

# SONIC ESB: AN ARCHITECTURE AND LIFECYCLE DEFINITION

*Copyright ©2005. Sonic Software Corporation.  
All rights reserved.*



---

## TABLE OF CONTENTS

|  |    |
|--|----|
| > 1.0 Introduction   | 2  |
| > 2.0 Overview of Sonic ESB Architecture and Lifecycle     | 3  |
| > 3.0 Services, Messages, Mediation, and Connectivity      | 4  |
| 3.1 ESB Services and Messages                              | 5  |
| 3.2 ESB Processes  | 11 |
| 3.3 ESB Services   | 14 |
| 3.4 Connected Business Services                            | 16 |
| 3.5 Connections  | 17 |
| 3.6 The SonicMQ Multi-protocol Communication Broker        | 19 |
| 3.7 Web Services   | 22 |
| 3.8 Security   | 25 |
| > 4.0 ESB Life Cycle and Distributed Services Architecture | 26 |
| 4.1 Development and Configuration                          | 27 |
| 4.2 Deployment Tools and Migration of Configurations       | 33 |
| 4.3 Distributed Services Architecture                      | 34 |
| > 5.0 Conclusion   | 37 |
| > 6.0 Glossary   | 41 |
| > 7.0 Appendix – Complete UML Class Diagram                | 52 |

---

## > 1.0 INTRODUCTION

Service-Oriented Architecture (SOA) is fast becoming the systems and software approach of choice for the majority of medium and large enterprises.<sup>1</sup> According to the principles of SOA, IT professionals design, implement, and deploy information systems from components that implement discrete business functions. These components, called “services”, can be distributed across geographic and organizational boundaries, can be independently scaled and can be reconfigured into new business processes as needed. This flexibility provides a range of benefits for both the IT and business organizations.

Core to the IT infrastructure supporting SOA is the Enterprise Service Bus (ESB), which connects, mediates and controls all communications and interactions between services and provides error and exception processing. ESBs have been chosen as the best method for development and deployment of processes, ensuring reliability and ultimately success with the introduction and operation of SOA-based systems. Its design permits rapid change in services and in the connections among services, and provides the management visibility into services and processes across a distributed environment.

When looking at the cost of infrastructure and change in development and deployment processes, as well as ultimate performance and reliability of SOA projects, key questions arise with regards to technology, management, and skills transfer. These questions include:

- > How do I build processes which span new service-enabled applications as well as existing legacy systems?
- > How do I provide the performance expected of enterprise systems while easily accommodating changes in demand?
- > How do I isolate applications from faults resulting from server and communication failures?
- > How do I manage processes that will interact with services spread across an organization, and between organizations?
- > How do I manage and monitor the infrastructure, as well as the processes and services deployed within it?
- > How do I allow the organization to change processes, rules, data mapping and relationships between applications with minimal effort and disruption?

Sonic ESB<sup>®</sup> (hereafter referred to simply as “ESB”) sets a reference mark for the SOA marketplace with which to examine these questions. The ESB is complemented by additional products in the Sonic SOA infrastructure family including Sonic Workbench, Orchestration Server, XML Server, Collaboration Server, and Database Server.

Given the power of an ESB to support SOA implementations it is imperative to understand in detail the functions and structure of the ESB. This paper will describe each of the key ESB functional areas through exposition of its function and benefits accompanied by conceptual structure diagrams based on an industry-standard approach, UML (Unified Modeling Language)<sup>2</sup>. The use of UML diagrams provides a conceptual view of the ESB components – and is useful for learning and discussion of the ESB mechanism. Other diagrams provided in this document employ ESB notation from David Chappell’s book [Enterprise Service Bus](#) and are used to depict examples of more expansive ESB use cases.

---

<sup>1</sup> Heffner, Randy, “Delivering Real World SOA”, Forrester Research, 2005.

<sup>2</sup> [www.uml.org](http://www.uml.org)

<sup>3</sup> Chappell, David A., [Enterprise Service Bus](#), O’Reilly Associates 2004

## > 2.0 OVERVIEW OF SONIC ESB ARCHITECTURE AND LIFECYCLE

Core to a service-oriented architecture is the support of services, processes, and their interactions. Services can be either ESB services which are natively hosted and therefore execute entirely within the ESB environment, or connected business services which are connected to and therefore integrated by the ESB. The connected services can be of many different types, such as web services hosted by J2EE or .NET application servers, or legacy systems with web services “wrappers” or JMS (Java Message System) adapters.

As an example, Figure 1 shows an SOA that integrates and mediates a multi-channel financial trade process, where an ESB connects an order management system that is composed through a “wrapped” connected business service to a compliance system which is an ESB service. The ESB provides the connectivity between services, the XML transformation service to allow services with varying interfaces to communicate, and the business process of the service invocations which invokes services’ actions in sequence.

This example illustrates how an equity trade order message, originating as a message from an order management system on the left, would undergo a set of processing steps. First would be a transformation of data elements to support the requirements of the compliance and trading services. Second would be the actual trading services step. Third and fourth would be the triggering of the funds transfer process and ultimately a logging service would record the trade as complete. Again, rather than an integrated application, in this case the trade process is executed as a sequence of heterogeneous process steps, coordinated by the SOA infrastructure.

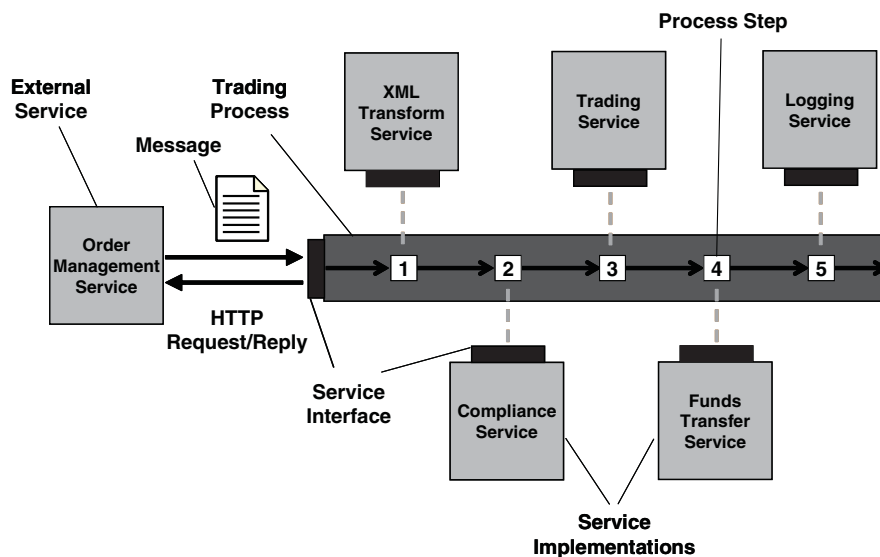


Figure 1. Example of SOA Application

The ESB provides a single service abstraction<sup>4</sup> across all operating environments and all network topologies. The advantage to this approach is the ability to simplify access to all systems through a common services paradigm including common development, deployment, and management tools.

<sup>4</sup> This service abstraction can be described in industry-standard WSDL, effectively hiding the details of a services’ implementation from other services.

---

A principal benefit of SOA, this helps to reduce cross-service dependencies that make distributed systems hard to change.

To reconcile otherwise incompatible operating environments or network topologies, the ESB provides special mediation services which transform and route information between the services that provide the core business functions of a SOA application. Furthermore, these mediation services allow the flexibility to integrate new components as services into the SOA application without requiring changes to existing components.

SOA applications are often termed “composite applications” because services are integrated through mediation services and business processes to create broad-reaching applications. The breadth of these applications can be understood in the perspectives of both business function and geographical distribution.

The practical concerns of SOA applications include how to manage the development and distributed deployment of the applications as well as how to monitor and manage the production systems. The Sonic ESB distributed services architecture provides the ability to configure, deploy, monitor, and manage world-wide ESB-based SOA applications.

In the next section, this paper will present the ESB architecture by looking first at the core ESB capabilities for sending messages between services, the types of services supported by the ESB, ESB mediation, and connectivity between services. Then, in section 4, there will be a review of the ESB lifecycle from development to production, followed by an overview of the mechanisms which provide reliability and scalability to the distributed services architecture<sup>5</sup>.

### > 3.0 SERVICES, MESSAGES, MEDIATION, AND CONNECTIVITY

This section defines the key capabilities of the ESB with regards to the integration of services through messages and mediation, the kinds of available ESB services and connected business services, and the core infrastructure to support connectivity between services.

Communications between services are message-based. These messages carry the requests from one service to another as well as the reply. The contents of the message could be in XML, or a proprietary binary data format. The transformation service is a mediation service which converts the format of data carried in messages to and from services with otherwise incompatible data format specifications.

Data transformation is one kind of mediation. Others include the content-based routing of messages from one service to another, the interconnection of synchronous and asynchronous services, and the creation of business processes as the composition of services invocations.

---

<sup>5</sup> For additional detail, please refer to the Sonic ESB documentation set, “Sonic ESB V6.1 Documentation”, [http://www.sonicsoftware.com/products/sonic\\_esb/documentation/](http://www.sonicsoftware.com/products/sonic_esb/documentation/).

The benefit of performing transformation and routing logic within an ESB as opposed to within applications is derived from configuration-driven, late-bound behavior of the ESB. As business requirements begin to drive demand for more complex transformation and routing logic, the ESB serves as the point where such changes can be made rapidly from a single console. By contrast, executing a set of related changes in multiple applications typically involves complex dependency analysis and coordinated development and deployment. In an ESB, applications can become loosely coupled, avoiding the traditional inter-lock between application deployment cycles of multiple application components, often owned by different divisions or businesses.

Figure 3 shows a UML class diagram<sup>6</sup> with the basic ESB structures supporting services and communications between services: ESB service, ESB message, ESB address, ESB endpoint, and connection<sup>7</sup>. The union of all of the UML class diagrams in this paper is shown in a single diagram in the Appendix.

### > 3.1 ESB SERVICES AND MESSAGES

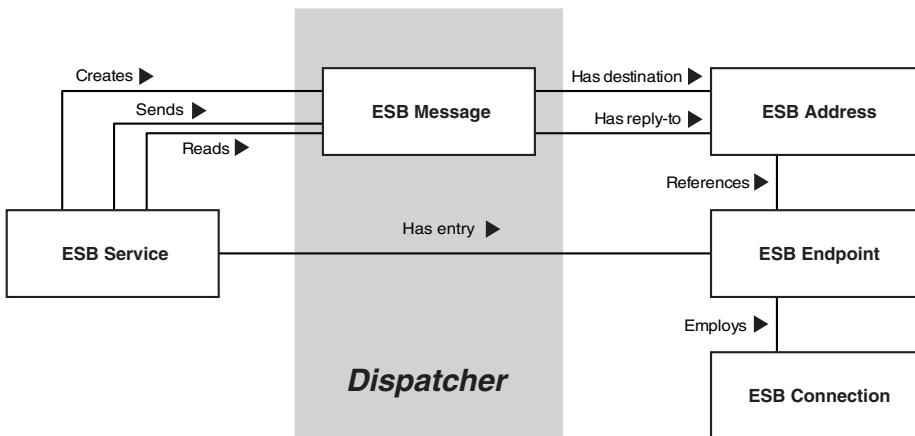


Figure 3. Class Diagram for ESB Messages

An ESB service creates an ESB message, with either XML or other content, and sends it to a designated destination ESB address. The ESB address provides a reference to an ESB endpoint which is a named destination for receipt of messages. The receiving ESB service listens on the ESB endpoint for receipt of a message. If the sending service requires a reply, then a reply-to address referring to the sending service is sent included in the message.

<sup>6</sup> In a UML Class Diagram, the classes, shown as boxes, represent a set of instances of the class. Associations between the classes are shown as named lines. The arrowhead, next to the name of the association is the 'reading direction' of the association, so in Figure 3, you can read "An ESB Service reads an ESB Message".

<sup>7</sup> The first mention of each class will be highlighted in bold type.

The fact that the ESB uses a logical endpoint and connection abstraction between the service and the transport layer provides several advantages:

- > ESB service logic can be far simpler; it need not accommodate specific details of the transport layer.
- > ESB services can be adapted to new transport and protocol requirements through configuration-driven mechanisms rather than re-coding and re-compiling “hard-wired” logic in the service. The net effect is a more agile environment, one which encourages greater service reuse.

The endpoint provides the capability for ESB services to receive messages with a specified quality-of-service (QoS): “exactly once” which uses persistent message stores to ensure that no messages are lost or delivered more than once; “at least once” to guarantee by using a persistent message store that each message is delivered, but may be delivered more than once; or “best effort” which provides no guarantee of delivery, but does not require the use of persistent message storage making it faster.

The messaging and its QoS are supported by an ESB connection which provides the communications channel and queuing mechanism. For Sonic ESB, while a variety of connection types can be used to interact with connected services, including HTTP(S) and JMS (Java Message Service), JMS provides the connections between ESB services and ESB processes. Connections are described in section 3.4.

For example, Figure 4 shows a UML object diagram<sup>8</sup> matching the portion of Figure 1 in which the compliance system sends a message containing a trade request to the trading system.

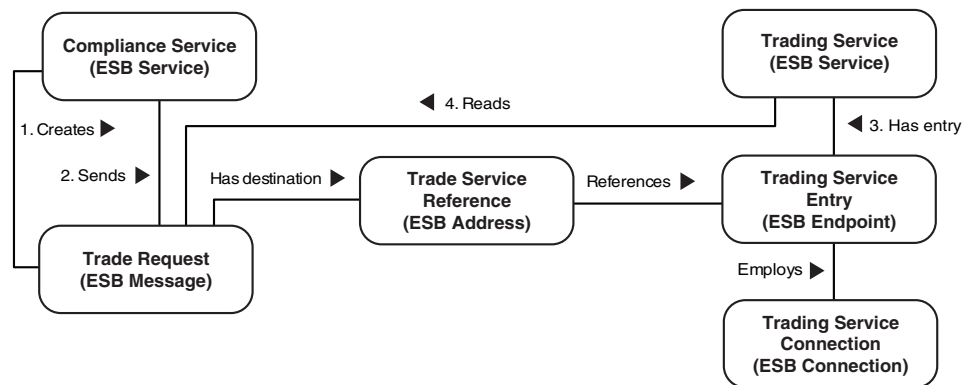


Figure 4. Object Diagram for Message sent between ESB Services

Figure 5 shows the example in Figure 4 extended to show a confirmation reply sent back to the compliance service. The reply-to address sent in the trade request message becomes the destination address for the confirmation response message.

<sup>8</sup> An object diagram is an example of instances of the classes. Shown with each object is a name of the instance as well as its class name.

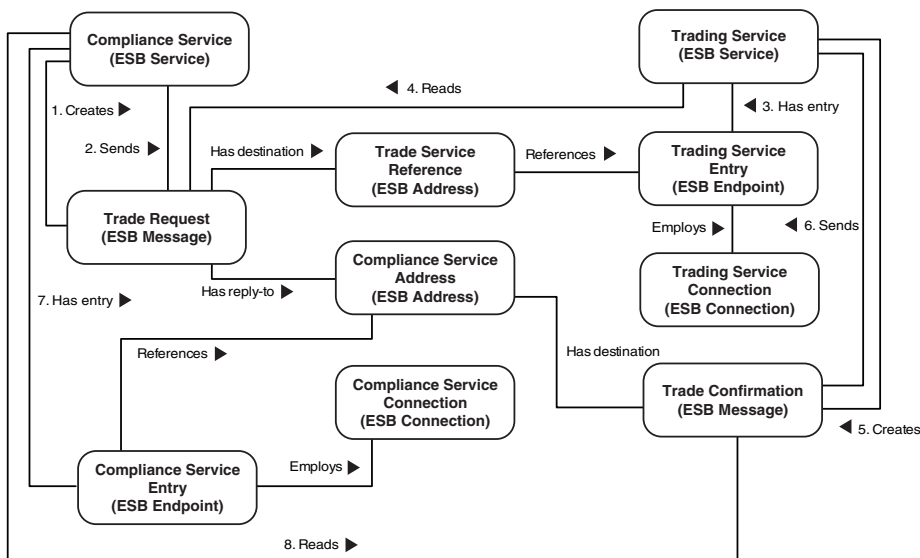


Figure 5. Object Diagram for request and reply between ESB Services

It is important to note the power of the “reply-to” address in the context of the ESB. The reply-to feature allows the service to be invoked from many different contexts. It provides the intelligence not only to respond to the specific requester, but also to respond using the correct protocol. For example, a single ESB service could be exposed as a web service, a native JMS service, another ESB service, or as an EDI protocol. This ESB service would have the capability to respond correctly in any context.

