

The Enterprise Service Bus: Making service-oriented architecture real



M.-T. Schmidt
B. Hutchison
P. Lambros
R. Phippen

The Enterprise Service Bus (ESB) is the infrastructure which underpins a fully integrated and flexible end-to-end service-oriented architecture (SOA). This paper details the essential meta-data and capabilities of the ESB. It presents a summary of the key concepts of the ESB and defines the integration model for it, including key user roles. These roles are fulfilled using meta-data that describes the service endpoints, such as the service interface and policy requirements and capabilities. The ESB manages this meta-data through a registry, which supports configuration, connection, matchmaking, and discovery of service endpoints. Some typical mediation patterns that are used to satisfy endpoint policies are explored, and usage patterns are described in which the ESB is used to implement real SOAs.

INTRODUCTION

Many papers in this journal discuss service-oriented architecture (SOA)—what it is and its benefits and value propositions. Other papers describe tangible implementations of SOAs. This paper abstracts and learns from these and similar experiences to identify the essential characteristics of an Enterprise Service Bus (ESB): the meta-data that describes service requestors and providers, mediations and their operations on the information that flows between requestors and providers, and the discovery, routing, and matchmaking that realize a dynamic and autonomic SOA.

In particular, this paper explains how the ESB provides the tools and runtime infrastructure to realize the promise of SOA formulated in the iconic “publish-find-bind” triangle (see *Figure 1*) that was

popular in the early days of the SOA revival caused by Web Services (see Reference 1). As illustrated in Figure 1, the ESB manages and exploits meta-data describing interaction endpoints as well as the domain models used to describe the capabilities of those endpoints; it supports configuration of links that bridge between capabilities demanded by service requestors and those offered by service providers, dynamically matching requestors with providers and in the process establishing and enacting contracts between those interaction endpoints.

©Copyright 2005 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/05/\$5.00 © 2005 IBM

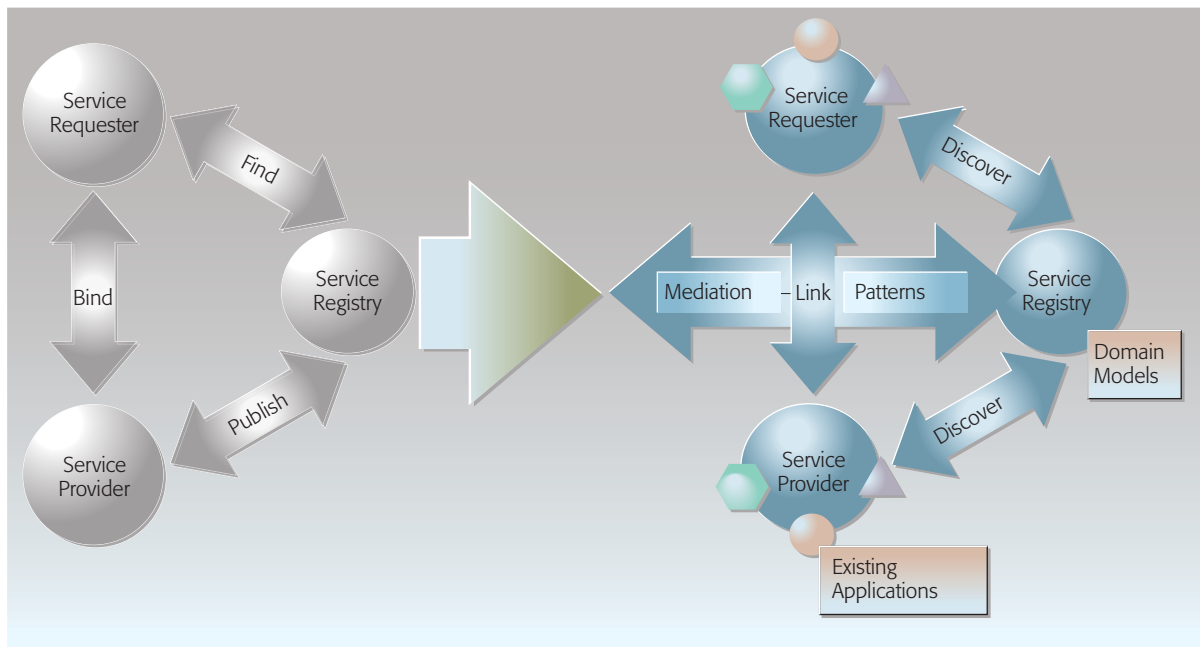


Figure 1
ESB underpinnings for SOA

This paper is structured as follows: we begin with an overview of the main characteristics of the ESB, followed by a detailed discussion of the concepts of the ESB programming model and the standards supporting those concepts. We conclude with a description of ESB use cases and usage patterns.

ESB IN A NUTSHELL

The ESB enables an SOA by providing the connectivity layer between services. The definition of a service is wide; it is not restricted by a protocol, such as SOAP (Simple Object Access Protocol) or HTTP (Hypertext Transfer Protocol), which connects a service requestor to a service provider; nor does it require that the service be described by a specific standard such as WSDL (Web Services Description Language), though all of these standards are major contributors to the capabilities and progress of the ESB/SOA evolution. A service is a software component that is described by meta-data, which can be understood by a program. The meta-data is published to enable reuse of the service by components that may be remote from it and that need no knowledge of the service implementation beyond its published meta-data. Of course, a well-designed software program may use meta-data to define interfaces between components and may reuse components within the program. The distin-

guishing feature of a service is that the meta-data descriptions are published to enable reuse of the service in loosely coupled systems, frequently interconnected across networks.

What do we mean by “publishing” a description of a service? Descriptions of the services available from a service provider can be made accessible to developers at the service requestor, possibly through shared development tools. The ESB formalizes this publication by providing a registry of the services that are available for invocation and the service requestors that will connect to them. The registry is accessible both during development and at runtime. Components such as J2EE** EJBs** (Java** 2 Enterprise Edition Enterprise JavaBeans**) or database-embedded functions may be published as services, but not every J2EE EJB is a service, and not every J2EE EJB is accessible by means of the ESB. In general, EJBs need additional meta-data, and possibly additional bindings, published to the ESB registry in order to make them available as services.

Publication of the service requestors and providers allows their meta-data to be administered through the ESB registry and enables their relationships and interactions to be visualized and updated. Nonetheless, ad hoc requestors and providers may also

connect to the ESB without first being registered, for example, subscribers to a “publish/subscribe” topic. In that case, their interactions will not benefit from the full dynamic capabilities of the ESB, described later.

The ESB populates the registry with meta-data about services in three different ways. When services are deployed to the runtime environment, they can be simultaneously and dynamically added to the ESB; meta-data associated with components already deployed can be explicitly added to the ESB; or the ESB can discover services and service interactions that are already deployed and incorporate meta-data describing them in the registry.

Note that the ESB is the infrastructure for inter-connecting services, but the term ESB does not include the business logic of the service providers themselves nor the requestor applications, nor does it include the containers that host the services. Hosting containers and free-standing applications are enabled for interaction with ESBs with varying levels of integration, depending on the range of protocols and interoperability standards supported. Most containers (e.g., J2EE application servers, CICS*, Microsoft .NET**) integrate with an ESB across the SOAP/HTTP protocols, but fewer have direct support for SOAP/JMS (Java Messaging Service) over a particular brand of JMS provider. After the ESB has delivered its payload to a container, its responsibilities are fulfilled. Within the container, the service invocation may be redirected among the machines in a cluster, or it may be responded to from a local cache. These are some of the normal optimizations within an application server environment, and they complement the routing and response capabilities of the ESB between the service providers it interconnects. Similarly, the ESB is the connectivity layer for process engines that choreograph the flow of activities between services. The process engine is responsible for ensuring that the correct service capabilities are scheduled in the correct order. It delegates to the ESB the responsibility for delivering the service requests, rerouting them if appropriate.

