models standards support service construction, composition, and orchestration, as well as data management and presentation. Service construction and composition programming models tend to be language-specific. Examples include the following .NET and Java APIs for web services:

- **ASMX:** Active Server Pages for .NET (ASP.NET) includes an annotated-code programming model and framework called Active Server Methods (ASMX) for building plain old XML (POX) services and web services using SOAP 1.1 and WSDL 1.1.
- **WSE:** The Web Services Enhancements (WSE) framework for .NET augments ASMX, adding support for MTOM, WS-Addressing, WS-Security, WS-SecureConversation, and WS-Trust.
- **WCF:** Windows Communication Foundation (WCF) is a .NET 3.0 annotated-code framework that replaces ASMX and WSE. Although Microsoft doesn't position it so, WCF could be construed to be an ESB. WCF provides a unified programming model and runtime system that enables an application to communicate over a variety of communication protocols using an extensible channel plug-in model. WCF includes plug-in channel modules for POX, SOAP, Microsoft Message Queue (MSMQ), and .NET Remoting. WCF supports SOAP 1.1, SOAP 1.2, MTOM, WS-Addressing, WSDL 1.1, WS-Policy, WS-MetadataExchange, WS-Security, WS-SecureConversation, WS-Trust, WS-SecurityPolicy, WS-ReliableMessaging, and WS-Transaction.
- **JAX-WS:** The Java API for XML Web Services (JAX-WS) defines an annotated-code programming model and framework for building web services. JAX-WS does not support SOAP encoding, a deprecated XML encoding format defined in the SOAP 1.1 specification. JAX-WS 2.1 was approved by the JCP in May 2007, and it is a required component in Java EE. JAX-WS supports POX, SOAP 1.1, SOAP 1.2, MTOM, and WSDL 1.1. JAX-WS defines a handler framework for implementing support for WSF extensions, but standard frameworks and APIs for each specification have not been defined.
- **JAX-RPC:** The Java API for XML-based RPC (JAX-RPC) provides a programming model and framework for building web services. JAX-RPC has been superseded by JAX-WS, although JAX-RPC is still required to support services that use SOAP encoding. JAX-RPC is a required component in Java EE. JAX-RPC 1.1 was approved by the JCP in October 2003.
- **JAX-RS:** The Java API for RESTful Web Services (JAX-RS) will define an annotated-code programming model and framework for building POX services that adhere to REST architectural constraints. JAX-RS is a work in progress at the JCP.

The Service Component Architecture (SCA) defines a language- and protocol-neutral service component model, and it has recently gained significant vendor buy-in. SCA is discussed in the "Service Component Architecture" section of this Technology & Standards document.

The Business Process Execution Language (BPEL) defines a language for orchestrating web services. BPEL is discussed in the "Business Process Execution Language" section of this Technology & Standards document.

Data management programming models enable processing and manipulation of data—particularly XML data. The data management programming models that pertain most to ESBs include:

- **XSLT:** Extensible Stylesheet Language Transformations (XSLT) is a language for transforming an XML document using a style sheet. XSLT 1.0 was ratified by W3C in November 1999. XSLT 2.0was ratified by W3C in January 2007.
- **XQuery:** XML Query Language (XQuery) is a language for querying, merging, and transforming one or more XML documents. XQuery 1.0 was ratified by W3C in January 2007.
- **SDO:** Service Data Objects (SDO) defines a uniform, language-neutral programming model and framework for mapping data to and from object graphs and a variety of data sources, including XML, relational databases, and enterprise information systems. The SDO specifications were developed by an informal vendor consortium called Open SOA whose members include BEA, Cape Clear Software, IBM, Interface21, IONA Technologies, Oracle, Primeton Technologies, Progress Software, Red Hat, Rogue Wave Software, SAP, Siemens, Software AG, Sun Microsystems, Sybase, TIBCO, Xcalia, and Zend Technologies. SDO specifications were published for Java and C++ in November 2006 and December 2006 respectively. Draft specifications for C and COBOL were published in May 2007, and the PHP community has released a PHP implementation of SDO.