As shown in Figure 6, each ESB service or ESB process can be configured with a set of ESB addresses for the receipt of messages and for the sending of pre-defined types of responses. These addresses are:

- > **Entry:** The address of the endpoint from which the dispatcher accepts incoming messages for invocation of the service.
- > **Exit:** The address for the service’s response message. This is often set to ‘reply-to’ to send the message back to the requester. It could also be configured to send the response to the next service in a business process.
- > **Fault:** The address for sending a service error message indicating that the service was unable to process the request. Typically, this is configured to a fault destination address, but this could also be configured to the reply-to address of the requester.
- > **Reject:** If a message cannot be delivered because, for example, it is improperly formatted or has an invalid address, then it is sent to the configurable rejected message address.

Note that the use of these standard addresses allow the developer to remove a certain amount of application handling logic from the service and move it to the more configurable ESB. This can greatly simplify the overall effort to create services.

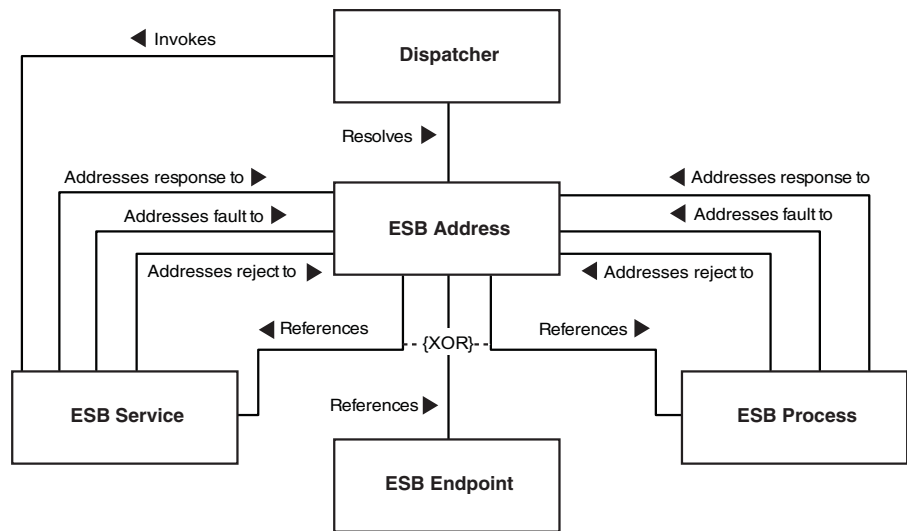


Figure 6. Class Diagram for Standard ESB Services and Process Configuration

When an ESB process has configured endpoints which differ from those of services invoked as a process step, the process configuration overrides that of the service. This allows the behavior of the service to be specialized for each use in a process. For example the configured fault and rejected message addresses for a service may be different depending on which process employs it.

Each ESB address can refer to one of:

- > ESB Endpoint: An abstraction of the underlying communication transports.
- > ESB Service: This refers directly to the service name, and has the same effect as referring to the configured entry endpoint of the service.
- > ESB Process: This is a sequence of services composed into a business process as explained in detail in the next section.

Each service instance has one or more dispatchers which accept incoming messages to a service, prepare the message contents for the service, invoke the service, accept its output, format an outgoing message and resolve the addresses on messages and send them to the proper endpoints, services and processes.

For example, an ESB service such as the compliance service in Figure 1 can configure its address set as shown in the object diagram in Figure 7. The dispatcher is configured to listen for service invocations on the “compliance service test” endpoint. The exit is configured to responses to the

“reply-to” endpoint, which will route the response to the requesting service. Faults and rejects are sent to the test error processing service. When this service is used in an ESB process, as described in the following section 3.2, these addresses can be overridden by the ESB process configuration.

Important to the development and distribution of ESB services is that the setting and changes to the standard service configurations are done through interactive tools without requiring programming. This will be explained in additional detail in section 4.

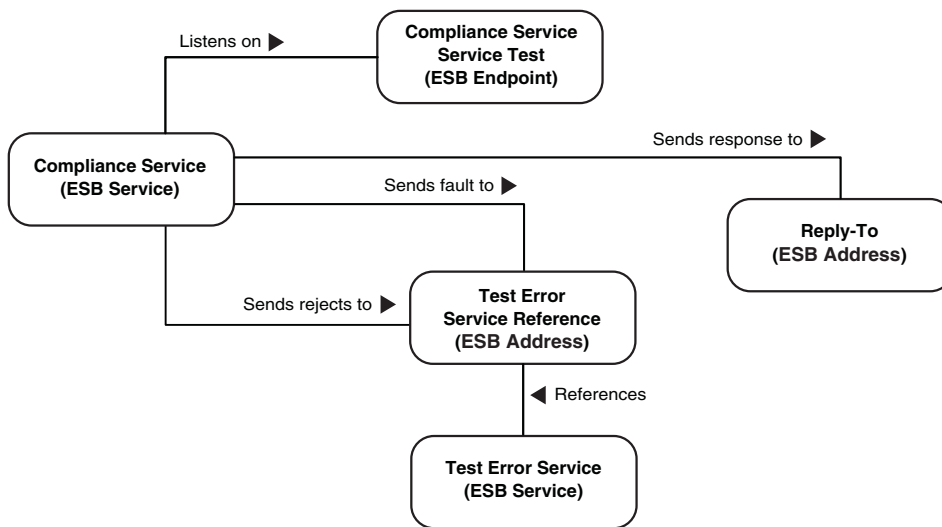


Figure 7. Object Diagram for Standard ESB Services Configuration

As illustrated in Figure 8, the dispatcher plays a central role in the execution of a service. The dispatcher manages the sending and receiving of ESB messages, invokes the service when a message arrives on its entry point, and, as will be explained in the next section, interprets the ESB itinerary for ESB processes.

The dispatcher logic is a common component in every ESB container, the deployed distributed services execution environment. This frees the service developer to focus on service business logic, rather than worrying about underlying dispatching requirements such as itinerary processing, message format transformations, monitoring and error recovery.

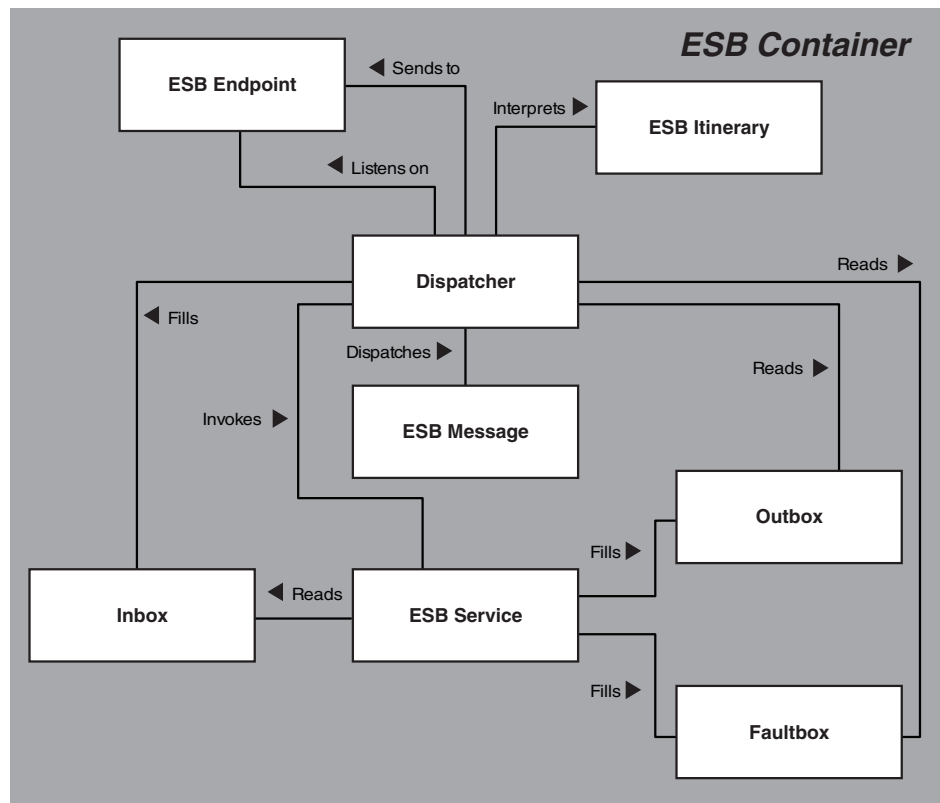


Figure 8. Dispatcher in ESB Container

A dispatcher invokes a service each time a message arrives at the service's entry endpoint. From a programming perspective, all services communicate through standard methods for reading messages and for sending messages. In particular the service obtains its input message from its inbox. Output messages are placed in either the outbox or faultbox for delivery by the dispatcher when the service's invocation is complete.

There can be multiple dispatcher instances to work with multiple instances of a service deployed in a given ESB container. In addition, dispatchers in multiple containers distributed across the enterprise can listen on ESB endpoints with the same name and hence be triggered by the same events. These two models provide great flexibility to scale services and provide service redundancy. Multiple services, configured with the same entry endpoint, naturally load balance as the service instances compete to consume messages from the shared endpoint. Additional instances of an ESB container can be, with its configured set of services, created dynamically in response to rapid increases in load.

The simplicity and symmetry of standard input and output methods with configurable service entry and exit endpoint addressing provides great flexibility in the arbitrary combination of services. This approach combined with the scalability model afforded by having multiple containers, each with multiple threads of a given service, contributes greatly to the ESB's support of service re-use and location transparency, principal benefits of SOA.

## > 3.2 ESB PROCESSES

The ESB provides for the management of business processes as a sequence of services, as shown by example in the five-step trading process in Figure 1. The ESB process is defined by an ESB itinerary which contains a sequence of process steps. Each process step can invoke one (or multiple) ESB services, other ESB processes, or a connected web service. The ESB itinerary travels with an associated ESB message providing a completely distributed execution environment for the ESB process. The distributed nature of an ESB process makes it highly scalable. Process state always travels with the message, so ESB process capacity scales with increased throughput of the underlying communications infrastructure. Quality of service guarantees of the underlying communications infrastructure makes this approach highly reliable as well.

The same ESB service can be used in many different ESB processes with the usage specialized to that process' needs. Optionally associated with each process step are runtime parameters which customize the invocation of a service for this particular process. For example, the XML transformation service can be specialized by associating a specific XSLT stylesheet with the use of the XML transformation service in a process step. In this example, the runtime parameter value might be the URL for the XSLT stylesheet defining the transformation logic.

Figure 9 shows the key classes for the configuration and execution of ESB processes including the ESB itinerary, process step, runtime parameter, WSDL definition, and ESB container classes.

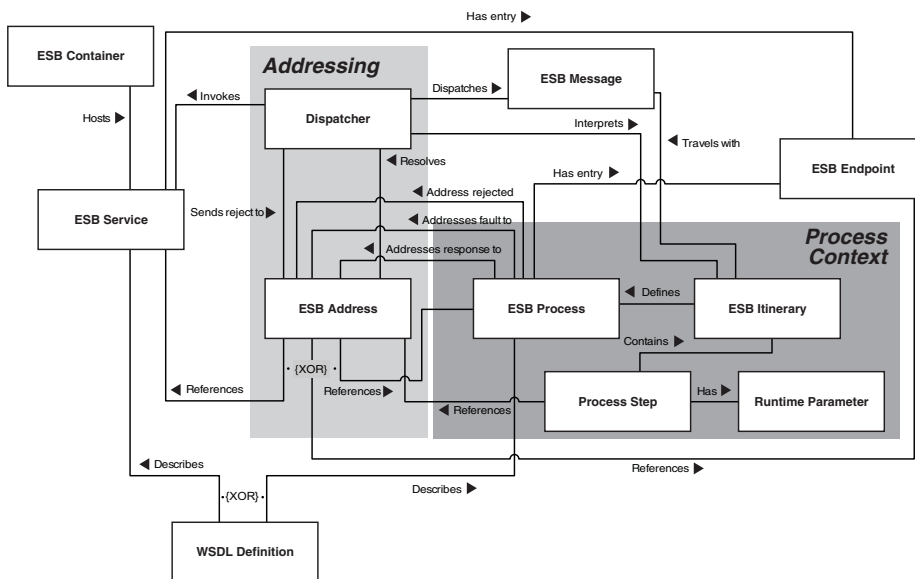


Figure 9. Class Diagram for ESB Process and Addressing

---

As will be explained in more detail in section 4.3, ESB services execute within the ESB container. The ESB container instantiates ESB services and provides: the systems management interface for its services, the management of communication protocols, the capability to have multiple threads executing services for scalability, and the standard capability to execute distributed processes (by the dispatcher). Using a lightweight ESB container to host services avoids the need to deploy monolithic application servers on every machine to host services.

There is no central processing engine for the ESB itinerary because each ESB container can process and route the next step in the ESB itinerary. The itinerary can be likened to a baggage tag on a piece of baggage which states the initial flight and connecting flights to get the bag from its source to its destination. Similarly, the ESB itinerary travels with the ESB message from service to service in a process until it reaches its last service, the final destination. In addition to the baggage metaphor, itineraries can call for message replication to multiple destinations as well as for content-based routing.

This distributed approach provides efficient execution without the need to communicate to a central site as processing can continue even with losses of connectivity. This provides for “stepwise” reliable processing in which processing can continue in an individual container independent of the status of other containers. Appropriate QoS messages from one container to another are held waiting for any container experiencing outages. Furthermore, transparent load balancing by the ESB allows for multiple containers to listen in parallel on a single ESB endpoint allowing for ongoing processing even if one container fails.

The WSDL (Web Services Definition Language) definition applicable to the ESB process and ESB service allows their interfaces to be described using the Web service standards as defined by the W3C<sup>9</sup>. These interfaces can be used within ESB processes or in order to make an ESB service externally callable as a Web service. This will be described in more detail in section 3.5.

Figure 10 shows an object diagram of the ESB process shown in Figure 1. In particular it includes the ESB itinerary with its sequence of references to the five services in the process, in which each service’s dispatcher will in turn send the service’s resulting message on to the next service in the process. It also shows an XSLT stylesheet as a runtime parameter for the XML transform service that provides customized behavior for that service as used in this process. Section 4.1 will show how in addition to steps as service invocations, the ESB itinerary can include other kinds of steps such as decision steps, fan-out steps, and the invocation of Web services and other processes.

---

<sup>9</sup> World Wide Web Consortium, “Web Services Activity”, <http://www.w3.org/2002/ws/>

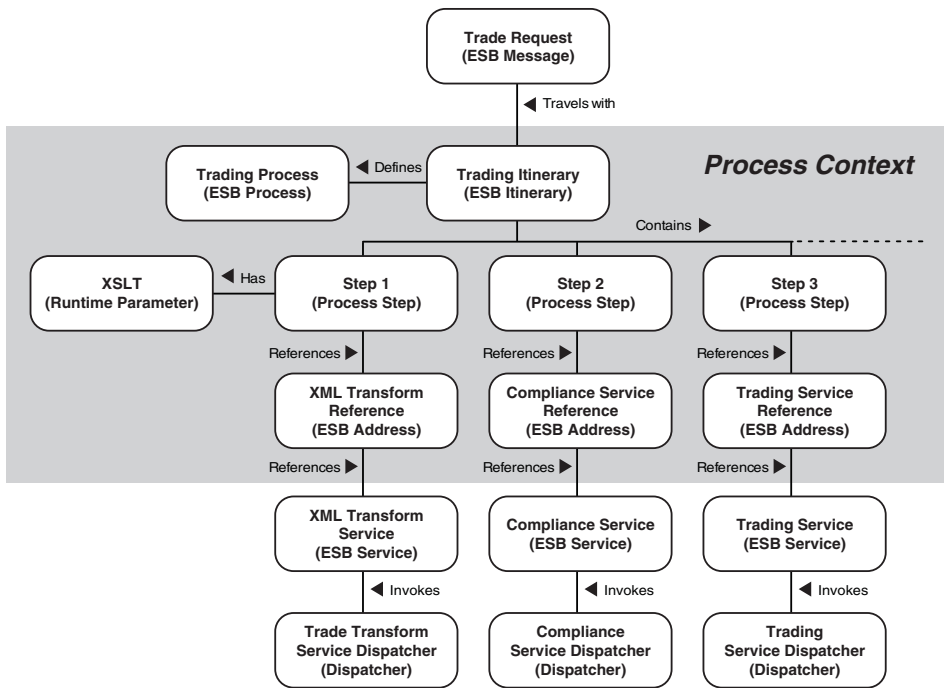


Figure 10. Object Diagram for an example ESB Process

> 3.3  
ESB SERVICES

ESB services can be of a number of types. These types are shown in Figure 11 as sub-classes of ESB service type<sup>10</sup>.