A core tenet of SOA is that service requestors are independent of the services they invoke. As a result, it is not surprising that the ESB is essentially invisible to the service requestors and providers that use it. A developer can use an API (application

programming interface), such as JAX-RPC (Java API for XML-based RPC [remote procedure call]) to a Web service, or distribute messages with the WebSphere* MQI (Message Queue Interface) to a message queue, without considering whether these requests are flowing directly to the service or are traversing an ESB. Similarly, a service provider can be written as a J2EE EJB or a servlet without any specific application code to make it accessible through an ESB. Despite this, one of the values of the ESB is that it takes on the responsibility for many of the infrastructure concerns that might otherwise surface in application code. Thus, although developers can use APIs for service invocation, they do not need to add logic to deal with security, for example.

The ESB virtualizes the services that are made available through the bus. The service requestor, both in its application logic and in its deployment, does not need to have any awareness of the physical realization of the service provider. The requestor does not need to be concerned about the programming language, runtime environment, hardware platform, network address, or current availability of the service provider's implementation. In the ESB, not even a common communication protocol need be shared. The requestor connects to the bus, which takes responsibility for delivering its requests to a service provider, offering the required function and quality of service. Not surprisingly, the infrastructure of the bus is itself virtualized, allowing it to grow or shrink as required by the network and workload which it is supporting.

The flexibility that comes from an SOA, and the virtualization it implies, is fully realized by the dynamic nature of the ESB. All the meta-data, conditions, and constraints used to enable a connection from a requestor to a provider can be discovered, used, and modified at runtime. For example, a new implementation of a service in a different geographical region can be published to the ESB registry, and requests in that region can be routed to it without reconfiguration of the requestors. A service requestor might select a reduced level of assured delivery and see an improved level of performance as the ESB determines that it can use a different delivery protocol. This flexibility is available as a direct consequence of the role of the ESB registry. Because all relevant meta-data for the service provider and service requestors has been

placed in the ESB registry, it can be subsequently discovered and used to make dynamic changes.

To achieve much of this flexibility, the ESB accepts requests as messages, then operates on them, or “mediates” them, as they flow through the bus. Mediations can be an integral part of the ESB, providing (for example) transport mapping between SOAP/HTTP and SOAP/JMS, or routing a message to an alternate provider if response times fall below an acceptable value. It is also a feature of the flexibility of the ESB that mediations can be provided by third parties—by other products, ISVs (independent software vendors), or customers—to operate on messages as they flow through the ESB infrastructure. This allows, for example, ISV packages to implement advanced load-balancing features among services, or customers to add auditing to meet new legislation. Mediations can be deployed on the ESB without changing the service requestor or provider.

Mediations are the means by which the ESB can ensure that a service requestor can connect successfully to a service provider. If a service provider requires one format for an address field and a service requestor uses a different one, a mediation can map from one format to another so that the ESB can deliver the service request. If the service provider expects encrypted messages, a mediation can encrypt the “in-the-clear” service requests as they pass through the ESB. The ESB can react dynamically to the requirements of requestors and providers when they are described in their meta-data and held in the registry. In the case of message formats, this is usually achieved through a schema definition. For other service properties, policy statements, which may describe the encryption algorithms to be used or the requirements for auditing, can be associated with the meta-data of the service provider and requestor. The ESB consults this meta-data at runtime and can reconfigure the mediations between requestor and provider to match the requirements. By annotating a policy for the service providers in the ESB registry, the system administrator can, for example, ensure that the services meet the company’s new privacy guidelines. Thus the ESB implements an autonomic SOA, reacting to changes in the services it connects.

One of the major uses of mediations is in systems management. Mediations can be deployed in the ESB environment to enable request and response

messages to be monitored as they flow through the system, enabling service-level management or problem determination. Mediations can route service invocations to back-up data centers if there is a local problem or to new service providers as they are brought online. They can validate messages in terms of their format correctness, data values, or user authentication and authorization. Through these and other systems management capabilities, the ESB ensures that a loosely coupled and dynamically varying SOA is still manageable in a production environment.²

Many of the mediation capabilities just described are core attributes of the ESB, and the mediations are made available as part of the runtime environment. They are customizable, so that, for example, a generic table-driven routing mediation can be configured to use a specific table and a specific field in a message as the key. The ESB also provides tools to configure the interactions between services—to display the services available in the ESB, to interconnect them, to add policy requirements to a service or group of services, to identify mismatches in the endpoints, and to associate mediations to correct these, either explicitly or through automatic reconciliation of their policy declarations.

Much of the preceding discussion uses the terms *service requestor* and *service provider*, as is appropriate for the ESB. Service requestors and service providers are equal partners in the interaction, with the requestor simply being the endpoint that initiated the interaction. The interaction may continue with either endpoint sending or receiving messages. The ESB supports many different types of program interaction: one-way messages as well as requests and responses, asynchronous as well as synchronous invocation, the publish/subscribe model, where multiple responses may be generated for one subscribe request, and complex event processing, where a series of events may be observed or consumed to produce one consequential event. The ESB is also, in principle, transport and protocol “agnostic,” with the capability to transform messages to match the requestor’s preferred formats to those of the provider. In practice, most ESBs support SOAP/HTTP, which reinforces its role as an interoperability standard. They also support a range of other transports and protocols, some for use by service requestors and providers connected by the

ESB, and some for internal communication within the ESB.

THE ESB INTEGRATION MODEL

Having established the basic concepts and features supported by the ESB, we focus on ESB-based SOA solutions. The ESB *integration model* captures those aspects of the overall solution that are relevant for the ESB tools and infrastructure to facilitate interactions between ESB-managed service endpoints. It enables the various user roles involved in creating and managing those solutions to express the ESB-relevant information that they contribute or monitor, and the ESB runtimes to manage interactions accordingly. In essence, the model contains meta-data describing service endpoint requirements, capabilities, and relationships, including information describing the specific details of interaction contracts.

Not all user roles use the model directly. A business analyst, for example, might define a set of key performance indicators (KPIs) that need to be translated into events produced by the underlying implementation artifacts and potentially into parameters of a service interaction contract. Users in the architecture and design space, such as a solution architect, might define service capabilities and requirements in more abstract terms than those required by the ESB integration model. Other roles, such as the application developer, create and use service capabilities and requirements for the integration model. The user roles that are the most interesting from the ESB perspective are those of the integrator and the solution administrator. We next discuss how integration specialists assemble solutions from existing service components and how solution administrators configure and reconfigure those solutions. We use UML** (Unified Modeling Language**) models to illustrate the concepts those user roles deal with; these are high-level conceptual models not to be confused with product- or implementation-specific models of the underlying runtimes or specific standards in this space. We do, however, hint at relationships to products and standards where appropriate.

Integration specialists assemble business solutions from a set of service components. They do not have to understand the implementation details of those components (process coordination, existing applications, interactive tasks, etc.); all they need to

understand is the capabilities offered by components and the requirements of the components they use, with respect to other components. The ESB service registry provides the required information about those components and, together with the ESB runtime, enables integration specialists to perform component-assembly tasks, selecting components required to implement the solution, resolving dependencies those components might have on other components, and interposing the mediations required to make components interact.

Solution administrators deploy and customize the solutions they get from their integration specialist colleagues: they may be given a set of component relationships which they simply adopt; they may choose to override the defaults defined by an integration specialist; they may have to compensate for the fact that an integration specialist has not resolved certain variables of a solution; or they may have to reconfigure a previously deployed solution due to changes in the solution environment. These tasks are usually done in the component development environment, but the ESB enables flexible configuration through late binding by providing this service to the solution-administrator role. The key to enabling this flexible configuration and reconfiguration of solutions is the explicit declaration of capabilities and requirements of service interaction endpoints. The next section explores the underlying service meta-data management capabilities provided by the ESB in more detail.