Presentation programming models are typically out of the scope of an ESB product. ESBs should support a variety of presentation modes, including desktop clients, web applications, portal servers, mashup servers, rich Internet applications (RIAs), and rich mobile applications (RMAs). ESBs support these presentation modes through a variety of protocol channels.

For information about these and other WSF programming models, see the Reference Architecture Template, "Web Services Framework Standards."

WSF Management

WSF management standards support runtime management and provisioning capabilities. Unfortunately, management standards are woefully immature, and because there are no established standards, ESBs support management and provisioning using proprietary methods.

For information about emerging WSF management standards, see the Reference Architecture Template, "Web Services Framework Standards."

Business Process Execution Language

Many ESBs supply an orchestration engine that can execute structured workflows representing a business process. The Business Process Execution Language (BPEL) is an orchestration language for defining these automated business processes. A process definition specifies the precise set of execution steps that govern runtime interactions among applications, data sources, and other entities required to complete the process. In particular, BPEL defines a syntax for specifying an automatic workflow for invoking web services (services that are described by WSDL). Orchestrations defined by BPEL are then in turn exposed as web services.

Web Services Business Process Execution Language (WS-BPEL) version 2.0 was ratified by OASIS in April 2007, and the ESB vendors are strongly committed to this standard. Currently, though, many ESB vendors implement support for a previous nonstandard version known as Business Process Execution Language for Web Services (BPEL4WS) version 1.1.

In June 2007, a number of vendors, including Active Endpoints, Adobe Systems, BEA, IBM, Oracle, and SAP, published a set of specifications know as WS-BPEL Extension for People (BPEL4People) that enable BPEL process definitions to invoke human workflow activities in a standard way. These specifications will address a glaring omission in the current specification—that of human participation—but they have not yet been submitted to a standards body.

For more information about BPEL, see the *Application Platform Strategies* overview, "Crossing the Divide: The Mechanics of Process Execution."

Service Component Architecture

Service Component Architecture (SCA) is a set of specifications that describes a component model for building and hosting services and an assembly model for building applications based on service composition. SCA is not an orchestration language; it can't be used to define workflows. Instead it defines standards for representing services as components that can be wired together to create an application. Each component takes responsibility for invoking the next component in the sequence rather than relying on an orchestration engine to coordinate the process.

SCA is language and protocol independent. SCA components can be implemented using many languages, such as Java, C++, BPEL, and PHP, and they can communicate using various protocols, such as SOAP, RMI, and MOM. Currently, no one has defined a language implementation binding for .NET, but nothing about SCA precludes such a thing.

The fundamental abstractions in the SCA model are similar to those used in the Spring Framework. The SCA model is depicted in Figure 5. An SCA composite contains one or more components. A component may be implemented using any supported language. A component may allow *properties* (external data values) to influence its operation. Each component exposes one or more *services* (interfaces that may be consumed by applications or other components). A component also typically has one or more *references* (dependencies on other components). When assembling components, a reference must be wired to its associated service. References and services can support one or more *bindings* (invocation mechanisms and protocols). A composite application can in turn be exposed as a component, promoting its internal services, references, and properties for external consumption. A composite configuration, described in a Service Component Definition Language (SCDL) file, describes each component (its implementation, properties, services, references, and bindings), provides values for the properties, wires references to services, and associates policies with the components and wires.