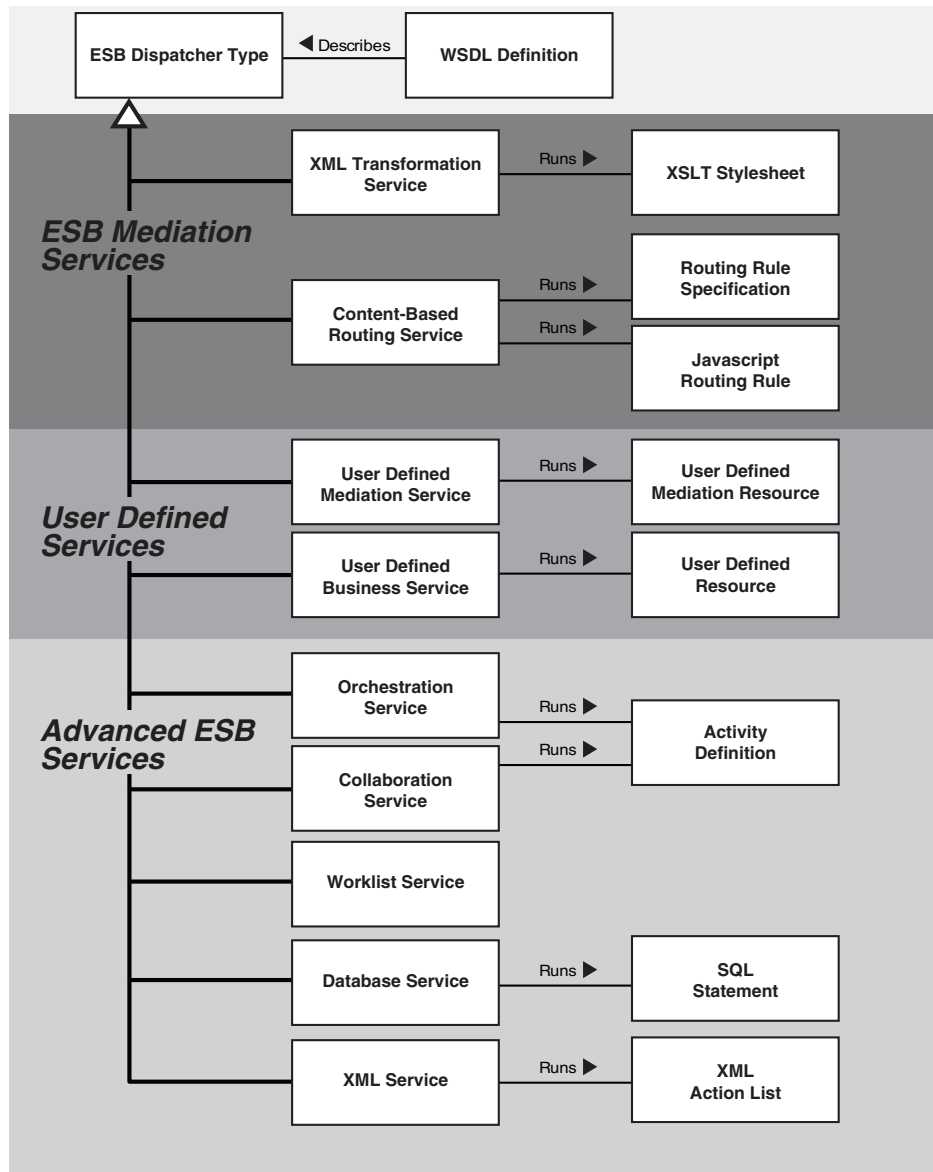


Figure 11. Class Diagram for ESB Services

The mediation services provide transformation and processing of messages in order to provide the right content in the right format to services which implement business functions.

The ESB mediation services include but are not limited to:

- > XML Transformation Service: Provides data conversion from one XML document to another according to either an XSLT specification or a transformation defined in JavaScript.

<sup>10</sup> In the UML diagram, the association with the hollow arrowhead is a “generalization” association showing sub-classes. This is sometimes read as “is a”, so for example from Figure 10 “the XML Transformation Service is a ESB Service Type”. The sub-class inherits the associations of the parent class.

- 
- > **Content-Based Routing Service:** Directs messages to various other service endpoints according to either configured routing rules or a JavaScript rule.

The ESB also provides the infrastructure for the development of and deployment of custom user-defined services for both mediation and business logic. This extensibility is a critical ESB differentiator which allows for the creation of ESB services which leverage capabilities available to all included services.

The ESB User-defined service types typically fall into two categories:

- > **User-defined Mediation Service.** Implements a mediation function such as data transformation between data formats other than XML. The resource files used to define this mediation are customized according to need. A common scenario creates a user-defined mediation service to move and transform files to and from a legacy application with proprietary data formats.
- > **User-defined Business Service:** Implements core business logic. This is a business service which executes entirely within the ESB environment. By contrast, connected business services (see below) execute in external environments such as application servers or legacy systems.

Another group of ESB services are the advanced ESB services which provide SOA related process control and storage services within the Sonic SOA Suite.

The ESB advanced services are:

- > **Orchestration Service:** Extends the capabilities provided by the ESB process and provides support for business process management (service orchestration) which require or benefit from centralized management of the process state. In particular, process joins require a centralized process state, and long-lived transactions are typically more easily managed in a centralized process model. The orchestration service business process is specified by a process activity definition which is created in Sonic Workbench in the form of a UML activity diagram. Additionally, the orchestration server supports process models specified in the BPEL4WS<sup>11</sup> workflow standard.
- > **Collaboration Service:** Provides for business processes extending across enterprises. It supports specific business-to-business (B2B) protocols such as RosettaNet and ebXML as well as the BPEL4WS workflow standard. Like the orchestration server, its processes are specified by process activity definitions.
- > **Worklist Service:** Provides management and presentation of end-user task lists generated by the execution of business processes by the orchestration and collaboration services.
- > **XML Service:** Stores, transforms and queries XML documents according to a sequence of actions specified in an XML action list.
- > **Database Service:** Provides full access to standard relational databases. The SQL statement resource includes SQL statements and instructions for passing parameters and results in ESB messages.

---

<sup>11</sup> Microsoft et. al., "Business Process Execution Language for Web Services Version 1.1", <http://www.oasis-open.org/committees/download.php/2046/BPEL%20V1-1%20May%205%202003%20Final.pdf>, May 2003.

### > 3.4 CONNECTED BUSINESS SERVICES

Connected business services are services implementing business logic in environments external to the ESB operating environment. These services are integrated through a variety of types of connections described in the next section. A composite SOA application typically comprises connected business services integrated by the ESB with ESB services into a business process. In Figure 1 the order management system is a connected business service which communicates via an HTTP connection. The compliance service, trading service and funds transfer service are business services that have been interfaced directly to the ESB as user-defined ESB services.

The classes of connected business services are shown in Figure 12.

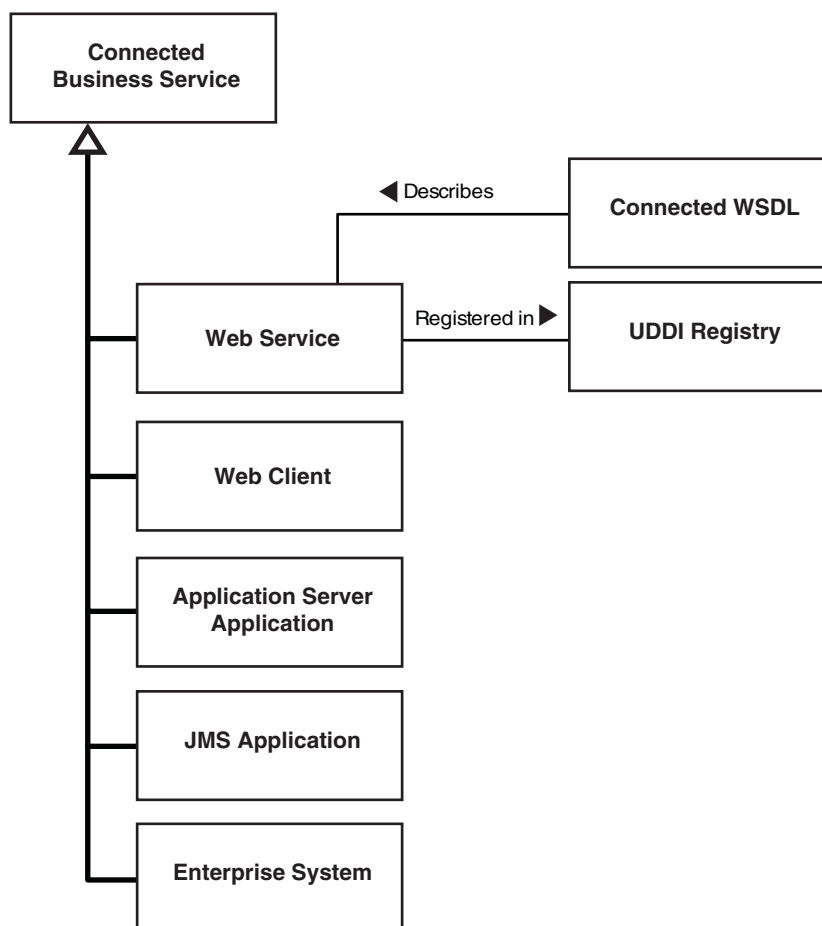


Figure 12. Class Diagram for Connected Business Services

The connected business services include:

- > Web services: Services with interfaces defined according to web standards. The interface to a Web service is defined by a WSDL (Web Services Definition Language) definition which can be

stored in a registry meeting the OASIS UDDI standard<sup>12</sup>. As explained in section 4.1, the ESB development tools can link to a UDDI registry to obtain the WSDL definition for a connected Web service.

- > Application server applications: Those applications which are implemented on J2EE or .NET application servers. These are for applications which have interfaces accessible by HTTP(S) communications, but not as standard Web services.
- > JMS (Java Message Service) applications: Any application with a native J2EE or JMS adapter can be a connected business service. Sonic and other third parties provide JMS adapters for a wide variety of enterprise applications.
- > Legacy systems: This is a “catch-all” for existing systems which have adapters or “wrappers” implemented for one of the ESB-supported connection types: HTTP(S), JMS or JCA (Java Connection Architecture), as described in the next section.

Each of these types of connected business services can communicate with each other through the ESB, and with ESB services providing mediation and business logic.

The ESB connection specifies which supporting communications channel is used by an ESB endpoint to connect with an ESB service or a business service. As shown in Figure 13, this ESB connection may be implemented as a web services connection, JMS connection or JCA (J2EE Connector Architecture) connection.

## > 3.5 CONNECTIONS

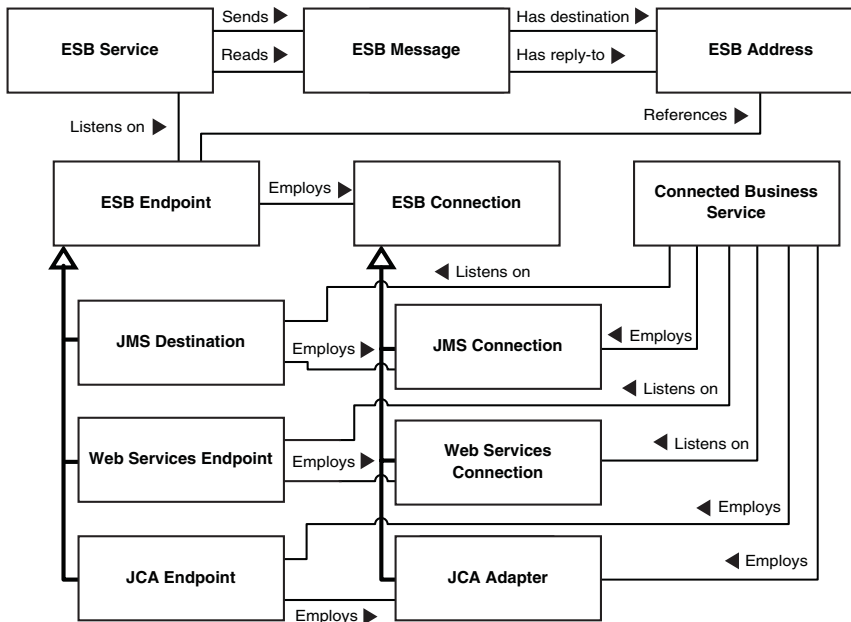


Figure 13. Class Diagram for Connections

<sup>12</sup> OASIS, [http://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=uddi-spec](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec)

The connection employs user configurable asynchronous and synchronous delivery semantics including “publish-and-subscribe”, “send-and-forget” and “request-reply”. In the example in Figure 1 the connection between the order management service and the trading process is HTTP request-reply. If a connected business service or ESB service needs to change the manner in which it connects, this can be affected without requiring changes to its implementation. When services require differing types of connections, the ESB provides the mediation between the differing protocols<sup>13</sup>.

Figure 14 shows examples of some of the key objects used in various connection scenarios.

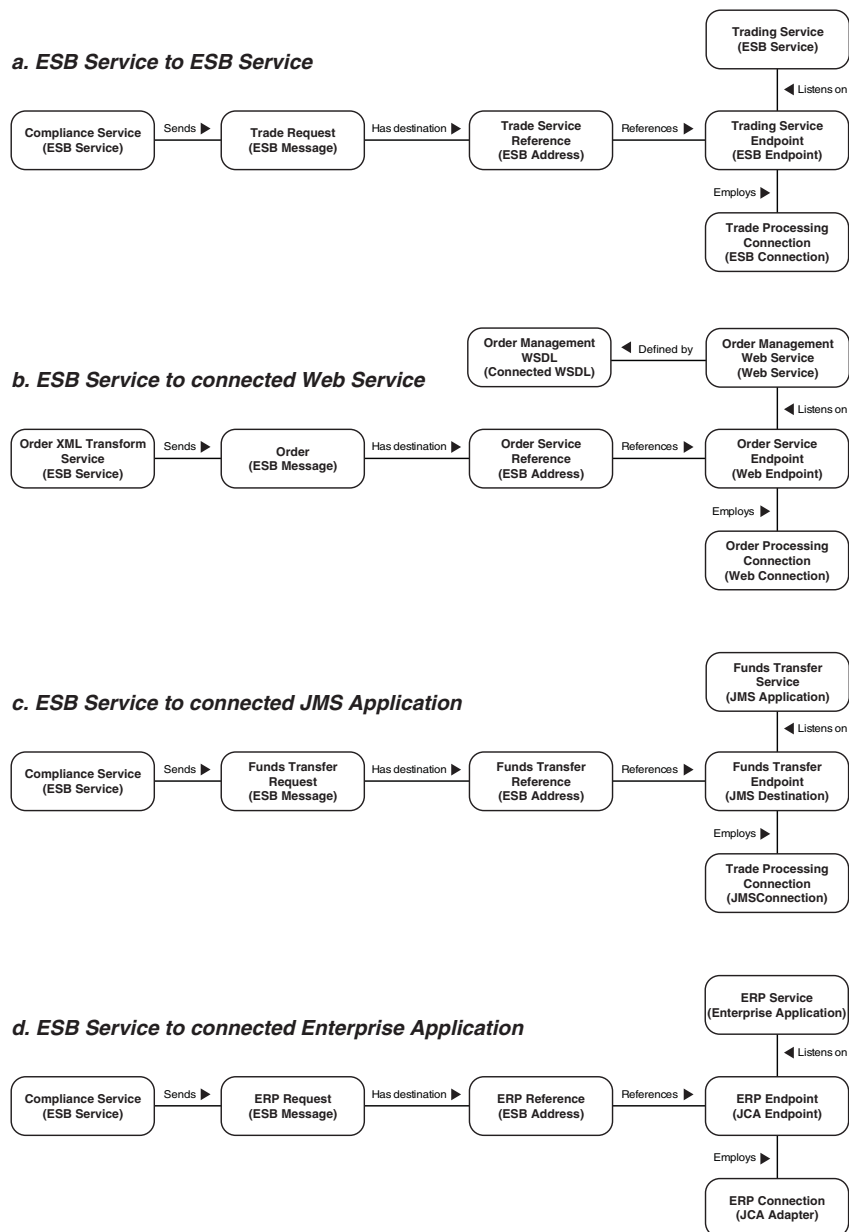


Figure 14. Example Object Diagram for Connections

<sup>13</sup> A special optimization capability for intra-container messaging (for in-memory connections) is also available for situations in which multiple services are hosted in the same ESB container. Refer to the Sonic ESB Developer’s Guide for more information.