WHAT PROVIDERS OFFER AND REQUESTORS WANT—SOA META-DATA

The key to service virtualization and dynamic matchmaking between service requestors and providers is the explicit declaration of capabilities and requirements of interaction endpoints. Service meta-data describes capabilities of software assets independent of their implementation specifics. It does not assume a specific programming model for the realization of the services that offer their capabilities for use by other components, nor for the realization of services that require certain capabilities to be provided by other components. It facilitates interoperability among a broad spectrum of service providers and requestors; an existing CICS/COBOL application can declare its service capabilities and expectations exactly like a business service newly implemented in J2EE or a Web service offered by a business partner that uses SOAP/HTTP. Service

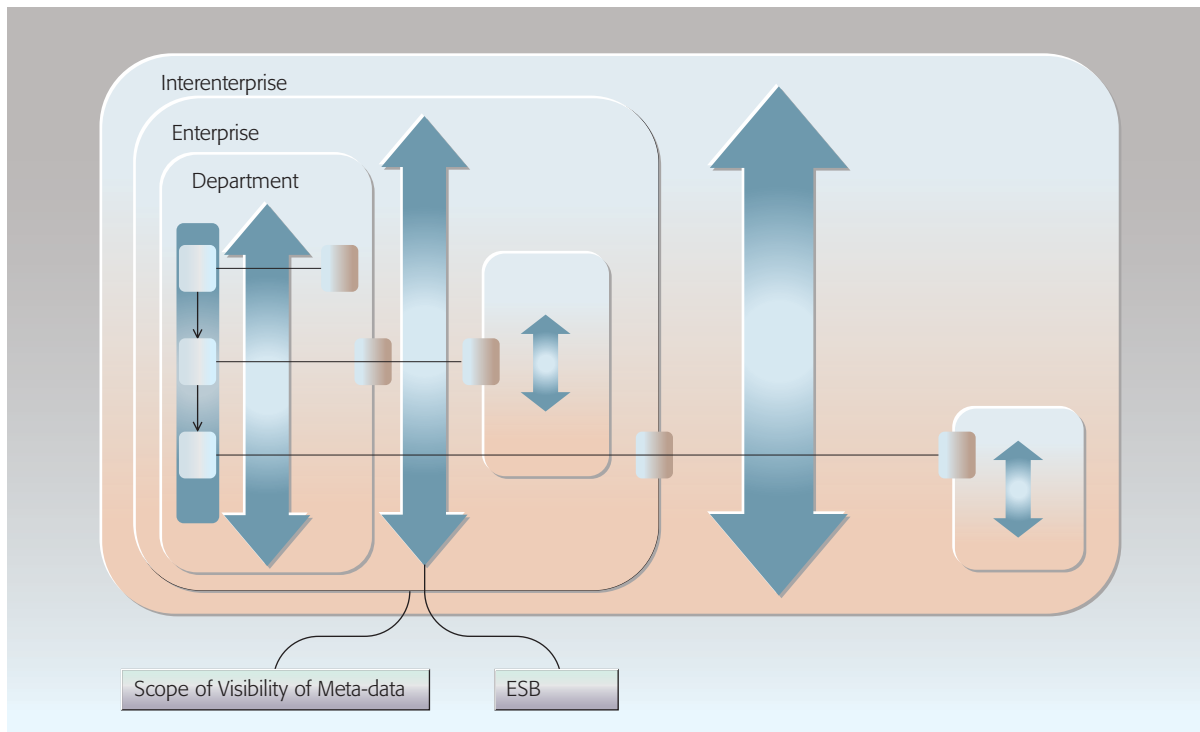


Figure 2
Scope of service assembly and service meta-data visibility

meta-data also supports an “assembly-from-parts” model for implementing new business process applications from a catalog of services.

Before discussing the kind of meta-data about a service that is relevant for ESB-managed interactions in detail, it is important to note that requirements for explicitly recording meta-data about service interaction endpoints are relative to the scope of visibility of the underlying application artifacts (see *Figure 2*). Even in moderately homogenous development environments (e.g., departmental integration projects using messaging features of an application server) it can make sense to use the service component abstraction and an ESB to construct solutions. In that case, a minimal set of meta-data declarations (e.g., interface declarations only) for each service component is sufficient to enable ESB-facilitated service component assembly. If the principles of SOA are applied on a larger scale (on an enterprise level, using a message broker, for example), more explicit declaration of service component capabilities and requirements are necessary. In this case, not only service interfaces but

also quality-of-service assumptions might have to be considered to enable service users to understand under which circumstances they can use a service. An even more detailed declaration of service capabilities is required when attempting to apply SOA on an inter-enterprise scale. In this case, declaration of interaction patterns, service-level agreements, and other factors become relevant. Figure 2 also illustrates the fractal (self-similar) nature of the ESB. Various ESB agents collaborate to realize an integration infrastructure that enables a business process created in a departmental context to interact with service that may be local to the department, hosted in other departments of the same enterprise, or even hosted by other enterprises.

Having established the relativity of meta-data needs for various levels of ESB elements, we take a more systematic look at the spectrum of meta-data that could be recorded for a service component. As established earlier, the main elements of the ESB programming model are service requestors and service providers. Most of the ESB-relevant meta-

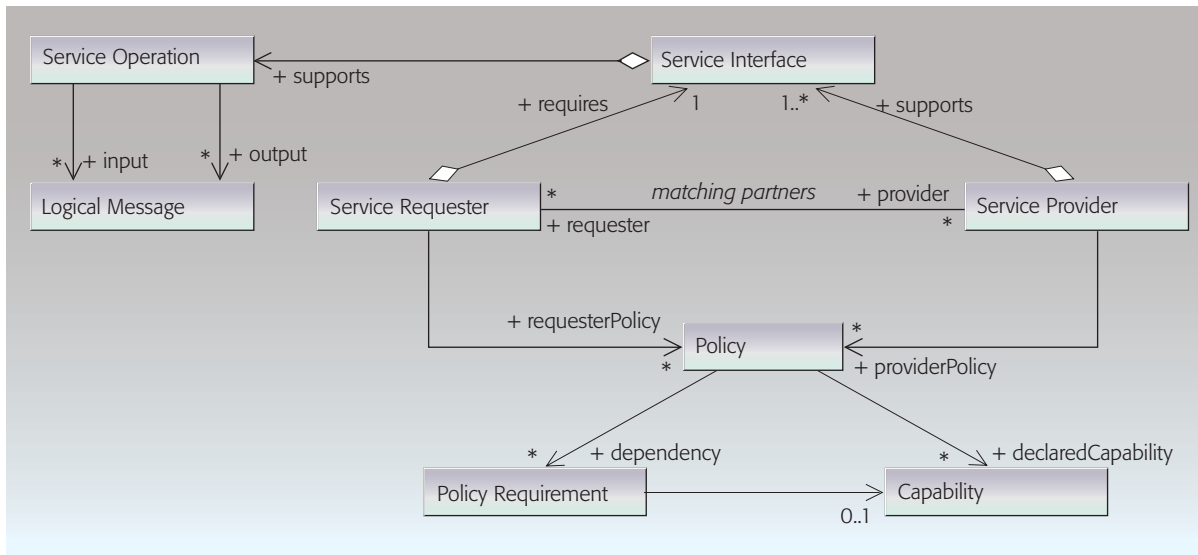


Figure 3
Service capability and requirements declaration

data about those interaction endpoints can be classified as interface declaration or policy annotation.

The objective in recording meta-data is the establishment of matches between potential partners so that they can interact. A given requestor matches a particular provider if their interface and policy declarations are compatible or can be made compatible. The simplest (and in the real world, the rarest) form of compatibility is a perfect match. In this ideal case, a requestor needs the exact operation a provider offers with the qualities of interaction service defined by the provider's policies. The only practical way to make this work is the "requestor makes it right" principle that is common practice in traditional RPC-style interactions. Using this principle, the application developer of a requesting application consults the ESB service registry to find a provider and implements the requestor to fit the provider's specification. In SOA scenarios, a more flexible matchmaking model is required. The ESB supports such a model, but this comes at a price; namely, potential interaction partners need to provide sufficient information about their requirements and capabilities.

