Artix™

Building Service Oriented Infrastructures with Artix

Version 4.0, March 2006

Making Software Work Together™

Updated: 28-Apr-2006

Contents

List of Figures

Preface

What is Covered in this Book

This book discusses what makes a service oriented architecture (SOA), the advantages of SOA to integration, and how Artix facilitates the deployment of an enterprise quality SOA. It illuminates the value of a SOA. It shows how an ESB such as Artix plays a key role in developing a SOA and how Artix, in particular, provides the features required to build a distributed, robust collection of services.

The book then goes on to provide a detailed look at the distributed, extensible architecture of Artix. It discusses how Artix endpoints implement services. This discussion includes a discussion of how the plug-in architecture makes it easy to add functionality to an endpoint. It also provides a detailed discussion of many of the internal components of the Artix runtime.

Who Should Read this Book

While this book does contain some highly technical discussions, much of the book is geared toward a novice reader. A basic knowledge of distributed computing concepts is assumed.

How to Use this Book

This book is organized as follows:

- Chapter 1 provides a general description of service-oriented architectures and how enterprise service buses make them possible. It also discusses how Artix, in particular, fits into this picture.
- Chapter 2 provides a high-level description of Artix's architecture. It looks at how Artix connects endpoints to a network using its pluggable messaging stack.

- Chapter 3 provides a high-level description of how some of the enterprise features in Artix are implemented. It looks at what components are used and how they are deployed.

The Artix Library

The Artix documentation library is organized in the following sections:

- Getting Started
- Designing and Developing Artix Solutions
- Configuring and Deploying Artix Solutions
- Using Artix Services
- Integrating Artix Solutions
- Integrating with Enterprise Management Systems
- Reference Documentation

Getting Started

The books in this section provide you with a background for working with Artix. They describe many of the concepts and technologies used by Artix. They include:

- Release Notes contains release-specific information about Artix.
- Installation Guide describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- Getting Started with Artix describes basic Artix and WSDL concepts.
- Using Artix Designer describes how to use Artix Designer to build Artix solutions.
- Artix Technical Use Cases provides a number of step-by-step examples of building common Artix solutions.

Designing and Developing Artix Solutions

The books in this section go into greater depth about using Artix to solve real-world problems. They describe how Artix uses WSDL to define services, and how to use the Artix APIs to build new services. They include:

- Building Service-Oriented Architectures with Artix provides an overview of service-oriented architectures and describes how they can be implemented using Artix.
- Understanding Artix Contracts describes the components of an Artix contract. Special attention is paid to the WSDL extensions used to define Artix-specific payload formats and transports.

- Developing Artix Applications in C++ discusses the technical aspects of programming applications using the C++ API.
- Developing Advanced Artix Plug-ins in C++ discusses the technical aspects of implementing advanced plug-ins (for example, interceptors) using the C++ API.
- Developing Artix Applications in Java discusses the technical aspects of programming applications using the Java API.

Configuring and Deploying Artix Solutions

This section includes:

- Configuring and Deploying Artix Solutions discusses how to configure and deploy Artix-enabled systems, and provides examples of typical use cases.

Using Artix Services

The books in this section describe how to use the services provided with Artix:

- Artix Locator Guide discusses how to use the Artix locator.
- Artix Session Manager Guide discusses how to use the Artix session manager.
- Artix Transactions Guide, C++ explains how to enable Artix C++ applications to participate in transacted operations.
- Artix Transactions Guide, Java explains how to enable Artix Java applications to participate in transacted operations.
- Artix Security Guide explains how to use the security features of Artix.

Integrating Artix Solutions

The books in this section describe how to use Artix as a bridge between other middleware technologies and service-oriented middleware technologies.

- Artix for CORBA provides information on using Artix in a CORBA environment.
- Artix for J2EE provides information on using Artix to integrate with J2EE applications.

For details on integrating with Microsoft's .NET technology, see the documentation for Artix Connect.

Integrating with Enterprise Management Systems

The books in this section describe how to integrate Artix solutions with a range of enterprise management systems. They include:

- IBM Tivoli Integration Guide explains how to integrate Artix with IBM Tivoli.
- BMC Patrol Integration Guide explains how to integrate Artix with BMC Patrol.
- CA WSDM Integration Guide explains how to integrate Artix with CA's WSDM product.

Reference Documentation

These books provide detailed reference information about specific Artix APIs, WSDL extensions, configuration variables, command-line tools, and terminology. The reference documentation includes:

- Artix Command Line Reference
- Artix Configuration Reference
- Artix WSDL Extension Reference
- Artix Java API Reference
- Artix C++ API Reference
- Artix .NET API Reference
- Artix Glossary

Getting the Latest Version

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Artix Library

You can search the online documentation by using the **Search** box at the top right of the documentation home page:

http://www.iona.com/support/docs

To search a particular library version, browse to the required index page, and use the **Search** box at the top right, for example:

http://www.iona.com/support/docs/artix/4.0/index.xml

You can also search within a particular book. To search within a HTML version of a book, use the **Search** box at the top left of the page. To search within a PDF version of a book, in Adobe Acrobat, select **Edit**|**Find**, and enter your search text.