---

The ESB requires a reliable, scalable multi-protocol broker to implement communications infrastructure and for connections with integrated business services, including Web services. SonicMQ provides a robust infrastructure for the ESB. Many of the attributes of the ESB with regards to reliability and scalability derive from the reliability and scalability of SonicMQ.

The broker is referred to as “multi-protocol” because of the need to support multiple connection types including JMS, HTTP(S), and the WS-\* web services protocols on top of SOAP and HTTP(S). Typically, JMS protocol is preferred for ESB service-to-ESB service communications for its high throughput and low latency, and WS\* protocols provide on-the-wire interoperability with connected services. In either application, the SonicMQ broker supplies the required protocols and also bridges to other widely-used messaging systems.

The ESB normally uses JMS to implement connections between ESB services as shown in Figure 15 in which ESB connections and ESB messages are implemented as JMS connections and JMS messages. The next section will show how HTTP(S) and web services are implemented by the multi-protocol broker.

## **> 3.6 THE SONICMQ MULTI-PROTOCOL COMMUNICATION BROKER**

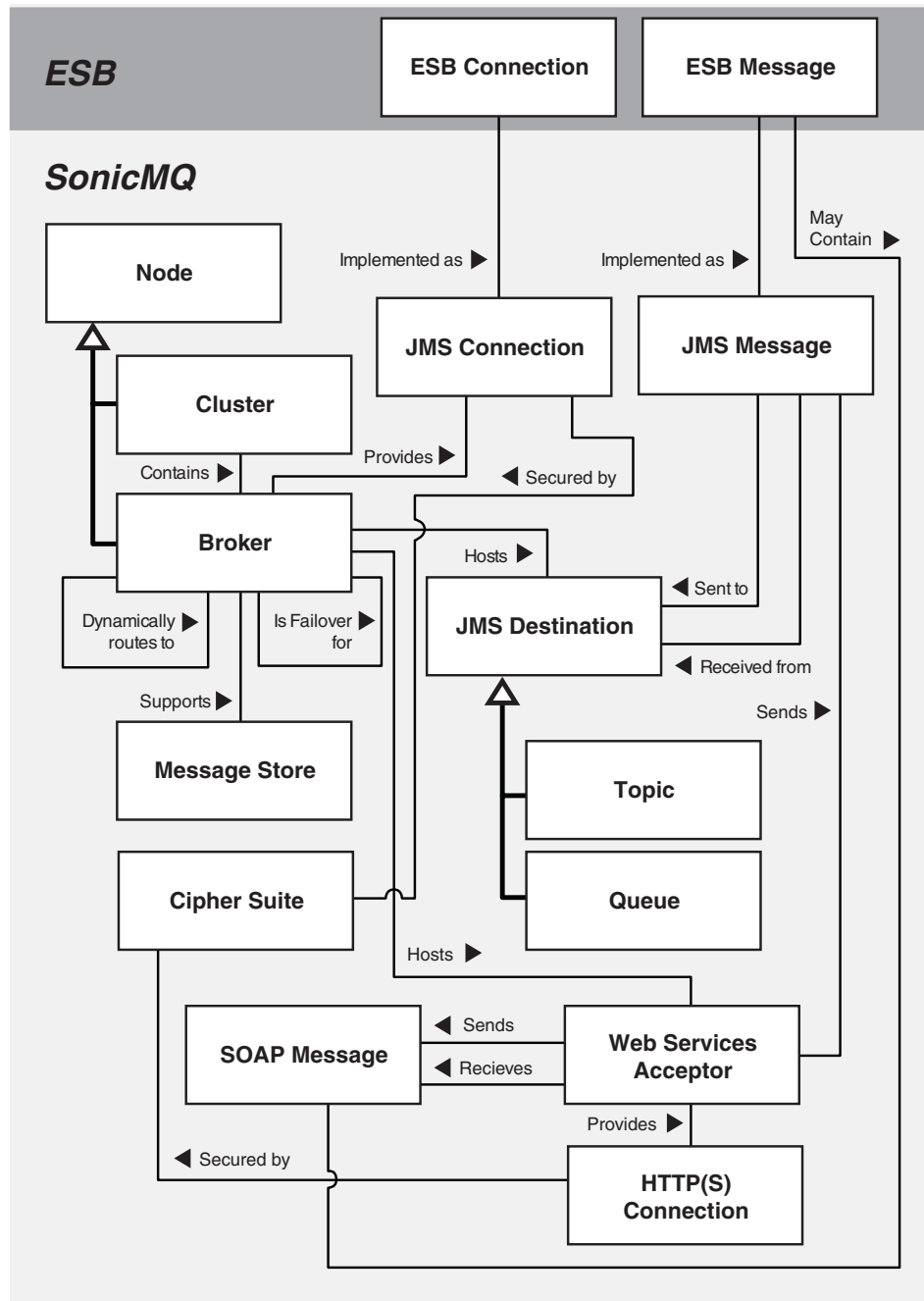


Figure 15. Class Diagram for SonicMQ support of ESB Connection

A node can be either a broker or a cluster. The broker provides the JMS connection as well as HTTP(S) and Web services. Brokers can be clustered for scalable throughput. A broker can also be associated with other brokers in a cluster to act as an active secondary server, providing stateful failover.

As an example, Figure 16 shows the objects involved the implementation of the trade processing ESB connection as would support the process in Figure 1.

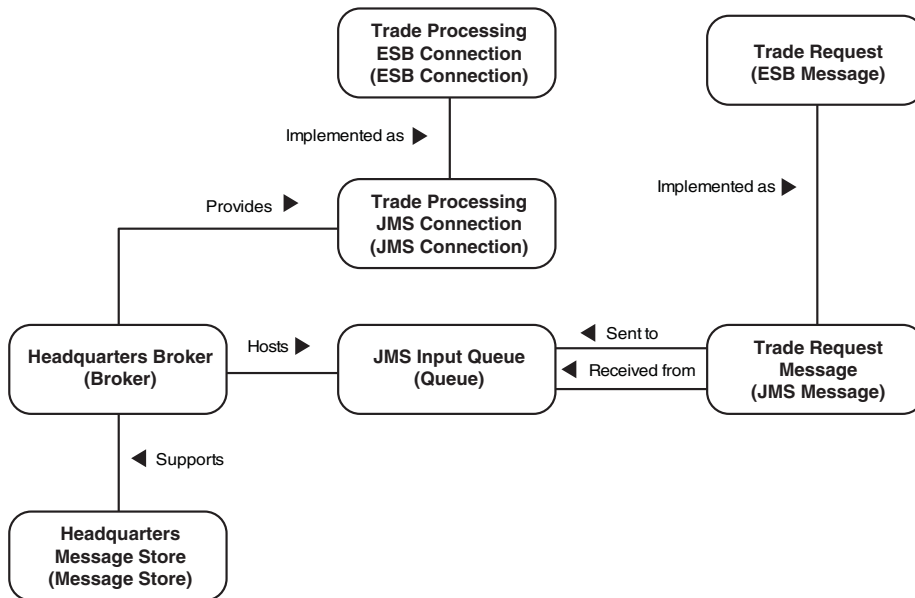
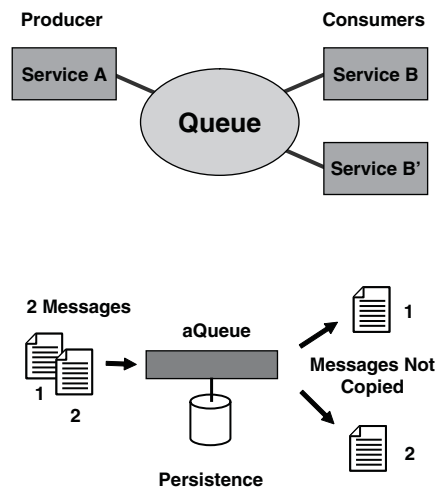


Figure 16. Example Object Diagram for SonicMQ

The example in Figure 16 shows the ESB connection reading from a queue in which another service writes. By using the publish-and-subscribe messaging model, the queue could be changed to a topic, and many services could read from it without making changes to the writer. This is another important aspect of loose-coupling in an SOA, in which the cardinality of service relationships can be changed without disruption of existing services. In the ESB this model is implemented by a JMS topic. Figure 17 illustrates the behaviors of queues and topics.

### Queues provide point-to-point



### Topics provide publish-subscribe

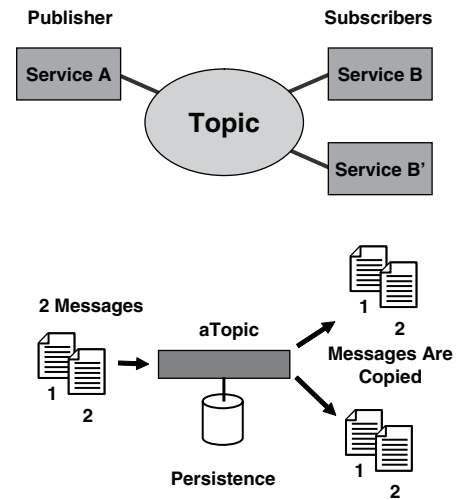


Figure 17. Queue and Topic Behavior

## > 3.7 WEB SERVICES

The ESB can make full use of Web services interoperability, including:

- > Exposing ESB services and processes as Web services for use by Web services clients.
- > Calling of Web services from within ESB processes.
- > Communications from within ESB processes to ESB services, processes, and endpoints using Web services interfaces and SOAP protocols.

Industry-standard WSDL (Web Services Definition Language) definitions can be created for ESB service, ESB process, and ESB endpoints. This allows the use of standard tools and integration with other SOA infrastructure components such as a UDDI registry. For example, in Figure 1 the entire trading process itself is exposed as a web service. The WSDL definition can be stored in either the ESB repository or in a UDDI registry.

Figure 18 shows the classes which provide web services functionality by the ESB. Inbound Web service requests are made through SOAP messages to the Web services acceptor on the broker, and forwarded to the entry endpoint for processes and services. Outgoing Web service requests are structured according to the Web services call file (created in the development tools, see section 4.1) and sent over Web Service Connection (implemented as a JMS connection) to a broker which manages the actual HTTP(S) connection. The broker also implements Web services standard (and proposed standard) protocols including:

- > WS-ReliableMessaging<sup>14</sup>: Provides reliable delivery for Web services (SOAP) messages.
- > WS-Addressing<sup>15</sup>: Provides for asynchronous messages to Web services (rather than only synchronous request-reply) as well as for fault handling. The WS-Addressing Endpoint Reference construct is very similar to the ESB Address, providing for straightforward interoperability with Web services.
- > WS-Security<sup>16</sup>: Provides authentication, signing, and encryption for Web services messages in entirety or in part.
- > WS-Policy<sup>17</sup>: Specifies in WSDL (by WS-PolicyAttachment<sup>18</sup>) the policy options needed by WS-ReliableMessaging, such as "Exactly-Once Ordered", and for WS-Security, such as "encrypted payload".

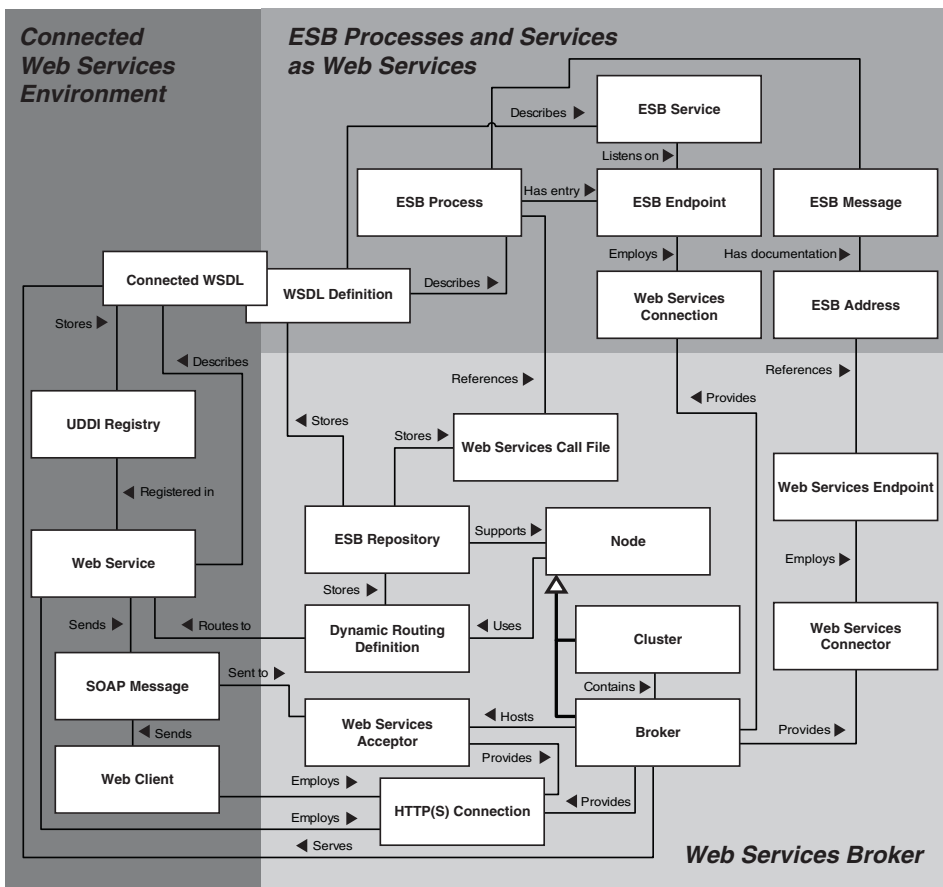


Figure 18. Class Diagram for Web Services

<sup>14</sup> Microsoft et. al., "Web Services Reliable Messaging Protocol (WS-ReliableMessaging)", <http://msdn.microsoft.com/library/en-us/dnglob-spec/html/WS-ReliableMessaging.pdf>, February 2005.

<sup>15</sup> World Wide Web Consortium, "Web Services Addressing (WS-Addressing)", <http://www.w3.org/Submission/ws-addressing/>, August 2004.

<sup>16</sup> OASIS, "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)", <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>, March 2004.

<sup>17</sup> Sonic Software, et. al., "Web Services Policy Framework (WS-Policy)", <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-policy.asp>, September 2004.

<sup>18</sup> Sonic Software, et. al., "Web Services Policy Attachment (WS-PolicyAttachment)", <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-policyattachment.asp>, September 2004.

The implementation of these protocols provides interoperability among all Web services connected through the ESB and ESB services exposed as Web services. When Web services need to communicate with a non-Web service, the ESB provides the mediation required to reconcile the different protocols and QoS models. Figure 19 shows an object diagram of an SOA application providing Web services through the ESB. It shows the sequence in an ESB service made available as a Web service being invoked by an external Web services client<sup>19</sup>.