The simplest and most common form of service meta-data is the interface declaration. Interaction endpoints describe the messages they can process or

will produce, as well as the message exchange patterns supported. In **Figure 3**, service providers support and service requestors require support for service interfaces that are made up from service operations, which in turn represent message interchanges in terms of logical messages supported by the endpoint. Ideally, the interface part of a declaration of service capabilities and requirements is described by using WSDL, with XML schema describing the structure of messages to be exchanged. Nevertheless, there are many examples of successful ESB implementations that use only a subset of WSDL (e.g., XSDs [W3C** XML Schema definition language] for message declarations for processing information, often with specific annotations to capture additional meta-data about message formats) or home-grown meta-data schema to capture the information. Those approaches, in general, do not scale beyond relatively homogenous, "localized" ESBs, but they support the point that it is more important to make the capabilities and requirements of services explicit rather than use a specific declaration formalism.

Policy declarations further qualify capabilities of interaction endpoints; simply put, a policy expresses anything a component wants the world to know about it other than what messages it understands. This is a very general concept of policy, and it

actually covers things that are sometimes factored out into behavior declaration or semantic annotations of services; as explained previously, we present a very coarse-grained, conceptual model here that may well be refined during translation into an operational model.

As discussed previously, in many localized ESB scenarios, policy declarations are not required. In the general case, interaction endpoints use policies to declare the capabilities they offer or policy requirements they have with respect to other services. Note that the model illustrated in Figure 3 allows both service providers and requestors to declare capabilities and requirements by use of a policy. A requestor can declare not only requirements for providers that wish to interact with it, but also capabilities which specify actions that it is willing to perform before it dispatches a request to a provider (e.g., encrypting messages). Service providers use policies to declare not only the capabilities they offer, but also the requirements they might have for requestors that want to use them. More precisely, policy declarations can provide (1) information about characteristics of the declaring component's internal behavior which affect the interaction, (2) constraints on a peer's invocation of the declaring service, (3) constraints on the target interface of an interaction partner, or (4) information about characteristics of the service component's internal behavior regarding the respective interface reference. The WS-Policy³ declaration establishes a framework for policy declarations of all sorts and enables reasoning about the compatibility of policies declared by potential interaction partners.

The policy declarations discussed here should not be confused with policy declarations intended for the service container that hosts the service component in question, which are outside the scope of this discussion. As indicated earlier, we use a very broad definition of policy and include declarations a service requestor or provider might want to make about expected or supported behaviors. For example, a provider might indicate sequencing constraints on its operations (e.g., operation A has to be invoked before B or C can be invoked), describe interaction patterns involving more than one operation (e.g., if you send me a request on operation A, then I will respond with a message on outbound operation B and expect you to respond with another request to my operation C), or maybe even describe

more than one interaction partner (e.g., if you send a request to my operation A, then I will send a request to my partner P, and after I hear back from him, I will send a response to you). Application developers can represent such behavioral specifications, for example, using the features for abstract process declaration provided by BPEL (Business Process Execution Language).⁴ Note that they will in general not expose the actual behavior of their services (e.g., an executable BPEL process) but rather a projection or abstraction of that behavior.

In the very broad definition of policy here, we also include semantic annotations of service interfaces that explain the meaning of messages exchanged with the service or the meaning of the service operations. Those annotations are useful when the semantics of services and their operations is not obvious from the naming conventions used. The annotations can be as simple as relating elements of the service declaration to well-known terms, but they can also include declarations of preconditions or postconditions for operations. Work on standardizing semantic annotations is ongoing (see References 5 and 6).

We describe a conceptual model for the ESB, not a particular implementation. Service interfaces can be captured in a variety of ways, and from an abstract ESB perspective, the specific syntax used to describe them is far less important than the fact that they are recorded. Declaration of message sets using XML Schema, with annotations to capture information relevant to a specific message formatting, is an example of ESB-managed meta-data about interaction endpoints (in this case WebSphere MQ applications that produce and consume those messages). Where possible, however, we encourage use of the WS (Web Services) standards to declare capabilities and requirements of interaction endpoints: WSDL for service interface declarations, WS-Policy for any kind of policy annotation to those interfaces, and BPEL for specification of sequencing constraints.

Standards play an important role in advancing the syntactic normalization of meta-data about interaction endpoints. Nevertheless, in many cases, especially in ESB scenarios beyond departmental applications, deeper understanding of the semantics of the underlying services is required to perform any meaningful matchmaking. Service interfaces abstracted from a legacy application might make sense

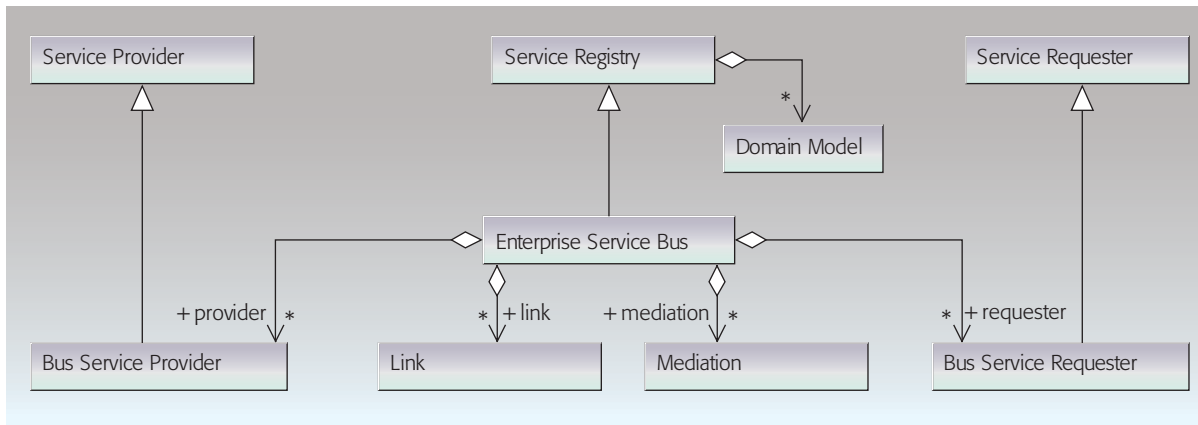


Figure 4
ESB service registry content

to requestors that know about the original application, but not to requestors outside of that small circle; often even a “perfect match” between services on an interface level does not imply that they can actually interact in any meaningful way. As indicated earlier, the semantics of services is often declared through reference to more commonly known concepts, captured in domain models. Domain models establish a frame of reference that makes it possible not only to explain the semantics of services but also to make statements about compatibility of services, and that in turn enables the definition of service interaction contracts which can be managed by the ESB infrastructure. We discuss these concepts in more detail in the next section.

THE ESB SERVICE REGISTRY

Simply put, the ESB plays two main roles in the service endpoint matchmaking game—the service registry manages all relevant meta-data about interaction endpoints (see *Figure 4*), and it also takes care of the matchmaking between those endpoints. This section discusses the service registry; the next section discusses the matchmaking.

The ESB is a service registry in the sense that it manages not only meta-data about the service interaction endpoints involved in the SOA, but also information about domain models. This information establishes a common understanding of services beyond the scope of visibility of an individual service requestor or provider.

The ESB captures information that can be used to better understand the practical content of the registry: domain models representing general knowledge about a topic area, independent of the specific domain applications represented as services in the registry. As before, we use a very generic definition of the term domain model because we want to establish an implementation-independent model for the ESB. Our definition covers domain models as simple as a topic space or a simple taxonomy that classifies events exchanged in publish/subscribe style interactions; it includes standard message sets used in specific industries or a set of “generic business objects” covering a specific application domain; and it extends to moderately complex ontologies describing concepts and their relations in a particular topic space. In the ESB integration model, domain models are used to establish semantics of the practical meta-data artifacts that the ESB cares about.

In many “local ESB” scenarios, little or no domain knowledge needs to be formalized—the user community involved simply knows the semantics, and a simple hint in the form of well-named interfaces and messages suffices. In the publish/subscribe messaging model, topic spaces can be used to classify message instances. Generic business objects or standard message sets applicable to a domain can be used to establish a (semantically) normalized view for messages exchanged within that domain (but usually not beyond it), thus enabling application developers to implement endpoint applications

without worrying about the specific semantics of the underlying service components. This enables service-level managers to observe the status of a system, based on knowledge about events which adhere to the Common Base Events (CBE) standards proposal (see Reference 7).