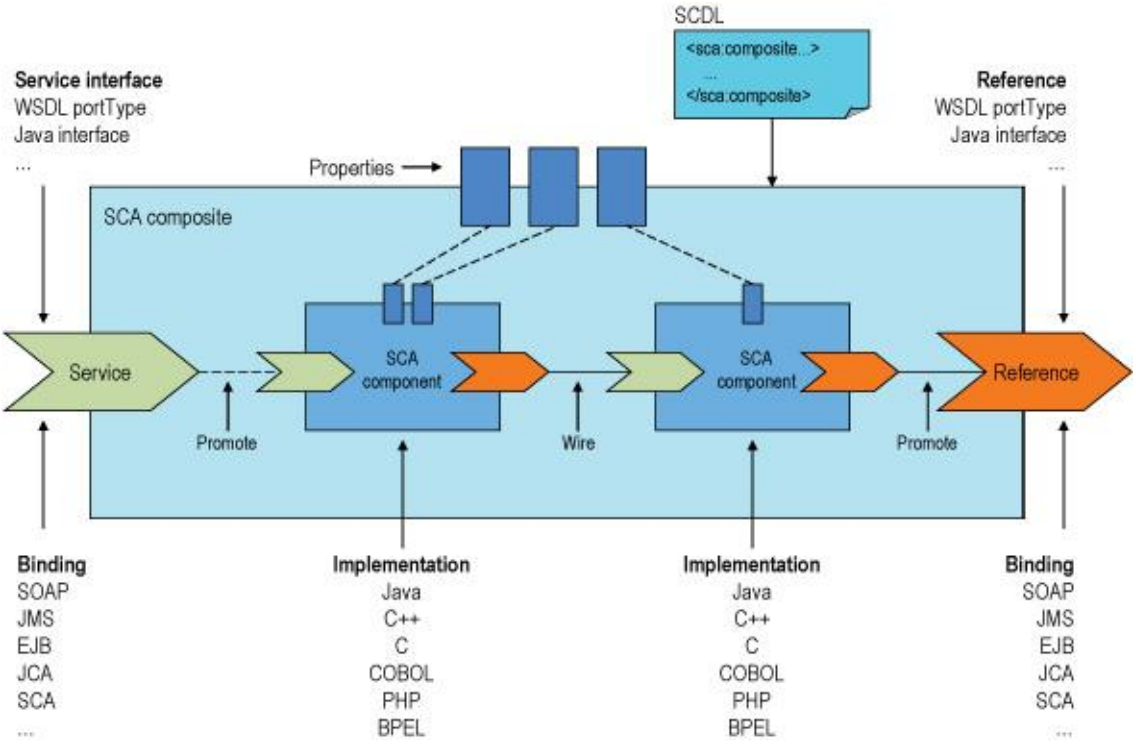

Figure 5: *The SCA Composition Model*

The SCA specifications were developed by an informal vendor consortium called Open SOA, whose members include BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress Software, Red Hat, Rogue Wave, SAP, Siemens, Software AG, Sun Microsystems, Sybase, TIBCO, Xcalia, and Zend. These vendors published the SCA v1.0 specifications in March 2007 and subsequently submitted them to OASIS. Open SOA has since published a few additional draft specifications adding support for C and COBOL component models and a binding for the Java EE Connector Architecture (JCA). Open SOA also plans to publish an additional draft specification defining an SCA component model for Java EE.

The SCA specifications are now managed by the OASIS Open Composite Services Architecture (Open CSA) Member Section, which comprises six technical committees (TCs). Each TC is responsible for one or more of the SCA specifications. As of September 2007, SCA comprised 13 specifications, which include two foundational specifications and define support for five programming languages (Java, C++, C, COBOL, and BPEL) and five bindings (SCA internal binding, web services, JMS, Enterprise JavaBeans [EJB], and JCA).

The two foundational specifications define an abstract SCA component and assembly model and a policy framework:

- SCA Assembly Model v1.0 defines the abstract SCA component and composite assembly model, as well as the SCA internal binding. The SCA internal binding is implementation-specific and can be used for interactions between references and services within a single SCA domain.
- SCA Policy Framework v1.0 defines a framework for associating policies (defined via WS-Policy or other unspecified means) with a service component or wire.

SCA supports multiple programming languages by mapping the abstract SCA component model to language-specific component models. The SCA language-specific specifications include:

- SCA Java Common Annotations and APIs v1.00, which defines a common set of Java annotations and APIs that are used by the other Java-related SCA specifications. These annotations and APIs are used to indicate the properties, services, and references within a component.
- SCA Java Component Implementation Specification v1.00, which describes how to implement an SCA component using a Java class and how to invoke an SCA component from a Java client application.
- SCA Spring Component Implementation Specification v1.0, which describes how to implement an SCA component in Java using the Spring Framework and how to reference an SCA component in a Spring configuration.
- SCA Client and Implementation Model Specification for WS-BPEL v1.00, which describes how to implement an SCA component using BPEL and describes an extension to BPEL that enables a BPEL process to invoke an SCA component.
- SCA Client and Implementation Model Specification for C++ v1.00, which describes how to implement an SCA component using C++ and how to invoke an SCA component from a C++ client application.
- SCA C Client and Implementation Specification v1.00 (Draft), which describes how to implement an SCA component using C and how to invoke an SCA component from a C client application.
- SCA COBOL Client and Implementation Specification v1.00, which describes how to implement an SCA component using COBOL and how to invoke an SCA component from a COBOL client application.