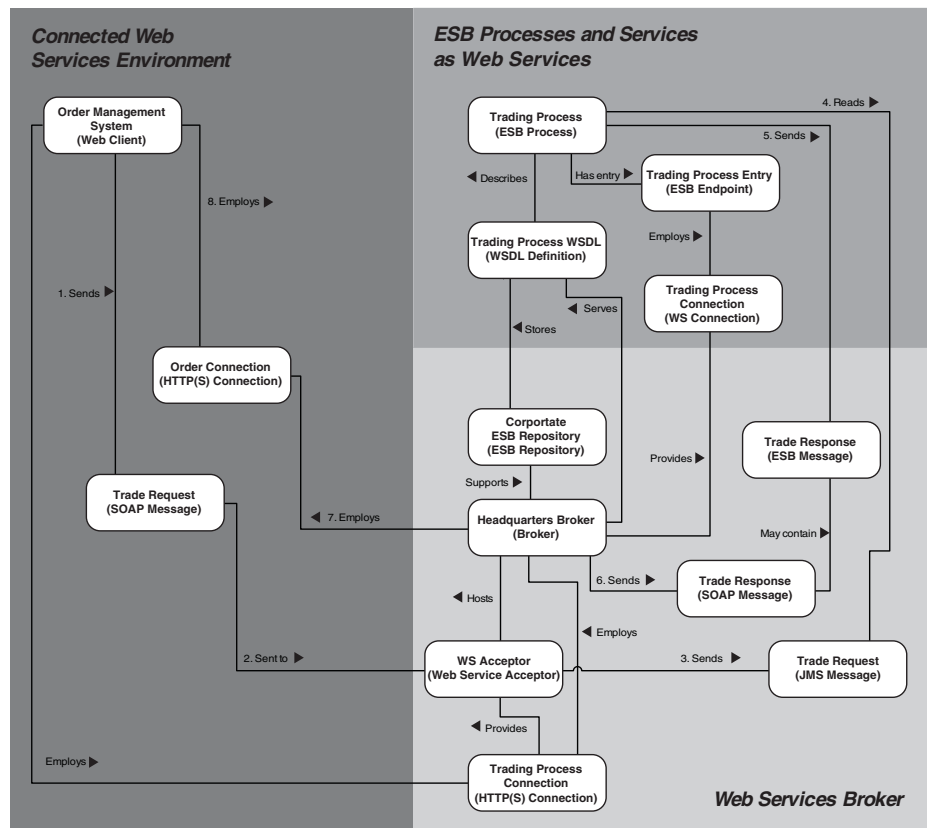


Figure 19. Object Diagram example for Web Services Communications

The ESB provides Web services brokering with all of the reliability, scalability, and distributed qualities inherent to the ESB.

<sup>19</sup> In resolving the ESB Address "Trade Process Entry" the ESB Dispatcher recognizes it as an ESB process and dynamically loads the ESB Process Definition into the ESB Message as an ESB Itinerary. The dispatcher then determines the first step in the Itinerary and addresses the ESB message to its entry point, start the process executing its first step.



> 4.0  
**ESB LIFECYCLE  
 AND DISTRIBUTED  
 SERVICES  
 ARCHITECTURE**

As will be described in section 4.3, the ESB extends these controls across multiple security domains in a distributed deployment. This allows for management of security infrastructure from any point on the network and provides a common layer of security for all service access. The ESB's security layer is flexible enough to link with different security requirements at each endpoint (for example, application services, .NET, or legacy environments). It can also provide the necessary mediation between different security models imposed by different organizations, allowing spanning business process to be conducted securely, while still abiding by the security requirements imposed by each specific organization.

Intrinsic to the ESB is rich support for the entire services lifecycle from development to deployment to management. A critical requirement is to allow such a migration to occur in the easiest error-free manner possible. In particular, the ESB allows the migration of services, processes, and resource files seamlessly between development, test and production systems, allowing administrators to modify deployment, policies, and connectivity without requiring changes to the services or processes themselves.

Figure 21 shows the SOA lifecycle stages overlaid above an outline of the full ESB class diagram in Appendix. This shows the flow and use of ESB components from development and configuration of services through to deployment in multiple environments and finally to the monitoring and management of production operations. This section will follow these stages across the diagrams explaining the tools and mechanisms of development, of service and process configuration, of deployment tools and the migration to production, and of the distributed services architecture supporting reliability, scalability, and manageability in the production environment.

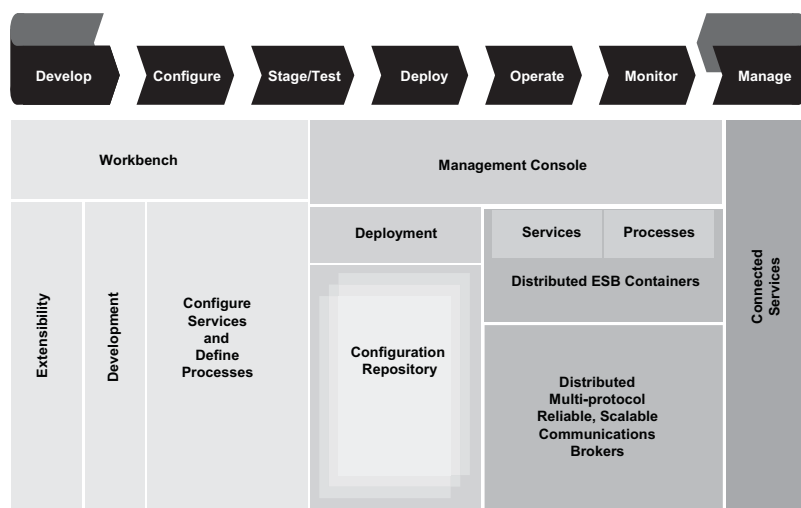


Figure 21. Overview of the ESB Lifecycle

Development and configuration of the ESB is done through the Sonic Workbench, an integrated services environment (ISE) which includes a number of editors, test tools, and the management console as shown in Figure 22.

## > 4.1 DEVELOPMENT AND CONFIGURATION

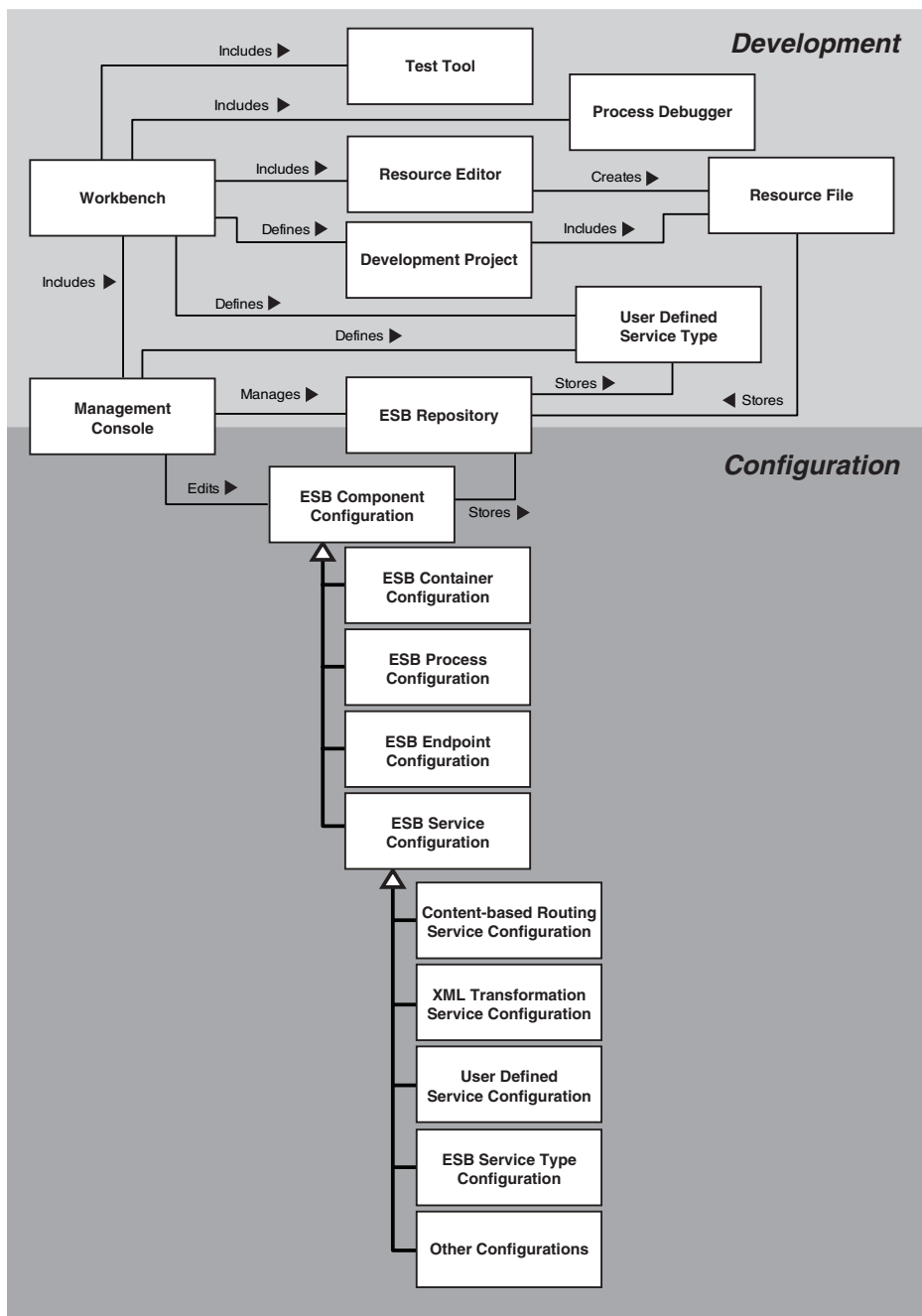


Figure 22. Class Diagram for Development Tools

---

The Sonic Workbench provides each developer with a set of tools for ESB development and deployment including creation of custom ESB service type implementations. The service type implementations include all of the code for execution of the custom service types; these implementations are distributed automatically through the distributed services architecture described in section 4.

For example, in Figure 1, the compliance system is a custom ESB service type. With a set of editors, in concert with third-party and industry-standard development environments, developers create resource files which are configurations stored in the ESB repository. Resource files are typically simple files containing such things as an XQuery, or XSLT, or custom configurations needed by custom services, such as a compliance rule set, for the custom compliance service.

The Sonic Workbench provides for the creation of development projects containing all of the configuration files for a specific service development project in support of a composite or ESB application. In the trading example this would include files describing the ESB process, the WSDL file used to expose the process as a Web service, and the compliance service implementation, among other resource files.

Test tools are provided to create scenarios and monitor the flow of messages through the ESB. Special support is given for the testing of Web services in the context of ESB applications.

The management console provides the management framework for configuring the infrastructure elements such as ESB services, ESB containers, and the broker. The management console allows for the configuration of new instances of ESB services and ESB containers to allow remote deployment of new functionality or remote deployment of additional services for increased throughput. For example, using this console, the connection type between all ESB Services in the Figure 1 trading process could be set to “once-and-only-once” to prevent the chance of message loss.

Figure 23 shows the class diagram for the most commonly-used resource editors in the Sonic Workbench.

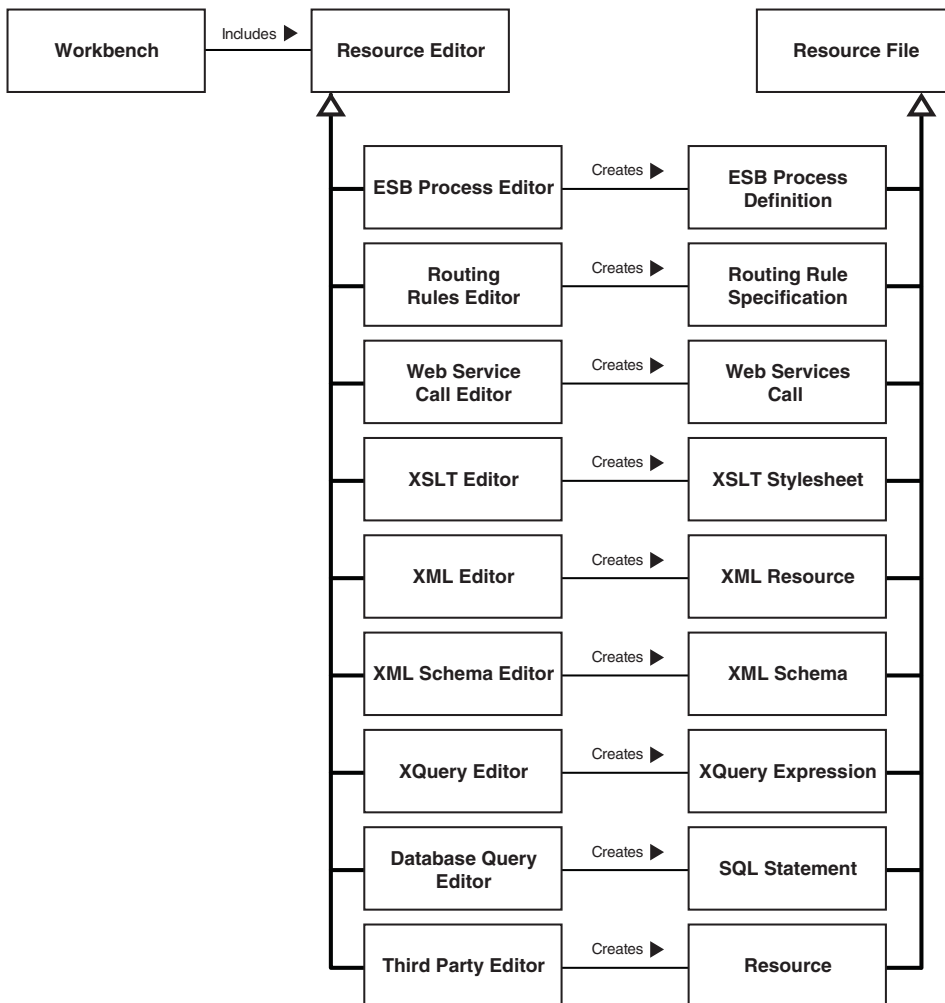


Figure 23. Class Diagram for Workbench Editors

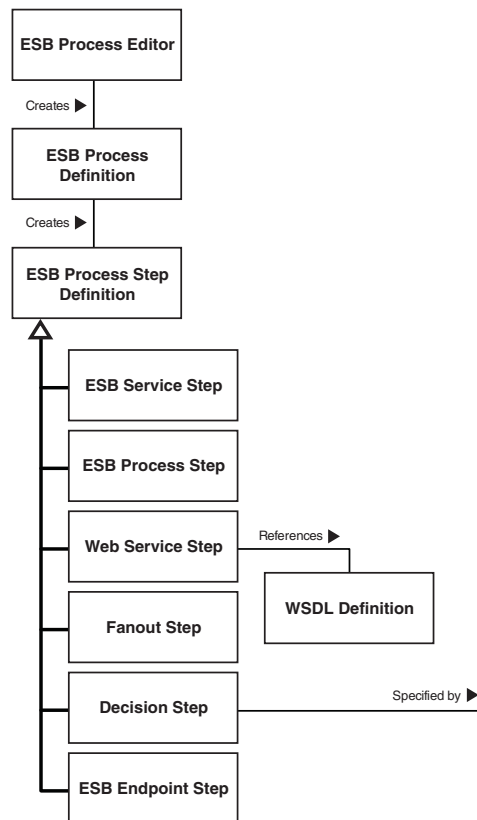
These editors include:

- > ESB Process Editor for visually creating and editing business process itineraries tying together multiple services to create a composite SOA application.
- > Routing Rules Editor to create the condition-destination routing rule specification used by content-based routing services.
- > XSLT Editor for creating the XSLT stylesheet defining an XML transformation for use by XML transformation services.
- > Web services Call Editor to define the mapping between WSDL interfaces and ESB services, processes, and endpoints in a Web services call file.
- > XML Editor for creating XML documents used by many services as XML resource files.

- > XML Schema Editor for defining an XML schema file used to define the types of data required for a Web services call.
- > XQuery Editor for creating XQuery expressions for use as queries against data structured as XML documents.
- > Database Query Editor for creating SQL statement resource files defining queries against relational database.

As a more detailed example of the types of specifications created by the editors, Figure 24 shows the class diagrams for two types of specifications: the ESB itinerary of an ESB process and the routing rules specifications for the content-based routing service.

**a. ESB Itinerary Specification**



**b. Content-based Routing Rules Specification**

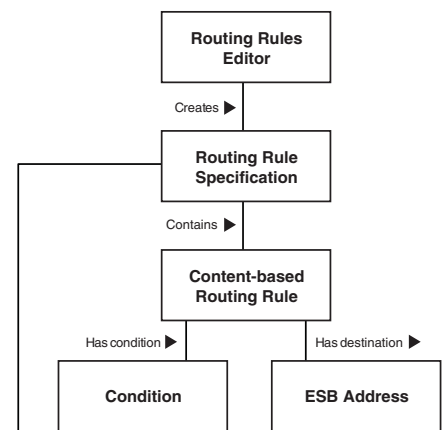


Figure 24. Class Diagram for Selected Mediation Specifications

The ESB process editor creates ESB process definitions which consist of a sequenced ESB itinerary of process steps passing messages from one step to the next. Each step is one of:

- > Service Step – Invokes an ESB service
- > Process Step – Invokes another ESB process

- > Web Service Step – Invokes a Web service
- > Fanout Step – Replicates the ESB message for delivery to multiple ESB addresses
- > Decision Step – Performs content-based routing among multiple ESB addresses
- > Endpoint – Delivers the message to a service endpoint

When composing services into a process, the default service address set configuration, and in particular the address for messages upon service exit, can be over-ridden for customized use within the context of the process.