Ontologies can establish a deeper semantic understanding of the interaction endpoints and facilitate use of more sophisticated ways to identify possible interactions between them (see Reference 8). The main objective of managing domain models and establishing semantic understanding of interaction endpoints is to enable matchmaking between those endpoints—the more explicit knowledge there is about capabilities and requirements of the endpoints, the more automated the matchmaking can be.

The main role of the ESB registry is to manage meta-data about the interaction endpoints themselves. To participate in ESB-managed interactions, endpoints need to register with the ESB. The ESB model represents registered service requestors as *bus service requestors* (BSRs) and registered service providers as *bus service providers* (BSPs). Service providers that are not registered as BSPs are invisible to the ESB for interaction partner selection. When a BSP or BSR is registered, its service interface and policy declarations are captured in the ESB service registry. At this point it is also possible to provide additional information about the service endpoint that might not have been provided with the original declaration. Semantic annotation is one example. Documenting or discovering relationships between the newly registered service and other artifacts in the service registry is another.

Both BSPs and BSRs can be created without any available “counterpart” that wishes to interact with them; conceptually, the ESB ensures that it will take care of connecting each BSP to requestors that send requests to it and that it will deliver the requests of each BSR to some matching provider. In a way the ESB is the “ideal provider” for a BSR and the “ideal requestor” for a BSP. It is the ESB’s job to make things right if there actually should not be such an “ideal counterpart.”

The ESB registry also holds details of links and mediations, which are described in the next subsections.

Like any good service registry, the ESB provides the following features:

- Discovery and management of meta-information about interfaces and capabilities of existing applications that can be used as building blocks for integration solutions. This includes analysis of legacy applications to discover meta-information about their interfaces, policies, and behavioral constraints, as well as exploitation of object discovery agents to capture meta-information about packaged applications.
- Management of meta-information about services. This includes WSDL declarations as well as WS-Policy declarations describing capabilities provided by services or required by service requestors and also BPEL-defined declarations of behavioral constraints for services (abstract BPEL processes) or actual behavior of those services.
- Management of domain models describing general knowledge about an application domain relevant for SOA-based business integration scenarios. Examples include industry-standard message sets, generic business objects, ontologies encoded in the OWL Web Ontology Language, and “contracts” for SOA interactions.
- Discovery and management of relationships between “real world” artifacts representing existing applications, service declarations, and domain models.
- Enabling generation of artifacts for the ESB runtime. Examples include Service Data Object⁹ (SDO) schema, maps between service interfaces for ESB mediations, and application adapters.
- Management of runtime meta-information for matchmaking between service requestors and service providers (e.g., service declarations with policies and compatibility rules) and SDO schema with annotations.

Links and mediations for dynamic SOA

The ESB supports two concepts to facilitate interactions between endpoints: it introduces *links* between service requestors and providers that enable basic connectivity between interaction endpoints with a configurable quality of service, and it provides the concept of *mediations* that can be used to configure and reconfigure the ESB by dynamic alterations to routing and qualities of service and to allow interaction endpoints to modify their behaviors. Links and mediations basically realize the contract between interaction partners that is implicit

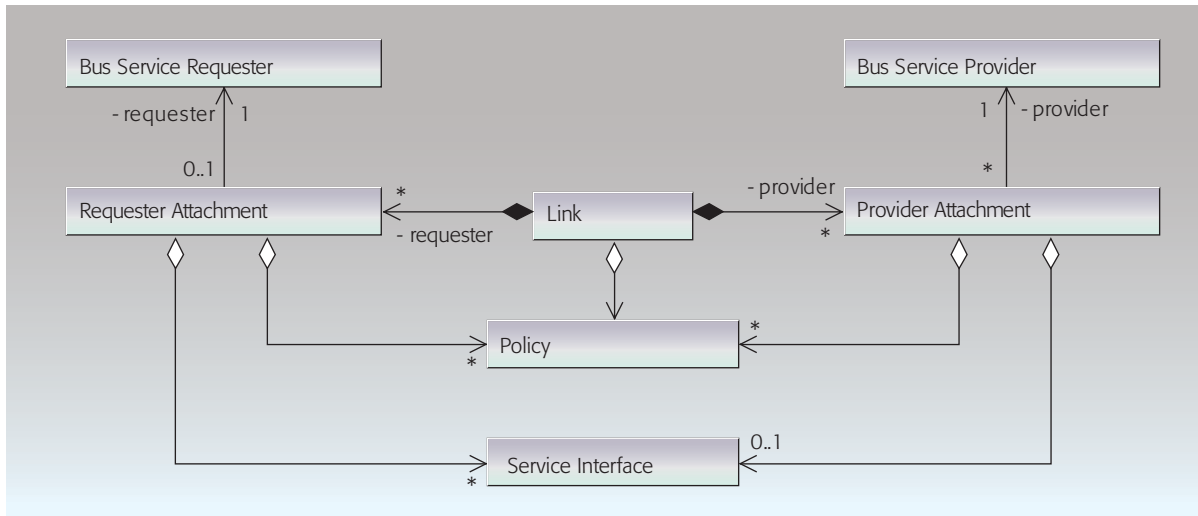


Figure 5
ESB links

in the declarations of the capabilities and requirements of those partners.

An ESB link has two endpoints, one for attachment of BSRs and one for attachment of BSPs; both ends can be qualified by using interface declarations and policies just like interaction endpoint declarations. A link defines the “ideal counterpart” for service requestors and providers: the provider attachment part of a link can be designed to provide an exact match for a particular registered BSP; and the requestor attachment part of the same link can be designed to provide an exact match for some registered BSR. The two ends of the link do not have to match—ESB-managed mediations on the link can enact the transformation required to make the ends meet.

Thus, an ESB link represents and implements the contract between interaction endpoints. It can be tailor-made for a particular requestor/provider pair. An integrator given the task to resolve the requirements of a business process component (here in the role of a service requestor) for services that implement process activities might select a set of service providers to be linked to the process and, in cooperation with the solution deployer, create a BSR representing the process, a set of BSPs representing the services invoked, and a set of links between them, configured to meet the requirements declared by the process.

Link configuration does not necessarily reflect only the requirements and capabilities of the endpoints that it connects; it can just as well implement requirements defined for a set of interactions, for example, in enterprise policies (such as logging all high-value transactions). A solution administrator can preconfigure ESB-managed links to support a specific quality of (interaction) service to be used by a number of interested interaction endpoints.

A link might be configured with a varying number of interaction endpoints attached to it. A solution administrator might register a BSP and attach it to a link that might not have any BSRs attached to it. Alternatively, the administrator might register an event source as a requestor and attach it to a link that will propagate the events it produces—potentially, without anybody listening at the other end. A link can be configured such that it dynamically determines the endpoints that need to be attached to it, e.g., depending on the content of the requests it is processing.

In the most dynamic case, a link can be created dynamically to perform matchmaking between dynamically established requestor/provider pairs. The configuration of the link can be derived from the requirements and capabilities of the endpoints that it is meant to connect.

As illustrated in *Figure 5*, both endpoints of a link as well as the link itself can carry policy declarations.