In addition to these specifications, the PHP community has launched an open source project called SOA PHP that includes a prototype implementation of SCA for PHP. IBM also provides prototype support for SCA components implemented with Ruby in IBM WebSphere Process Server.

SCA supports multiple invocation mechanisms and protocols via binding specifications. A service uses a binding to expose its capabilities to client applications and components. A reference uses a binding to describe the mechanism used to invoke its associated service. The SCA Assembly Model specification describes a vendor-specific binding (SCA internal binding) that can be used to wire references to services within a single SCA domain (a single instance of an SCA runtime environment). The additional SCA binding specifications include:

- SCA Web Service Binding Specification v1.00, which enables seamless interoperability between SCA components and web services that communicate using SOAP 1.1 or SOAP 1.2. The web services binding allows an SCA component to expose its capabilities as a web service or to reference and invoke a web service.
- SCA JMS Binding Specification v1.00, which enables seamless interoperability between SCA components and JMS applications. The JMS binding allows an SCA component to send and receive messages via a JMS queue or topic.
- SCA EJB Session Bean Binding v1.00, which enables seamless interoperability between SCA components and EJB applications. The EJB binding allows an SCA component to expose its capabilities as an EJB session bean or to reference and invoke an EJB session bean.
- SCA JCA Binding Specification v1.00, which enables seamless interoperability between SCA components and legacy applications accessible through a JCA adapter. The JCA binding allows an SCA component to expose its capabilities as a JCA resource or to reference and invoke JCA resources.

SCA has garnered wide support from ESB vendors and is likely to emerge as the prominent component and assembly model standard for ESBs. For a more in-depth overview of SCA, see the informative "Introducing SCA " white paper by David Chappell.

Java Business Integration

Java Business Integration (JBI) is a standard Java API that defines a plug-in framework for integration infrastructure systems. It defines a standard set of system-level interfaces that enable a JBI-compliant container or resource adapter to be plugged into any JBI-compliant integration system (such as an ESB). JBI is similar in concept to JCA except that it does not have a dependency on Java EE.

The fundamental concepts of the JBI framework are shown in Figure 6. JBI acts as an intermediary that routes messages between service endpoints, and it relies on a single common format to enable heterogeneous interoperability. The core of the framework is the Normalized Message Router (NMR). All messages routed by JBI are first converted into a normalized format that separates routing and QoS information (referred to as the message context) from the application payload. Service endpoints are accessed through plug-in components that communicate with the NMR using the JBI system-level APIs. JBI describes two types of plug-in components:

- **Service engines** are containers that host service endpoints. Example service engines include Java containers for plain old Java objects (POJOs) and EJBs, a BPEL engine, and an engine that executes XSLT scripts. JBI does not specify any details regarding the component models for these containers.
- **Binding components** are connectors to remote service endpoints. A JBI-compliant system must provide a binding component for SOAP. Binding components could also provide access to endpoints that communicate using JMS, JCA, CORBA, or any other protocol.