Figure 25 shows the user interface for the process editor in an Eclipse-based<sup>20</sup> development environment.

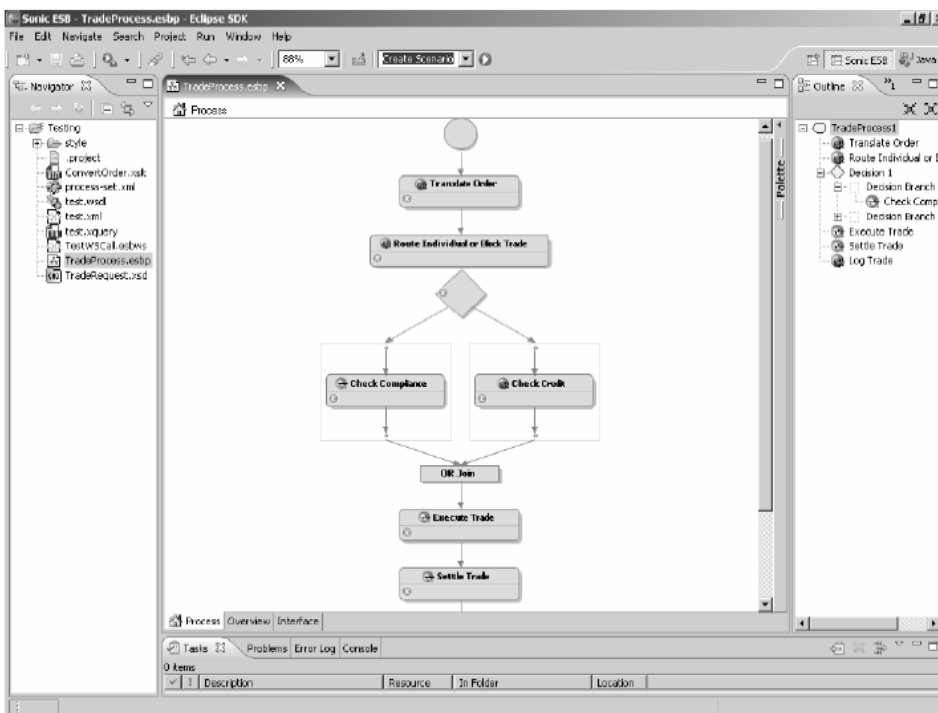


Figure 25. Example of ESB Development Tools - ESB Process Editor

A content-based routing rule contains a set of conditions with an associated message destination for the associated condition. The condition typically is a test on some value contained in the message received, or it may be derived from local data for a “context-based” routing decision. For example, consider a trading process in which a trade request is sent to one service if it is a stock trade and is sent to another service if it is an option trade. The user interface for the content-based routing rule editor is shown in Figure 26.

<sup>20</sup> eclipse.org, <http://www.eclipse.org/>.

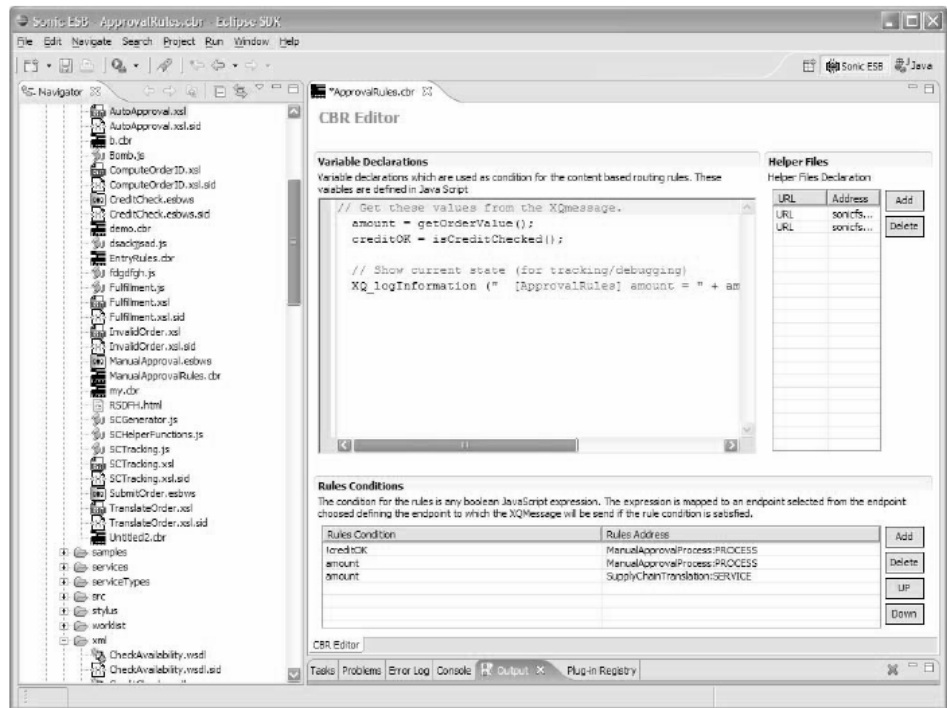


Figure 26. Example of ESB Development Tools - CBR Editor

Users can create completely new ESB service types to extend the mediation or business service capabilities of an ESB. Third-party vendors can also create ESB service types for use in the ESB. When creating a new service type, an Integrated Development Environment (IDE) is used in conjunction with Sonic Workbench to create all of the implementation files for the new service type.

The lifecycle of the ESB service types from configuration through deployment to production is the same regardless of ESB service type - including the ESB mediation services, XML transformation and content-based routing, as well as the advanced ESB services (section 3.3).

Given either pre-defined or user-defined service types, configuration is performed to prepare a service for deployment to an execution environment. The management console provides configuration controls to the standard service configuration items such as the entry endpoint and the exit ESB address. Additionally, configuration properties specific to that type of service are set.

For example a content-based routing service can be specialized with the specific rules shown in Figure 26.

ESB containers are configured by assigning services to each container and defining how many parallel threads of execution will be provided for each service in the container. (Each thread creates a separate dispatcher along with associated inbox, outbox, and faultbox for that thread's

instance of the service). Furthermore, multiple containers with the same services can listen on a single ESB endpoint providing transparent load balancing and redundancy.

As shown in Figure 22, these ESB component configurations are stored in the ESB repository (along with all of the resource files needed by the services). These configurations are typically first stored in a development ESB repository and then migrated to test and production as explained in the next section.

Moving configurations from development to test and staging to production environments is done with the deployment tool shown in Figure 27. Selected resource files and configurations are moved through the creation of intermediate archive files and the use of the deployment map file to map service and message parameters to operate properly in the targeted environment where quality of service settings, security policies and ESB addresses may differ. The tailoring rules file specifies which service parameters can be set in the deployment map.

## > 4.2 DEPLOYMENT TOOLS AND MIGRATION OF SERVICES AND CONFIGURATIONS

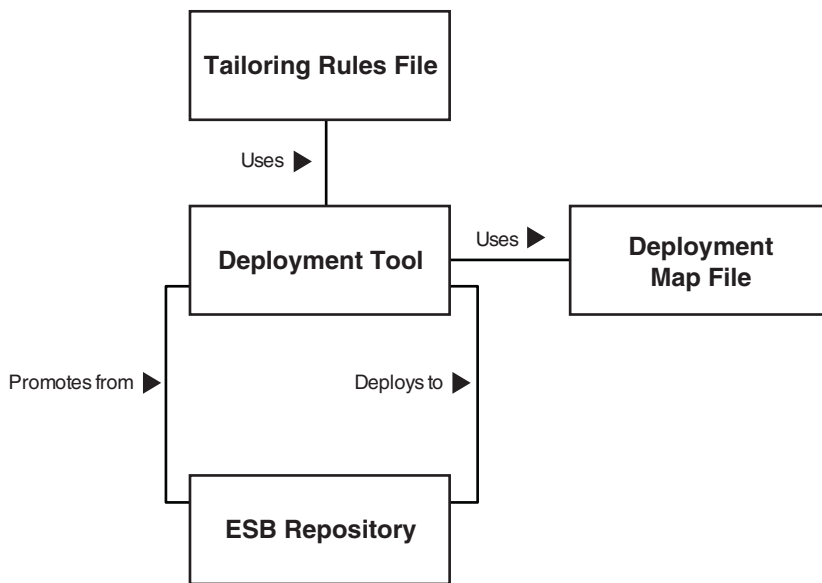


Figure 27. Class Diagram for Deployment Tool

Figure 28 shows the promotion and deployment of configurations from the development ESB repository to the test ESB repository to the production ESB repository.

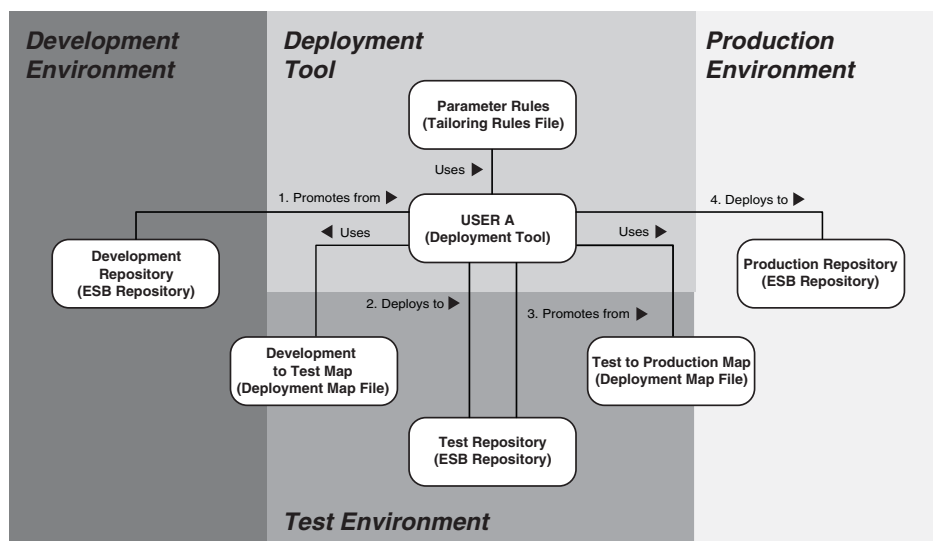


Figure 28. Object Diagram for Deployment Tool

This diagram shows a relatively simple case with one development environment, one test environment and one production environment. Many environments are more complex and may comprise multiple development environments, including individual developer workstations, multiple test environments including integration test, acceptance test, and staging, and production environments in many geographic locations. The ESB deployment tool can be used with appropriate map files for all of these situations, providing reliably reproducible deployment.

### > 4.3 DISTRIBUTED SERVICES ARCHITECTURE

The ESB provides flexibility, availability, scalability, and manageability through its distributed services architecture, which has the following important capabilities and properties:

- > New services and new service types can be implemented through configuration, without the need to write programs.
- > An ESB service interface can be changed without requiring coding changes and a consequent development cycle to recompile and deploy that service.
- > Service instances can be remotely provisioned in ESB containers located anywhere on the network, allowing flexibility to deploy new functionality and to manage performance through load-balancing or multiple service instances. Allows the deployment of services to the physical location where they may perform best, due to host processing capability or proximity to other resources.
- > Local access to configuration artifacts needed by services, providing quick service startup and ability to run even in the event that a network failure prevents access to the ESB repository.
- > Automatic distribution of ESB service implementation and configurations across a distributed network for deployment of new or changed services.
- > The distributed ESB service containers and communication brokers can be managed using the same underlying messaging infrastructure as that used for service communications, with the ability to partition communications for scalability and reliability.

- > Events, including entry and exit of services invoked by an ESB process, can be tracked to manage process execution across even a highly-distributed deployment.
- > Automatic load balancing across multiple instances of a service in support of performance and reliability goals.
- > CAA (Continuous Availability Architecture) provides high reliability multi-protocol messaging through back-channel replication of message data from primary to secondary brokers. This ensures continuous, ordered delivery of messages in the event a communication broker fails<sup>21</sup>.
- > DRA (Dynamic Routing Architecture) allows messages, including Web services requests and responses, to be automatically routed across a federated multi-hub deployment of brokers. This permits messages to enter any broker in the ESB, and be routed to the correct destination, regardless of the domain which hosts this destination, subject to enforcement of security rights<sup>22</sup>.

Figure 29 shows the key classes used to support distributed services architecture in a production environment. The classes are grouped according to: distributed ESB containers, distributed ESB management, distributed ESB broker, and connected business services.

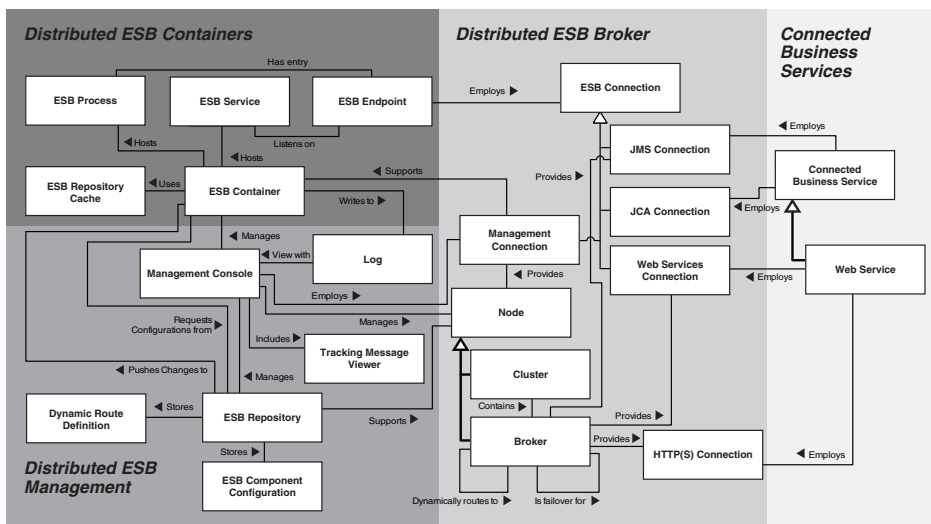


Figure 29. Class Diagram of Distributed Services Architecture

ESB service configurations and ESB container configurations are stored in the production ESB repository. As introduced in section 3.2, the ESB container instantiates ESB services and provides: the systems management interface for its services, the management of communication protocols, the capability to have multiple threads executing services for scalability, and the standard capability to execute distributed processes. The management console is used to place ESB containers and their configured services onto multiple distributed systems where interaction policies, transport bindings, and communication layer security policies are established.

<sup>21</sup> Sonic Software, "Continuous Availability for Enterprise Messaging, Reducing Operational Risk and Administration Complexity", April 2004.  
<sup>22</sup> Sonic Software, "Clustering and DRA in SonicMQ, Bringing Scalability and Availability to the Enterprise Backbone", January 2004.

---

The ESB provides highly reliable support for distributed systems. An ESB repository cache is used to provide local copies of the configuration files used by ESB containers and ESB services executing on distributed systems. The cache is populated by requests from the container to the ESB repository as well as changes pushed from the ESB repository. The cache provides the ability for distributed systems to be both fast and reliable, and allows continued independent operation even when the production ESB repository is unavailable or unreachable.

There is a separate management connection between each ESB container and the brokers used for management communications. This permits management communication messages to use a physically separate set of systems from the service communications messages. The management connections have security controls and configuration independent of the services' connections. Management functions are built making use of standard JMX (Java Management Extensions) capabilities.

A key capability of the ESB infrastructure is to allow the generation of tracking messages to monitor the progress of an ESB process through the entry and exit at each step of the process. These messages are sent from across the distributed environment to a common destination for review and monitoring. A process can be configured with a tracking endpoint for receipt of the tracking messages.

In addition, trace logs of all activities can be generated for review and monitoring. Activities which can be traced include ESB container events, endpoint tracing, message dispatching, ESB invocation, and application debugging messages. In addition to the ESB management console, third-party monitoring tools can be used.

Reliability and scalability of the communications infrastructure is provided through clustering of brokers, with linear scalability as brokers are added. Transactional fault-tolerance is achieved through failover brokers using the Continuous Availability Architecture.

Dynamic route definitions in the Dynamic Routing Architecture (DRA) allow local clusters to be connected in a global network, transparent to the ESB applications. The ESB security model supports cross-domain authorization and authentication for DRA.

An important advantage of the ESB's distributed services architecture is that the topology, clustering, failover configuration, and quality-of-service of the infrastructure can be changed without requiring any change to the services.

The flexibility for distribution of ESB services and ESB containers with a fault-tolerant and high-performance infrastructure is illustrated in Figure 30.