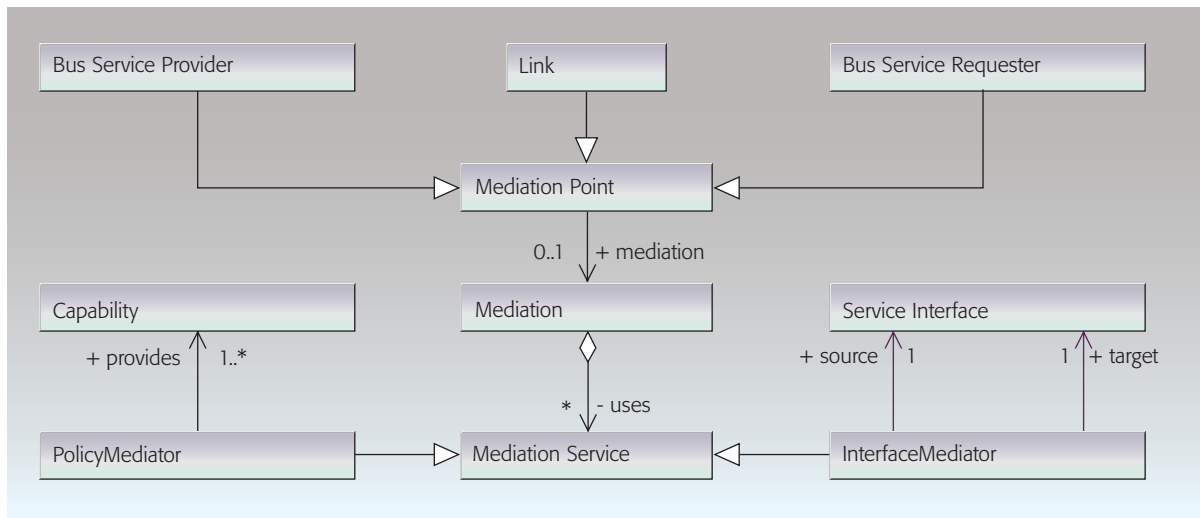


Figure 6
Mediation in the ESB integration model

Policies on the endpoints declare the constraints as well as the capabilities of the endpoints, whereas policies on the link itself can only represent capabilities implemented by the link for the benefit of attached endpoints that require the capability. Interfaces on the endpoints of a link are optional and often are omitted to indicate that the link can carry any type of message between associated endpoints. Note that service interfaces on both ends of a link can have different signatures. In this case, some mediation on the link needs to take care of the transformation.

SOA holds out the promise that services can be discovered from service directories and bound together to form new and exciting, or simply more efficient, applications. Unfortunately, existing applications were seldom designed to be linked together, and the integration specialist is faced with mismatches of protocol, format, and quality of service to reconcile, as well as requirements to make his new applications more flexible and resilient. The ESB addresses these challenges by interposing mediations between service requestors and providers which can reconcile their differences. In addition, mediations can reconfigure the links between requestors and providers, for example, to create an alternative routing or to create a reactive and autonomic system.

The ESB integration model supports the attachment of mediations at various points on a link between

interaction endpoints: a mediation can be associated with the registered service provider, registered with a service requestor, or attached to a link between them. This is formalized in the model through the concept of a *mediation point* (see **Figure 6**). A mediation point inserted at the requestor implies that the mediation will be performed no matter what provider the requestor interacts with; a mediation point activated at the provider end implies that the mediation will be performed whenever the provider receives a request, no matter which requestor it comes from; and a mediation associated with a link applies only to the specific interactions that occur through the link.

Mediations process messages as they flow through the ESB. *Interface mediations* operate on the message payload, which contains the information required by the service provider, and can change its content and its structure. In addition, the messages have contextual information associated with them, usually specified in message headers. This concept is familiar from SOAP headers, which contain, for example, the location of the service provider. Within an ESB, the message context includes additional quality-of-service and routing information about the link and the mediations required between the service requestor and provider. *Policy mediations* operate on the message context. In addition to the information in the messages, the ESB provides mediations with information about the ESB configuration by means of access to the ESB registry.

Mediations can change the content and format of the service requests or responses or modify their intended routing. Although these are useful and necessary functions of mediations, to fulfill the flexibility of an ESB, it is also important that a mediation can be restricted, for example, so that it cannot see sensitive payload information or change a particular routing. Mediations are characterized by the read or update access they require to the context and payload sections of the message, and this can be enforced by the ESB.

Mediation patterns

Mediations are not formally restricted in their capabilities, but their intended role is satisfying integration and operational requirements within the infrastructure, rather than implementing business-level processes. Within this role, there are several basic patterns, which are seen repeatedly, either as individual mediations or within more complex mediations.

The *monitor* pattern is used to observe messages as they pass through the ESB without updating them in any way. Mediations that conform to this pattern use the information in the context and payload in many different ways; for example, to monitor service levels, to assist in problem determination, to meter usage for subsequent billing to users, or to record business-level events, such as purchases above a certain dollar value. This pattern can also be used to log messages for audits or for subsequent data mining. The first two examples would require read access to only the context information in the message. The others would require read access to the payload too, though the messages could be logged as raw bytestreams, for later parsing.

The *transcoder* pattern changes the format of the message payload without changing its logical content. For example, it may convert a SOAP message into a JMS/Text message with an XML payload that matches the body of the SOAP message (possibly by mapping SOAP header fields into JMSProperties). Mediations which conform to this pattern can often be created automatically when there is a clear definition of the two formats and of the relationship between them. This pattern requires update access to the payload.

The *modifier* pattern updates the payload of the message without any change to the context in-

formation. It requires update access to the payload. There are two common subpatterns: transformation and enrichment. In the former, the message payload is transformed from one format (schema) to another, to match the definition of a message of the requestor to that of the provider. This includes “enveloping and de-enveloping” (the process of putting a message in one network format inside the format envelope needed for transmission over another network, or the corresponding removal of an envelope) and encryption. In the latter, the payload of the message is updated by adding information from external data sources, such as customization parameters of the mediation, or from database queries.

The *validator* pattern determines whether a message should be delivered to its intended destination. If not, it may silently ignore the message or may return a rejection response to the requestor. The check can be against the meta-data of the message, such as the schema, or permitted values for specific fields. Alternatively, the check can be against side information associated with the mediation, which may relate to one or more fields in the payload of the message or to information held in the message context, such as the origin of the message. This variant of the pattern includes authentication and authorization checks. Depending on the checks involved, this pattern requires read access to the message context or the payload, or both.

The *cache* pattern returns a valid response to the requestor without necessarily passing the request to a service provider. It maintains a cache of requests and their associated responses, and if it recognizes the request, it returns the response directly to the requestor. If the response is not available in the cache, the message is sent to the provider, and the cache is updated with the new response on its return. This pattern requires read access to the payload of the request and response messages and is unusual in that it only applies to request/response interactions.

The *router* pattern changes the intended route of a message, selecting between the service providers associated with the mediation. Simple selection would include routing between two versions of a service, with the percentage routed to the new version being increased by the system administrator as confidence in its capabilities increases. Another

example is routing to a local version of a service until it becomes overloaded, then routing to a more expensive remote service. This latter case could also take the importance of the message into consideration, as indicated by a “gold” user status or the size of a purchase, for example. This pattern requires update access to the context of the message and read access to the payload if it is needed for the routing-selection criteria.

The *discovery* pattern queries the ESB registry to discover the set of service providers that match the requirements of the requestor, selects one of them, and routes the message to it. This is an enhancement of the routing pattern; in this case, the set of possible service providers are not preconfigured at the mediation. Suitable providers match the requestor’s message format, the quality of service required, or the protocol supported from the mediation to the possible providers. This pattern allows for more flexible routing, for example in the failover situation mentioned previously, where a new remote data center can be brought online and its services registered, without having to update the configuration of every routing mediation. This pattern requires update access to the context of the message and read access to the payload if it is needed for the routing-selection criteria.

The *clone* pattern makes a copy of a message and modifies its route. The new message will then have a separate existence within the ESB. This pattern is useful in association with the monitor pattern, where the monitoring logic must not be allowed to delay the delivery of the message to its intended destination; for example, when the message is logged to a database. This mediation requires read access for the payload and update access for the context.

The *aggregator* pattern monitors messages from one or more sources over a time period and generates a new message or event, based on the input it considers. It defines a set of event types in which it is interested and uses aggregation rules to derive a new event. It may simply aggregate a specific set of events, or it may look for patterns in event streams and generate a complex event when a pattern is detected. This pattern is useful in complex event-processing scenarios.

Mediations can be explicitly configured by the integration specialist or the solution administrator.