Artix Online Help

Artix Designer and the Artix Management Console include comprehensive online help, providing:

- Step-by-step instructions on how to perform important tasks
- A full search feature
- Context-sensitive help for each screen

There are two ways that you can access the online help:

- Select **Help**|**Help Contents** from the menu bar. Sections on Artix Designer and the Artix Management Console appear in the contents panel of the Eclipse help browser.
- Press **F1** for context-sensitive help.

In addition, there are a number of cheat sheets that guide you through the most important functionality in Artix Designer. To access these, select **Help**|**Cheat Sheets**.

Artix Glossary

The Artix Glossary provides quick definitions and is a comprehensive reference for Artix terms. All terms are defined in the context of the development and deployment of Web services using Artix.

Additional Resources

The IONA Knowledge Base contains helpful articles written by IONA experts about Artix and other products.

The IONA Update Center contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to IONA Online Support.

Comments, corrections, and suggestions on IONA documentation can be sent to docs-support@iona.com .

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`Fixed width`	Fixed width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `IT_Bus::AnyType` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and introduce *new terms*.
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
`>`	The notation > represents the MS-DOS or Windows command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
`[]`	Brackets enclose optional items in format and syntax descriptions.
`{}`	Braces enclose a list from which you must choose an item in format and syntax descriptions.
`|`	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in `{}` (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select **File|Open**).

CHAPTER 1

Service Oriented Architecture

Service Oriented Architecture is a way of designing solutions around units of functionality that are implementation agnostic.

Overview

A *Service Oriented Architecture* (SOA) is a loosely-coupled, distributed architecture in which services make resources available to service consumers in a standardized way. SOA is language and protocol independent. By providing a way of describing services that is independent of implementation details, SOA makes it easier to develop and deploy systems that require large amounts of integration.

A key piece of technology used in enabling service orientation is an enterprise service bus (ESB). An ESB is the infrastructure that allows services to interact in a distributed environment. It handles the delivery of messages between different middleware systems, and provides management, monitoring, and mediation services such as routing, service discovery, or transaction processing.

In this chapter

This chapter discusses the following topics:

What is a Service Oriented Architecture?

Overview

Service Oriented Architecture (SOA) is the next logical step in the growth of software development methodology. It takes the concepts behind procedure-oriented design and object-oriented design and moves the layer of abstraction one step further away from the implementation details of a piece of atomic functionality.

It also builds on the concepts used to create distributed applications such as CORBA. Specifically, it uses an XML-based grammar for defining abstract interfaces. The interfaces define the messages passed between service and consumer using XMLSchema. By using XML-based types, service definitions make no assumptions about how the service is implemented.

Evolution of reusability in application design

The fundamental ideas behind service orientation are not new. For as long as people have been developing software one of the core concepts has been reusability of functionality. To achieve this, software languages and software design paradigms have evolved that encourage the compartmentalization of functionality. Functionality is grouped together into small, reusable units that can be used independently of the application for which they were originally intended. This not only makes them reusable, but also increases the ease with which large applications can be updated because a change to one unit of functionality does not necessarily require changes to the whole application.

The first leap forward in the quest for reusability was the move from line-by-line programming languages like BASIC to procedural languages like Pascal and C. These procedural languages brought about the procedure-oriented design paradigm. Software began being designed as collections of reusable procedures each of which performed discreet pieces of functionality.

For example, a banking application may have a procedure that handled deposits and a different procedure that handled withdrawals as shown in Figure 1. Using this paradigm you could reuse each of the procedures if

needed. It also made it easier to modify one procedure with out needing to modify the others. So if your bank needed to add a step to the withdrawal process, only one part of the code would require updating.

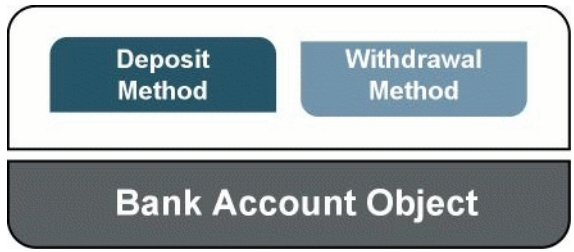

Figure 1: *Procedure Oriented Bank Account Application*

The next leap forward was the arrival of object-oriented programming languages like C++ and Java. Object-oriented languages, and object-oriented design, made reusability easier by introducing the concept of an object as an atomic unit of functionality. In this paradigm, an object exposes a well-defined interface that can be called on by other objects that need the object's functionality. Because an object is a self-contained entity and because its interface is well-defined it is highly reusable across many applications.

In an object-oriented design, the deposit and withdrawal procedures may be aggregated into an account object that has all of the methods needed for managing a bank account. While this may seem like a step backwards, it actually makes sense from a reuse standpoint. The individual methods of the account object will still adhere to the concepts of procedure oriented design and remain fairly independent. However, you now have a reusable

component that represents a business asset. The account object can be used in multiple applications, as shown in Figure 2, and the changes made to any of its methods will be shared by all the applications that use it.