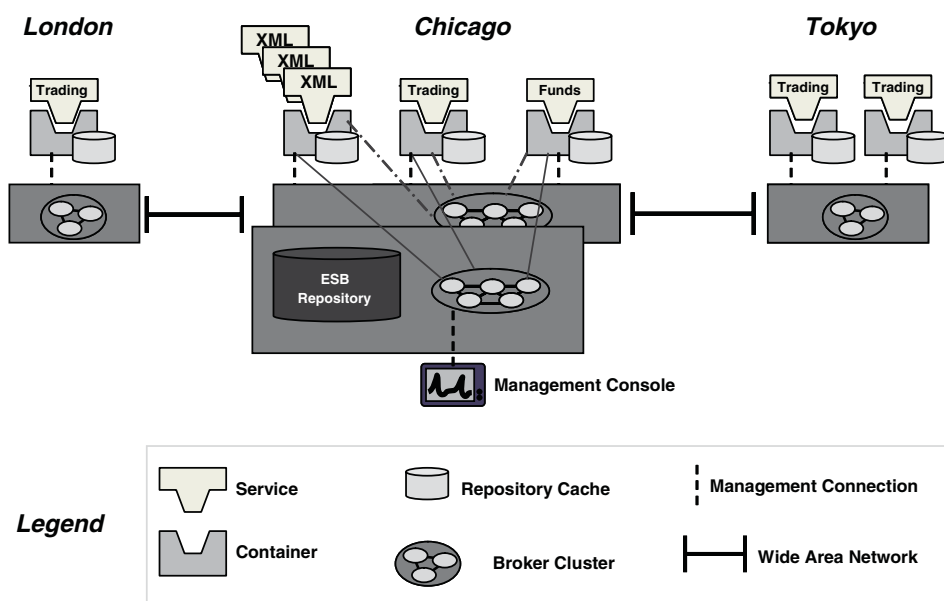


Figure 30. Distributed ESB Containers

This figure shows:

- > Multiple instances of a service in multiple distributed containers for reliability, performance, and load balancing
- > Multiple instances of a service in a single container for increased throughput
- > ESB repository cache at each container for performance and fault-tolerance
- > Dynamic routes (DRA) for transparent linkage between central and remote sites
- > Use of CAA with primary and secondary brokers for fault-tolerance
- > Different services in the same container for optimal messaging performance between the services
- > Management console using management connections for integrated view of entire distributed ESB

In this paper we described the architecture and lifecycle of the ESB. Key capabilities of this approach to development, deployment, and management include:

- > Rich set of development and configuration tools allow for easy configuration of ESB mediation services, processes, and the development of custom services. New business requirements can be implemented rapidly.
- > Deployment tools provide support for the ESB lifecycle and allow for the automated mapping of configurations and movement of implementation resources from development to staging/test to production environments.
- > Distributed execution and management environment that implements a distributed services architecture allows for flexible, high-performance, scalable, and continuously available SOA systems as illustrated in Figure 31.

## > 5.0 CONCLUSION

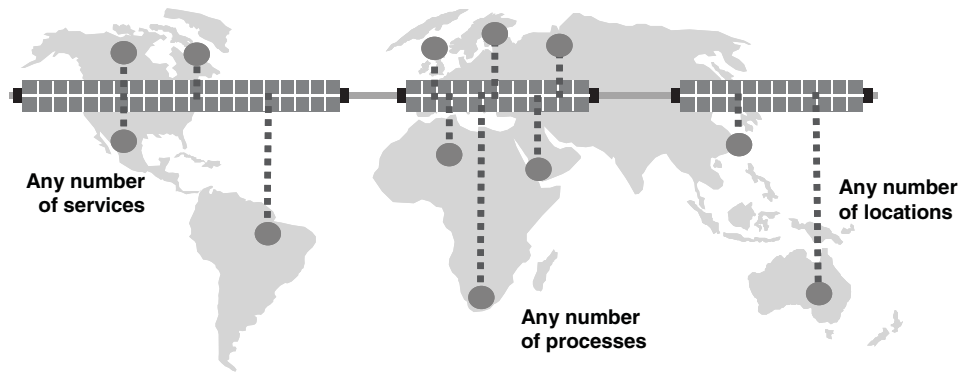


Figure 31. World-wide Distributed Services Architecture

In the book *Enterprise Integration Patterns*<sup>23</sup> defining standard patterns for enterprise application integration (EAI), the “message bus” is defined as the “architecture [which] enables separate applications to work together but in a decoupled fashion such that applications can be easily added or removed without affecting the others.” It is exactly this architecture which is provided by the ESB in an off-the-shelf “configure and go” reliable and scalable package.

Now we can answer the questions posed in the introduction.

- > How do I build processes which span new service-enabled applications as well as existing legacy systems?

By using the ESB to connect, mediate, and control both new ESB services and their interaction with existing applications integrated as connected services. Multiple forms of connections support existing applications which can be integrated into a seamless distributed business process. Synchronous and asynchronous applications or data sources can be integrated without any changes to the application or data implementations.

- > How do I provide the performance expected of enterprise systems while easily accommodating changes in demand?

By creating composite SOA applications implemented on the ESB infrastructure, SOAs can achieve increased reliability, scalability and performance. Distributed services architecture allows for the scaling of processing capacity with no change to the applications. Multiple instances of services can be dynamically deployed to process requests in parallel –with no changes to behavior of the services. Similarly, the supporting infrastructure can be dynamically reconfigured to increase throughput with no changes to the running processes or services. Services and infrastructure can be dynamically deployed when and where it is needed.

- > How do I isolate applications from faults resulting from server and communication failures?

<sup>23</sup> Hohpe, Gregor, and Wolf, Bobby, *Enterprise Integration Patterns*, Addison-Wesley, 2004.

---

The ESB distributed services architecture provides multiple layers of fault-tolerance including:

- > Services deployed to multiple containers on multiple machines in multiple locations
- > Broker clustering with Continuous Availability Architecture (CAA) failover
- > Distributed ESB repository cache able to continue operations if the ESB repository is unavailable
- > Distributed process implementation which operates – without a single central point-of-failure.

Each of these complementary approaches is transparent to the implementation of the services.

- > How do I manage processes that will interact with services spread across an organization, and between organizations?

The ESB allows for creation of business processes which can include both ESB services and connected services, including Web services. These services can be geographically and organizationally distributed. Unique capabilities including dynamic routing architecture (DRA) for connections across multiple clusters of nodes. The collaboration service extends the model to support commonly used business-to-business protocols such as ebXML. ESB itineraries provide a visually-defined and managed business process which spans domains in an organization's network infrastructure. Federated security permits full local access and deployment control as well as seamless global access and control across multiple security domains subject to the enforcement of local security rights.

- > How do I manage and monitor the infrastructure as well as the processes and services deployed within it?

The ESB supports distributed deployment of ESB services and processes to any ESB container or group of containers. It provides management and monitoring of the SOA infrastructure through management capabilities integrated into the deployed containers controlled through the management console. Processes, services, and containers are instrumented with tracking capabilities at both the message and composite SOA application level.

- > How do I allow the organization to change processes, rules, data mapping and relationships between applications with minimal effort and disruption?

An ESB provides the loose coupling required to achieve the flexibility promised by SOA. Services are mediated by the semantics of asynchronous messages; the messages they exchange are mediated by the transformation and other services hosted by the ESB. The logic controlling the sequence of execution, the destination of message delivery, and the recovery from errors is kept formally separate from the service's business logic. These architectural practices eliminate the hidden dependencies between services which make change difficult.

Processes, data mapping, and relationships between applications such as through content-based routing are implemented in the ESB with interactive tools through configuration rather than programming. This allows for easy modifications and incremental change. Changes can be deployed to the production environment through the deployment tool and then automatically propagated

---

throughout the distributed ESB. New services can be added and processes modified without requiring changes to the existing services. The ESB provides flexibility to change the quality of service of existing services to allow for new uses without requiring any changes to the existing service.

As an organization introduces SOA to their systems development architecture and processes, it is dependent on people, processes and technology to move through the levels of SOA maturity from initial services to optimized business services. The Sonic ESB and entire SOA Suite provide the proven SOA infrastructure to support this process, enabling business agility and transformation at the same time it promotes cost reduction through service reuse.

The Sonic ESB provides connection, mediation, and control of services and their interaction through visually modeled business processes. Its distributed services architecture supports the agility, scalability, availability and reach needed to respond to ever-changing business requirements in a complex enterprise. Combined with the development, deployment and management tools to support comprehensive lifecycle management, Sonic ESB fulfills the promise of enterprise SOA.

For more details on how the ESB sets the standard for business agility, global reach, service scalability and availability please call 1 866 GET SONIC (1-866-438-7664) or send email to [eval@sonicsoftware.com](mailto:eval@sonicsoftware.com).

### **Activity Definition**

A process definition in the form of a UML Activity Diagram created with the Activity Editor and stored in the ESB Repository. Used by Orchestration and Collaboration Service to execute business processes.

### **Application Server Application**

A connected business service implemented on J2EE or .NET application servers.

### **Broker**

A communication server which provides reliable communications between ESB Endpoints using a variety of protocols including web services. The Sonic ESB uses the SonicMQ multi-protocol Broker to provide continuous communications even in the event of a server failure.

### **Cipher Suite**

A suite of encryption software for securing JMS and HTTPS connections.

### **Cluster**

A group of Brokers which behaves as a logical Node and shares processing load. Topics and queues are made available to all brokers in a cluster. A cluster can be configured for Continuous Availability.

### **Collaboration Service**

An optional ESB Service Type provided by Sonic for modeling, running and managing business processes with partners using B2B protocols. The processes are defined with an Activity Editor and stored as Activity Definitions.

### **Condition**

An XPath expression or JavaScript Rule which defines a condition for routing messages based on their content or context.

### **Connected Business Service**

A generalization of any service which is hosted outside the ESB but is connected through some protocol and transport so that it can interact with ESB Services and ESB Processes. Examples include Web Services and Web Clients, Application Server Applications, JMS Applications and Enterprise Systems.

### **Content-based Routing Rule**

An individual rule for routing messages based on a True/False test contingent on particular message content.

---

### **Content-Based Routing Service**

A prepackaged ESB Service Type which offers an XPath- or JavaScript-based content-based routing service. One of the most commonly used ESB Service Types. It is used by the Decision Step in an ESB Process.

### **Database Service**

An optional ESB Service provided by Sonic for service enabling relational databases. SQL Queries are created with the Database Query Editor and configured to run with a set of database connection parameters. Supports high performance query and update of database including: DB2, Oracle, Sybase, MS SQL Service and any JDBC compatible RDBMS.

### **Deployment Map File**

A file contains rules for transforming identifiers and properties within deployment artifacts so they match the target ESB domain into which they are being promoted.

### **Deployment Tool**

A graphical tool for selecting and extracting development artifacts (resource files, configuration data and service implementations) for migration from development to test to production within different ESB domains.

### **Development Project**

A set of related implementation and resource files and ESB component artifacts that will be developed, tested and deployed as a project. Stored in the ESB Repository as a folder with subfolders for development, test and production uses.

### **Dispatcher**

A listener within the ESB Container which is configured to read messages on behalf of an ESB Service as well as to process ESB Itineraries. Before the ESB Service is invoked the dispatcher fills the Inbox and following execution of the service it reads the Outbox and Faultbox for results. If the service or dispatcher fails, the original message is not acknowledged and the transaction rolls back. If the service succeeds, the dispatcher acknowledges receipt and the message processing transaction completes. The service may put multiple messages in each of the Outbox and Faultbox boxes. Each message contains an address which is resolved by the dispatcher.

### **Dynamic Route Definition**

Configuration data used to define routing rules for message flow between Node, often used to provide forwarding between different locations. It can be used to map an ESB Address to a Web Services URL such that the external Web Service consumers and providers can directly address an ESB Service or ESB Process.

---

## **Enterprise System**

A connected business service exposed as an ESB Endpoint but implemented using either an ESB Service and a custom interface or one of the other types of ESB Endpoints. Once connected, these Enterprise System Services can be used within ESB Process Definitions as a normal ESB Endpoint.

## **ESB Address**

A reference to an ESB Service, ESB Process or ESB Endpoint. Is resolved through lookup by the dispatcher in the ESB Repository Cache.

## **ESB Component Configuration**

A generalization of all configuration data which is used to specify the name, parameters and related properties of an ESB Service, ESB Process, ESB Container or ESB Endpoint.

## **ESB Connection**

A component which provides a logical abstraction and a configurable layer of indirection between an ESB Endpoint and the physical connection(s) used to deliver messages.

## **ESB Container**

A Java-based management container that hosts ESB Services, Dispatchers and ESB Endpoints. The container requires IP connectivity and a management Connection to the ESB Repository for loading its configuration data and obtaining resource files. The management connection also is used for sending lifecycle commands and collecting systems management events.

## **ESB Container Configuration**

Configuration data containing the ESB Container name, its properties and a list of ESB Services which are to be hosted in the container.

## **ESB Endpoint**

A component which provides a logical abstraction and a configurable layer of indirection between an ESB Service and the underlying transports. The endpoint defines the quality of service (QoS) for both senders and receivers.

## **ESB Endpoint Configuration**

Configuration data comprising the ESB Endpoint name and its type (JMS, Web Service, or JCA).

## **ESB Itinerary**

The part of an ESB Message containing the remaining Process Steps in an ESB Process Definition. Each Process Step may contain associated runtime parameters which override the Initialization Parameters of that Service. ESB Itineraries may also include decision points and branching instructions.

---

**ESB Message**

A normalized message used by the ESB between two ESB Endpoints. It contains headers and parts with specific content types. Messages which are part of an ESB Process will contain an ESB Itinerary to control the process.

**ESB Process**

A sequence of process steps which are defined in a Process Editor and embedded in ESB Messages as an ESB Itinerary to control the flow of service invocations across a distributed set of ESB Containers.

**ESB Process Configuration**

Configuration data containing the ESB Process name, its entry, exit, fault and rejected message addresses along with other properties such as QoS and Time-to-Live.

**ESB Process Editor**

A graphical editor for creating ESB Process Definitions including the ESB Process Step Definition.

**ESB Process Step Definition**

The development-time specification of an individual Process Step in an ESB Process Definition. Contains the name of the step, an ESB Address and related Runtime Parameter values. There are at least six Step types including: ESB Service, ESB Process, Web Service, Fanout, Decision and ESB Endpoint.

**Process Step**

The ESB Address and runtime parameters necessary to invoke the next step of an ESB Itinerary.

**ESB Repository**

A configuration repository for storing ESB Service Type implementations, Resource Files, and related configuration artifacts. Each ESB installation has an ESB Repository which controls the unique naming of ESB Services, ESB Processes and ESB Endpoints. Used during development by Editors related tools and at runtime by the Dispatcher for dynamic deployment of artifacts and ESB Addresses resolution.

**ESB Repository Cache**

A data store maintained by each distributed ESB Container for local copies of configuration data, resources files and ESB Service implementation files. Provides resilience in processing when communications with the ESB Repository are interrupted.

**ESB Service**

An instance of an ESB Service Type, configured and running in a deployed ESB Container.

---

## **ESB Service Configuration**

Configuration data containing the name and initialization parameters needed to start an ESB Service (of any type) running within an ESB Container. Included are the entry, exit, fault, and rejected message addresses. At runtime these configurations may be overridden by ESB Itineraries containing Runtime Parameters. This allows ESB Processes to change the behavior of configured ESB Services to meet a specific context.

## **ESB Service Type**

A generalization of all ESB Services and their implementations. ESB Service Types come pre-packaged, as in the case of the Content-based Routing Service or can be User-Defined. For example a File-polling ESB Service Type is frequently implemented to allow files to be collected and delivered as ESB Messages. Each ESB Service Type has a set of resource files, allowable parameters (initialization and runtime) and an XML definition file to register the service type in the ESB Repository. Once an ESB Services Type is created it is automatically available for use as a Process Step within ESB Process Descriptions and will be dynamically deployed to any and all ESB Container(s) where references to the Service Type appear within an ESB Itinerary.

## **Faultbox**

An object within the ESB Container which the Dispatcher reads at the end of an ESB Service invocation. The contents of the Faultbox will be the error messages from the service.

## **HTTP(s) Connection**

The standard application protocol for World Wide Web communications. HTTPS is secured and encrypted by using SSL as the HTTP transport.

## **Inbox**

An object within the ESB Container which the Dispatcher fills prior to invoking an ESB Service. The contents of the Inbox will be the standard input to the service.