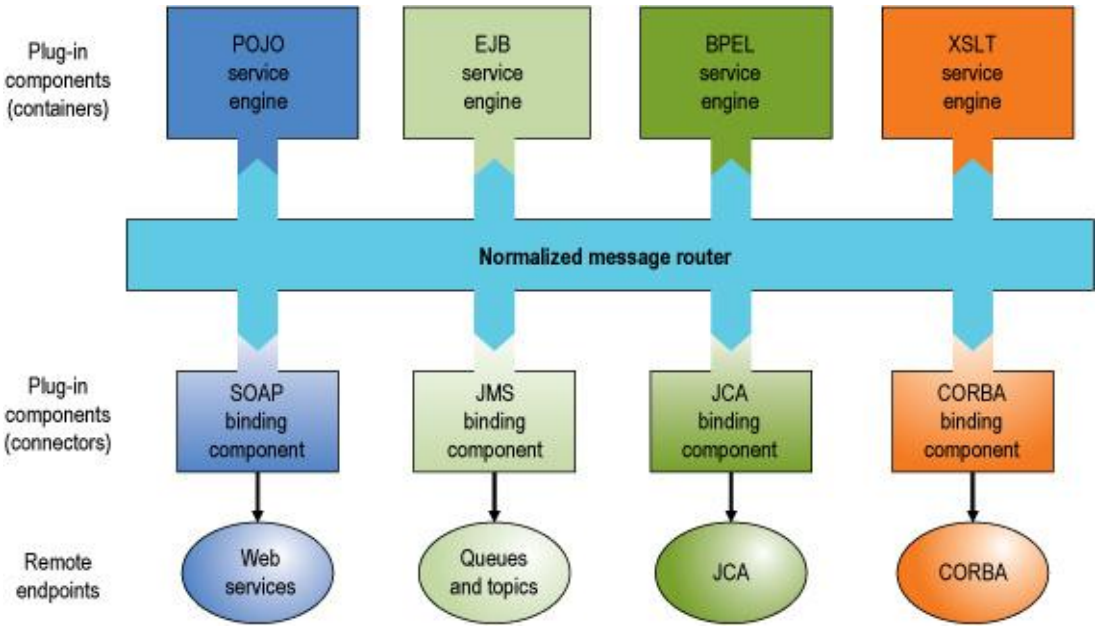

Figure 6: *JBI Framework*

JBI 1.0 was developed through the Java Community Process (JCP) and approved as a standard Java API in June 2005. Sun Microsystems also sponsors an open source community website, Open JBI Components, which has the goal of fostering an ecosystem of pluggable JBI components. JBI 1.0 is supported by a number of smaller ESB players, but it has not gained the support of the Java superplatform vendors, including BEA, IBM, Oracle, and SAP. A new Java Specification Request (JSR 312) was launched in March 2007 to define JBI 2.0, although BEA and IBM voted against the new effort.

Conclusion

An enterprise service bus (ESB) is a middleware solution that enables interoperability among heterogeneous environments using a service-oriented model. Although frequently associated with concepts like “integration” and “mediation,” an ESB also provides a service platform comparable to an application server. In fact, ESBs represent the consolidation of the integration and application server product categories. An ESB's true breakthrough feature is its ability to virtualize services. An ESB's service container abstracts a service and insulates it from its protocols, invocation methods, method exchange patterns, quality of service requirements, and other infrastructure concerns.

Notes

[1] "WebSphere software: ESB Overview." *IBM*. Accessed online 21 Aug 2007. http://www-304.ibm.com/jct03001c/software/info1/websphere/index.jsp?tab=integration/esb&.

[2] Anne Thomas Manes, Richard Monson-Haefel, Joe Niski, Lyn Robison, Chris Howard. "VantagePoint 2007–2008: Build for Today, Architect for Tomorrow." *Burton Group*. 30 Mar 2007. http://www.burtongroup.com/Client/Research/Document.aspx?cid=1058.

Author Bio

Anne Thomas Manes

Vice President and Research Director

Emphasis: Service-oriented architecture, web services, XML, governance, superplatforms, application servers, Java, J2EE, .NET, application security, data management

Background: Former Chief Technology Officer at Systinet, a SOA governance vendor (now part of HP). Director of Market Innovation in Sun Microsystem's software group. Manes' 28-year industry background also includes field service and education at IBM Corporation; customer education at Cullinet Software; product management at Digital Equipment Corporation; chief architect at Open Environment Corporation; and research analyst with Patricia Seybold Group.

Primary Distinctions: Named one of the 50 most powerful people in networking in 2002 by Network World. Listed among the "Power 100 IT Leaders," by Enterprise Systems Journal. A frequent speaker at trade shows and InfoWorld, JavaOne, and RSA conferences. She has also authored numerous articles in trade publications. Member of Web Services Journal editorial board. Authored "Web Services: A Manager's Guide," published by Addison-Wesley, 2003. Participated in web services standards development efforts at W3C, OASIS, WS-I, and JCP. Anne earned a BA in Economics at Wellesley College.