The former might apply in the case of a modifier mediation, and it would transform from the format of the requestor to that of the provider and vice versa. The latter might apply if the solution administrator wanted to add a monitor mediation to a particular link in the ESB to measure performance.

The ESB can also configure required mediations dynamically to match the policy requirements and capabilities of the requestor and provider. If a service provider requires encrypted messages, the ESB can configure an encryption mediation for the requestor. If the provider changes its algorithm, the next service request will fail; the ESB will query the provider’s meta-data, reconfigure the encryption mediation, and reissue the request.

ESB USAGE PATTERNS

The mediation patterns described provide some basic building blocks for the ESB. Higher-level patterns (which might helpfully be described as “usage patterns”) provide a means for describing and defining interactions and component topologies at the system or solution level and help us to see how and where the abstract concepts that we have been describing can be applied to specific implementation scenarios. Patterns enable and facilitate the implementation of successful solutions through the reuse of components and solution elements from proven successful experiences. IBM’s patterns for e-business provide one such example and, with specific relevance to the ESB, introduce a set of collaboration patterns that design or describe broad organizational relationships among applications and a set of interaction patterns that describe required behavior in greater detail (see Reference 10 for more information).

The fundamental concept in this case is that of the *broker application pattern*, in which distribution rules are separated from applications, enabling great flexibility in the distribution of requests and events and reducing the proliferation of point-to-point connections, thereby simplifying the management of the network and system. This basic pattern appears in several variations, and we will briefly consider each of these variations in this section.

Service and event-routing pattern: A request or event is distributed to at most one of multiple target providers (see *Figure 7*). Examples may include simple service selection based on context or the

content of the request or on more complex models, in which service requests can be routed to particular systems based on availability, workload, or detection of error situations. Service selection may involve the lookup of appropriate service providers in a service registry.

Protocol switch pattern: A routing pattern in which requestors and providers use differing network protocols (see **Figure 8**). Examples may include simple mapping of SOAP/HTTP requests onto a more reliable SOAP/JMS infrastructure or mapping between JMS and non-JMS applications.

Proxy or gateway pattern: A variant of the routing pattern (or protocol switch) which maps service interfaces or endpoints, possibly providing security functions (authorization and access control) and logging or auditing capabilities (see **Figure 9**). The proxy may also support disaggregation (and subsequent reaggregation) of a single request into multiple component subrequests. Examples of this pattern include service portals, in which a single point of contact is provided for multiple services and the details of “internal” services may be hidden from the service requestors.

Event distribution pattern: Events may be distributed to more than one target provider, based on a list of interested parties that is managed by the ESB (see **Figure 10**). Services that wish to be notified of such events may be able to add themselves to the interested-parties list. An example of this pattern would be the distribution of business events based on CBE through the common event infrastructure.

Service transformation pattern: Requestors and providers use different service interfaces, and the ESB provides the necessary translation (see **Figure 11**). This pattern exposes new service interfaces without requiring change or modification to an existing application or service. It may also be used when multiple providers support the same business function but provide different interfaces, allowing this difference to be hidden from the service requestor.

Matchmaking pattern: Another variant of the service routing pattern in which suitable target services are discovered dynamically based on a set of policy definitions (see **Figure 12**). This pattern is used in very dynamic environments where there are many

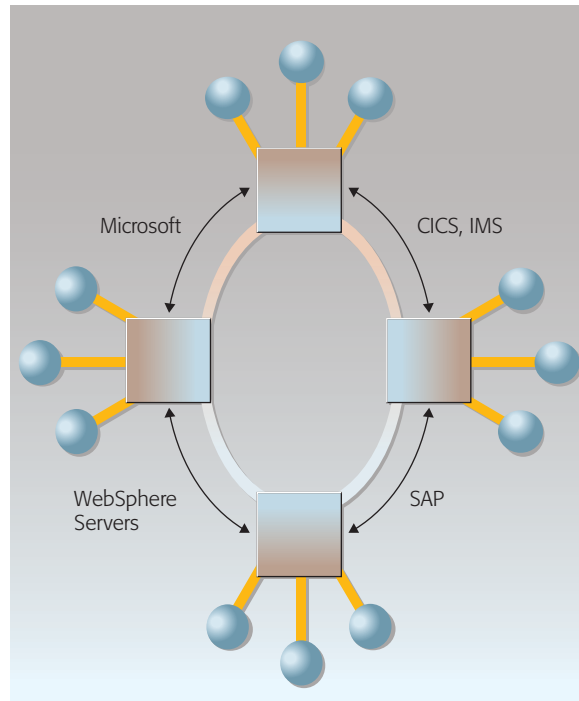


Figure 7
Service- and event-routing pattern

hundreds or even thousands of services attached to the ESB, and service implementations may or may not be available when any given request is issued.

These basic interaction patterns may also be used in conjunction with process-oriented interaction patterns. A process or workflow definition (defined by using BPEL or some equivalent language) extends the broker interaction pattern by orchestrating the execution sequence for a number of service inter-

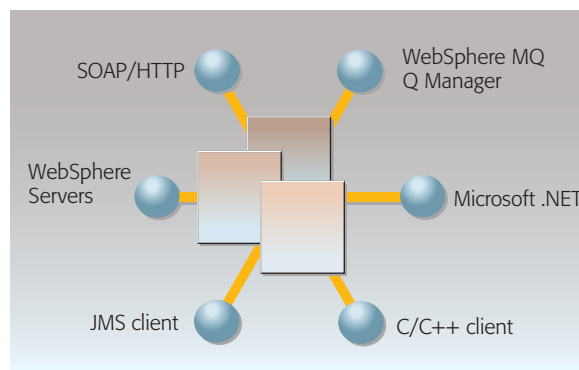


Figure 8
Protocol switch pattern

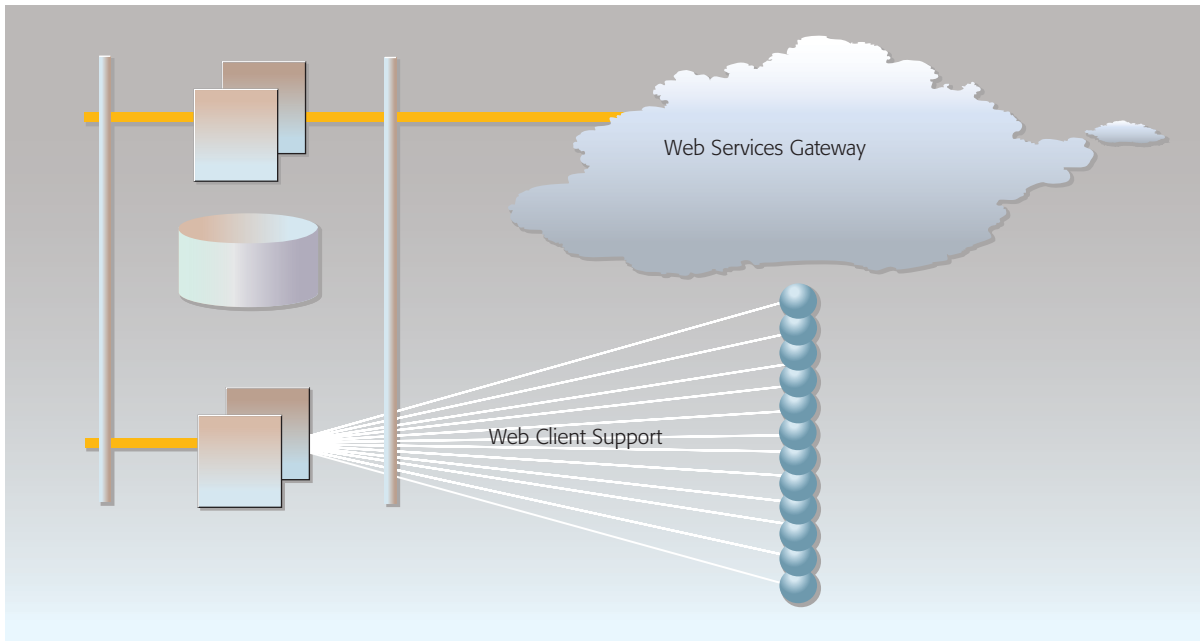


Figure 9
Proxy or gateway pattern

actions. By using these two patterns together, the service that orchestrates the interaction pattern can focus exclusively on business-process requirements, delegating issues of matchmaking, routing, and service selection to the ESB infrastructure.