## **JavaScript Routing Rule**

A resource file optionally used by the Content-Based Routing Service Type used in making routing decisions according to the execution of the JavaScript rule against the contents of the message.

## **JCA Adapter**

A standards-based Java Connectivity Architecture connection between J2EE applications and the ESB Container. Implemented through extensibility of ESB Endpoint Type components (not treated in this Definition document).

## **JCA Endpoint**

A specialization of ESB Endpoint for communicating with JCA-compliant Application Server Applications.

---

**JMS Application**

A connected business service exposed as an ESB Endpoint and implemented using the Java Message Service. Typically these applications are hosted in an application server external to the ESB and connected directly to the ESB through a JMS Connection to a JMS Destination (Topic or Queue). ESB Process Definitions can employ a JMS Destination as a normal ESB Endpoint within an ESB Itinerary.

**JMS Connection**

A connection between an ESB Endpoint and the Broker for reliable and scalable communication of JMS Messages.

**JMS Destination**

A named destination (Topic or Queue) hosted by a Node.

**JMS Message**

A Java object used to encapsulate messages of any type including: text, XML, binary and multi-part. ESB Messages are implemented as multipart JMS messages when they are configured to flow across an ESB Connection which uses JMS.

**LDAP Security Repository**

Short for Lightweight Directory Access Protocol, a set of protocols for accessing information directories.

**Log**

A disk file that is appended to by the ESB Container as events occur. For example, tracing events such as ESB Service invocation and exit.

**Management Connection**

A JMS Connection used by the Management Console, distributed ESB Containers, the ESB Repository, and Nodes for reliably transmitting systems management, configuration management, and monitoring actions.

**Management Console**

An administrative console for creating, configuring, and monitoring both ESB and Broker components.

**Message Store**

The persistent data store for unconsumed messages in a Broker.

---

**Node**

The identity of a communications Broker or Cluster. Dynamic Routing Definitions enable the movement of messages from one Node to another over JMS Connections.

**Orchestration Service**

An optional ESB Service Type provided by Sonic for modeling, execution and management of business processes using centrally-stored process state. The processes are defined with an Activity Editor and stored as UML Activity Diagrams.

**Outbox**

An object within the ESB Container which the Dispatcher reads following an ESB Service invocation. The contents of the Outbox will be the standard output from the service.

**Process Debugger**

A test tool for single stepping through the execution of an ESB Itinerary.

**Queue**

A communications channel that delivers a message to a single ESB Endpoint.

**Resource Editor**

A graphical component in the Workbench for creating, editing and debugging Resource Files. There are a number of such editors including: ESB Process Editor, Routing Rules Editor, Web Service Call Editor, XSLT Editor, XML Editor, XML Schema Editor, XQuery Editor, Database Query Editor, XAction Editor, and Third Party Editor.

**Resource File**

A file that supports an ESB Service Type implementation and is employed at runtime. It can be created by an Editor and is stored in the ESB Repository. An example would be an XSLT stylesheet used by the XML Transformation Service Type. Resource files have URLs and can be loaded dynamically by an ESB Service at initialization or runtime using an ESB Itinerary.

**Routing Rules Specification**

A resource file used by the Content-Based Routing Service Type used in making routing decisions based on the content of the message. Each rule has a condition and a destination ESB Address selected when the Condition is true.

**Runtime Parameter**

A set of parameters associated with an ESB Itinerary which override the Initialization Parameters of the itinerary's ESB Services at runtime. These are authored in the Process Editor. For example a Runtime Parameter could override the initialized XSLT Stylesheet for a given XML Transformation Service. Initialization Parameters permit ESB Services to be easily reused in multiple ESB Processes.

---

**SOAP Message**

A standards-based XML document which is compliant with the Simple Object Access Protocol standard. Used as a normalized message format between Web services.

**SQL Statement**

A JDBC-compatible instruction for querying or updating a relational database.

**Tailoring Rules File**

Specifies which service parameters can be set in the deployment map.

**Test Tool**

A client application for creating and sending test messages to ESB Services.

**Topic**

A communication channel that delivers copies of a message to multiple ESB Endpoints.

**Tracking Message Viewer**

A graphical tool for collecting and analyzing message flow from a given ESB Process or Service.

**Trust Store**

A keystore file containing the trusted SSL certificates.

**UDDI Registry**

A Universal Directory Discovery Interface (UDDI) compatible registry of Connected WSDL files and related artifacts. Used by the Web Service Call Editor to obtain WSDL files during the development of a Web Service Call resource file.

**User-defined Business Service**

An ESB Service Type created by a user to perform business logic within the ESB. Examples in Figure 1 in this Definition document include: Compliance, Trading and Funds Transfer ESB services.

**User-defined Mediation Resource**

A custom resource file defined by a user and run by a User-defined Mediation Service type.

**User-defined Mediation Service**

An ESB Service Type created by a user to extend the mediation capabilities of their ESB. Examples include binary file transform, file drop, process aggregation, and protocol bridges to enterprise messaging systems, SMTP, EDI and AS2.

**User-defined Resource**

A resource file defined by a User and run by a User-defined Business Service.

---

### **User-defined Service Type**

Any ESB Service Type whose implementation was developed by a User of the Sonic ESB. Service types are loosely categorized as being either for mediation or business logic.

### **Web Client**

A client application that requests a service by using an HTTP(s) Connection. It may expect to receive its reply synchronously and immediately over the same connection. For accessing ESB Services as Web Services, it uses the SOAP protocol.

### **Web Service**

A connected business service supporting standard Web service protocols. It is exposed in the ESB through a proxy ESB Service Type (not in Definition document). Typically hosted external to the ESB and connected through a Web Services Endpoint and HTTP Connection. ESB Services and ESB Processes can be exposed as Web Services in their own right through the use of WSDL Definition. ESB Process Definitions can employ Connected Web Services as Process Steps.

### **Web Service Call Editor**

A graphical editor for creating Web Service Call parameters which are used in Web Service Steps within ESB Process Definitions. Reads Connected WSDL file as the basis for mapping SOAP Messages into ESB Messages.

### **Web Services Acceptor**

An HTTP daemon (Web server) hosted by the Broker and supporting HTTP(S) Connections with Web Service and Web Client connected applications. Routes messages to the Broker for delivery to JMS destinations employed by ESB Endpoints.

### **Web Services Call**

A configuration file for mapping ESB messages parts to SOAP Message parts. This file is created within the Web Service Call Editor and stored in the ESB Repository. Once a Web Service Call is created it can be used in any ESB Itinerary regardless of the location of the ESB Container in which that Process Step executes. Compatible with WS-Reliable Messaging.

### **Web Services Connection**

A variation of an ESB Connection which employs standards-based HTTP(S) connection for communications between the Broker and an external Web Service or Web Client. In the case of asynchronous messaging there would be separate HTTP(S) Connections for the reply.

### **Workbench**

An Integrated Services Environment (ISE) for configuring ESB Service Types and ESB Processes. It includes various editors, debuggers, testing tools, and related deployment tools.

---

**Worklist Service**

An optional ESB Service Type provided by Sonic for managing the human-workflow tasks associated business processes of the Collaboration and Orchestration Service Types.

**WSDL Definition**

Web Services Description Language (WSDL) file which defines an ESB Service or ESB Process for use by Connected Services (Web Service and Web Client). See also Connected WSDL.

**XML Action List**

A script created by a User with XAction Editor and run by the XML Service to execute a pipeline of XML processing steps.

**XML Editor**

A graphical editor for creating XML Resource files.

**XML Resource**

An XML file used as a resource to define a User Defined ESB Service.

**XML Schema**

An XML Schema (.XSD) file.

**XML Schema Editor**

A graphical editor for creating, editing, and validating XML Schema files.

**XML Service**

An optional ESB Service provided by Sonic for storing XML documents. The service runs scripts called XML Actions to pipeline a series of storage, retrieval, query and transformation steps into one service invocation. Used most often for message aggregation.

**XML Transformation Service**

A pre-packaged ESB Service Type which offers a configurable XSLT processing engine to transform XML documents within an ESB Message from one format to another or split the document into multiple parts, with each part becoming a separate ESB Message. One of the most commonly used ESB Service Types.

**XQuery Editor**

A graphical editor for creating, editing, testing and debugging XML Query Expressions.

**XQuery Expression**

An ASCII file with an XQuery expression.

---

**XSLT Editor**

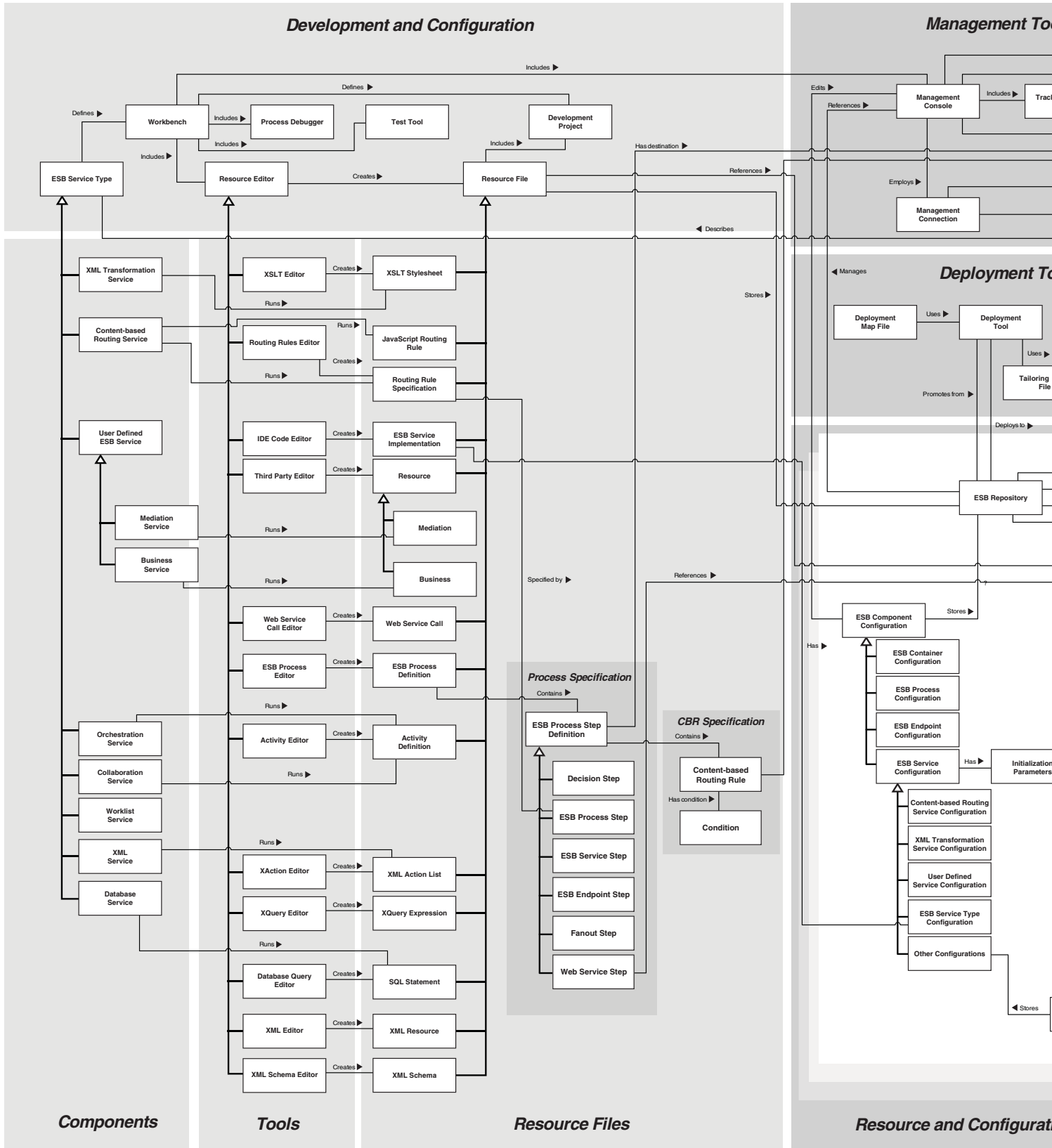
A graphical editor for creating, editing, testing and debugging XSL transforms.

**XSLT Stylesheet**

A resource file used by the XML Transformation and XML Service Types. Based on the Transform subset of the eXtensible Stylesheet Language (XSL) functional programming language.

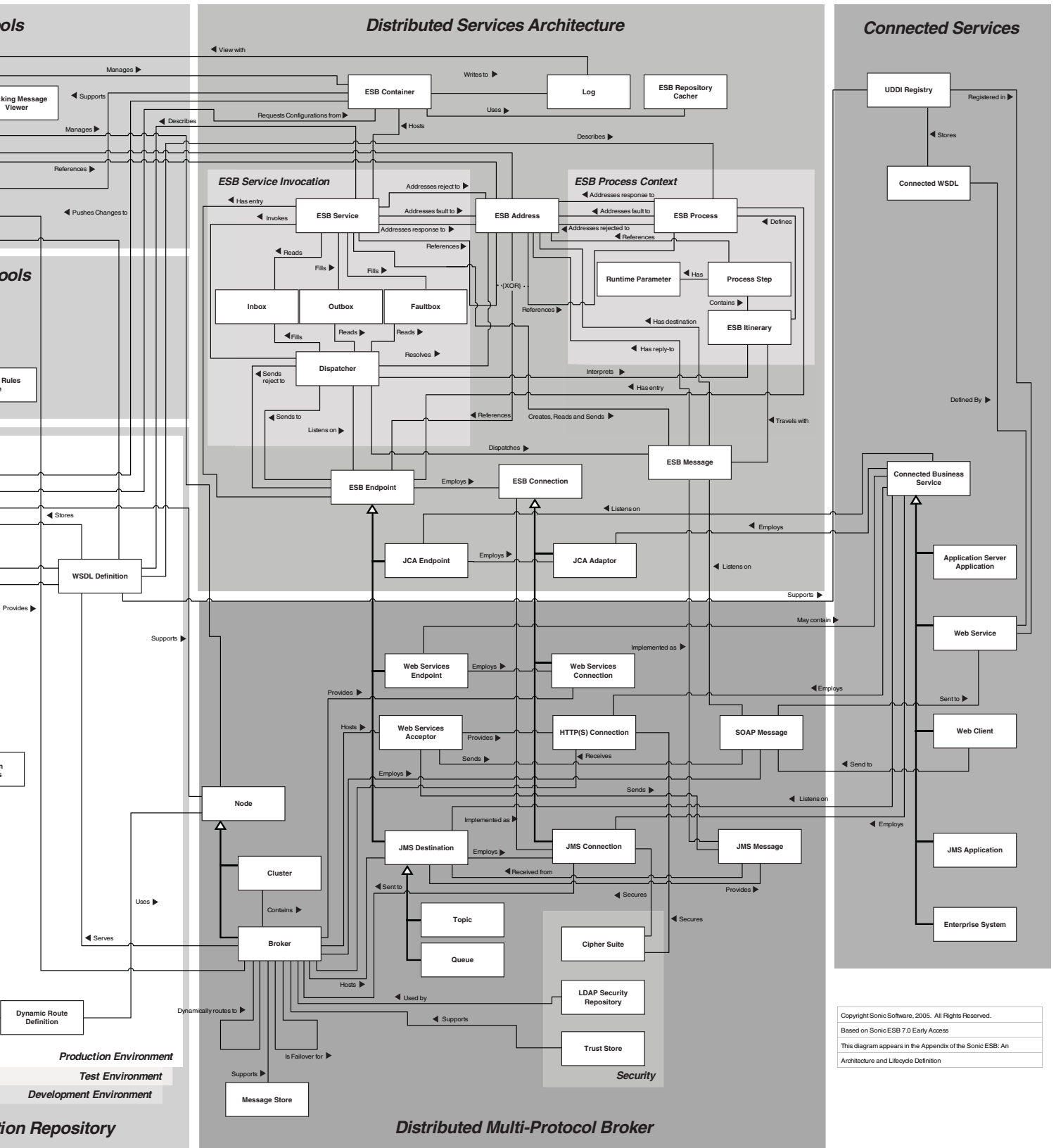
# Sonic ESB: Architecture

## Complete UML



# and Lifecycle Definition

## Class Diagram



Copyright Sonic Software, 2005. All Rights Reserved.  
 Based on Sonic ESB 7.0 Early Access  
 This diagram appears in the Appendix of the Sonic ESB: An Architecture and Lifecycle Definition