CONCLUSION

We have presented the ESB and its role as the infrastructure underpinning an integrated and flexible SOA. Our presentation identified service meta-

data managed through a service registry as a key component of the ESB, allowing integration specialists and administrators to create and manage service-oriented solutions. Clear definition of the interfaces, and of the capabilities and requirements of the services, enables mediations to reconcile differences between service requestors and providers. We discussed a range of mediation patterns.

We described ESB usage patterns in which these abstract concepts are applied to enterprise scenarios. These concepts are realized through a variety of

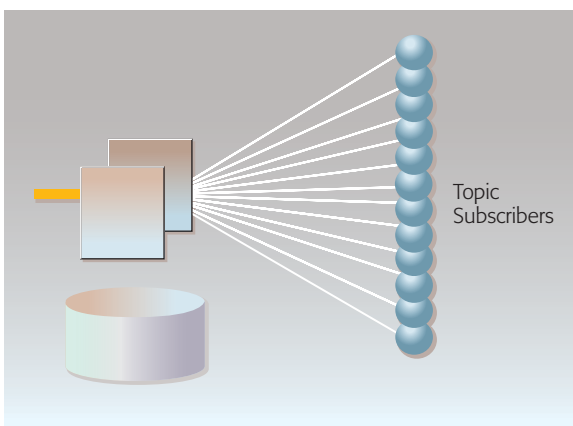


Figure 10
Event distribution pattern

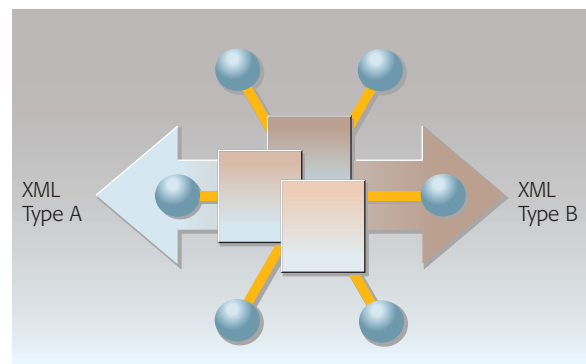


Figure 11
Service transformation pattern

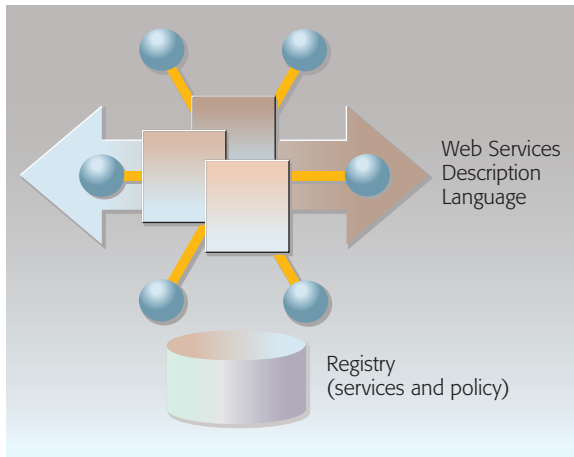


Figure 12
Matchmaking pattern

technologies and products in a large and growing number of customer solutions, including large-scale retail and brokerage applications. ESB adoption and use is expected to continue at full strength for the foreseeable future, and the ESB plays a central role in the implementation of the architecture for the IBM On Demand Operating Environment.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Sun Microsystems, Inc. Microsoft Corporation, Object Management Group, Inc., or Massachusetts Institute of Technology.

CITED REFERENCES

1. *Web Services Architecture Overview*, IBM developerWorks (Sept 2000), <http://www-106.ibm.com/developerworks/web/library/w-ovr>.
2. D. Cox and H. Kreger, "Management of the Service-Oriented-Architecture Life Cycle," *IBM Systems Journal* **44**, No. 4, 709–726 (2005, this issue).
3. *Web Services Policy Framework (WS-Policy)*, IBM, BEA, Microsoft, SAP, Sonic Software, VeriSign (September 2004), <http://www-128.ibm.com/developerworks/library/specification/ws-polfram/>.
4. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Golland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, *OASIS Web Service Business Process Execution Language V2.0 Working Draft* (May 2005), <http://www.oasis-open.org/committees/download.php/12791/wsbpel-specification-draft-May-20-2005.html>.
5. J. Farrell, R. Akkiraju, and M.-T. Schmidt, *Web Services Semantic Annotations, Technical Note, Version 1.0* (December 2004), <http://awwebx04.alphaworks.ibm.com/ettk/demos/wstkd/doc/services/demos/psme/webapp/WebServicesSemanticAnnotations.htm>.
6. OWL-S: Semantic Markup for Web Services, W3C Member Submission 22 November 2004 <http://www.w3.org/Submission/OWL-S/>.
7. D. Ogle, H. Kreger, A. Salahshour, J. Cornpropst, E. Labadie, M. Chessell, B. Horn, J. Gerken, J. Schoech, and M. Wamboldt, *Canonical Situation Data Format: The Common Base Event V1.0.1* (2004), http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/hyades-home/docs/components/common_base_event/cbe101spec/CommonBaseEvent_SituationData_V1.0.1.pdf.
8. *OWL Web Ontology Language Overview*, W3C Recommendation (February 10, 2004), <http://www.w3.org/TR/owl-features/>.
9. B. Portier and F. Budinsky, *Introduction to Service Data Objects*, (September 28, 2004), <http://www-106.ibm.com/developerworks/java/library/j-sdo/>.
10. C. Sadtler, D. Cotignola, B. Crabtree, and P. Michel, *Patterns: Broker Interactions for Intra- and Inter-Enterprise*, IBM Redbooks, SG24-6075 (2004), <http://www.redbooks.ibm.com/redbooks.nsf/0/532fca172da15c6c85256d6d0046192e?OpenDocument>.

Accepted for publication June 3, 2005.

Published online October 24, 2005.

Marc-Thomas Schmidt

IBM Software Group, Hursley Park, Hursley SO212JN (mitschmidt@uk.ibm.com). Mr. Schmidt is an IBM Distinguished Engineer and has been working on IBM Business Integration technologies for more than a decade, from workflow management systems to advanced message-oriented middleware and business-process-management technology. In his current role as ESB chief architect, he leads the work on technical architecture for IBM's ESB technologies.

Beth Hutchison

IBM Software Group, A2123, MP 189, Hursley Park, Winchester SO21 2JN (beth_hutchison@uk.ibm.com). Ms. Hutchison is a Senior Technical Staff member and a Web Services architect working on IBM ESB technologies. Since joining IBM, she has consistently worked on leading-edge technologies, initially as the lead developer for the first release of WebSphere MQ on distributed platforms. Subsequently, she became a performance architect for IBM Java virtual machines. She has now rejoined the MQ family and is working on systems management throughout the ESB.

Peter Lambros

IBM Software Group, Hursley Park, Hursley SO212JN (lambros@uk.ibm.com). Mr. Lambros is a Senior Technical Staff Member in the IBM ESB and MQ Development team, working on the strategy and architecture of IBM's integration technologies. He has been the chief architect of WebSphere Business Integration Message Broker since its inception as MQSeries Integrator and is also currently working on the architecture of mediation technologies for IBM's ESB products.

Rob Phippen

IBM Software Group, Hursley Park, Hursley SO212JN (phippen@uk.ibm.com). Mr. Phippen is a senior architect in the IBM ESB and MQ Development team, responsible for the specification and development of mediation components for WebSphere products. He has experience working on IBM's messaging technologies for more than five years, including time in the IBM MQ Strategy and Planning Team. ■