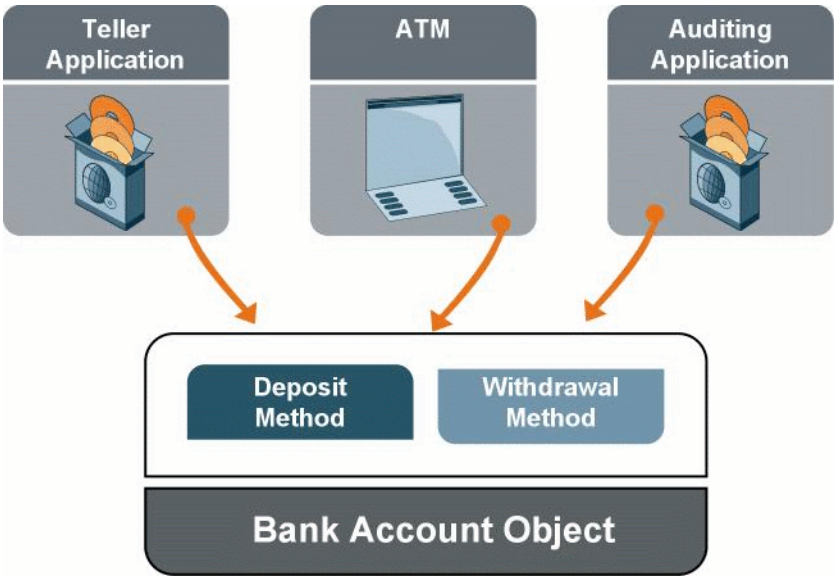

Figure 2: *Object Oriented Bank Account*

The problems of distributed application development

As the code used to write applications became more modular and reusable, applications were being broken up into pieces that were distributed across many machines. For example, an application that allows bank tellers to

make withdrawals and deposits is broken into a client and a server portion as shown in Figure 3. The server portion may also be broken up into several separate parts.

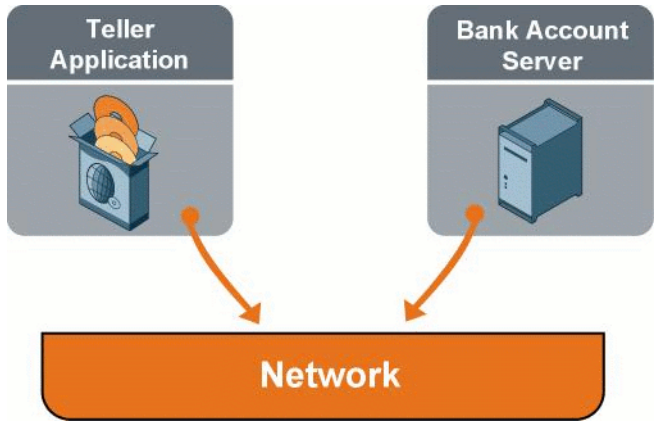

Figure 3: *Distributed Bank Account Application*

Breaking applications into multiple parts and distributing them across multiple platforms presented a new set of reusability problems. Early distributed applications were designed so that all of the parts were tightly coupled. The messages used to communicate between them were passed using proprietary formats. Often there were dependencies on specific networking hardware and protocols. One result of this tight coupling is that pieces of functionality can not be reused because it is difficult to integrate these islands of functionality. For example, if a bank had two systems that needed to do credit checks, each system would need to implement that functionality because they used different messaging styles or different networking technologies.

Many attempts have been made to solve the reusability and integration problems posed by distributed application development. Some solutions include CORBA, DCOM, MOMs, and large EAI servers. Each of these solutions got parts of the problem right, but never solved the entire problem. CORBA and many EAI solutions increased interoperabilty and reusability by providing abstract, implementation neutral definitions of atomic units of

functionality that could be used as a contract between parts of a distributed application. MOMs increased interoperabilty by defining the interaction between parts of a distributed system by the messages that are exchanged.

None of the solutions really solved the problem because they, like object-oriented programming languages, did not provide a way of breaking the dependencies that bound all the parts together. CORBA required that all of the distributed objects be CORBA objects. EAI servers required resource heavy central hubs and proprietary networking solutions. MOMs required that all of the parts used a particular messaging infrastructure that often required specific APIs to be used.

How SOA breaks the dependency chain

SOA breaks the chains of dependency by borrowing from the best ideas of all other paradigms. From object-oriented programming, SOA borrows the idea of atomic units of functionality with a well defined interface. From CORBA and EAI solutions, SOA borrows the idea of an implementation neutral interface definition language. From MOM, SOA borrows the idea of defining applications by they messages they exchange. The result is the concept of a *service*.

A service is an atomic unit of functionality defined by a set of message exchanges that are expressed using an implementation neutral grammar. A service, unlike an object, is an abstract entity whose implementation details are left largely ambiguous. The only implementation details of the service that are spelled out are the messages it exchanges. This ambiguity, coupled with the requirement that the messages be defined by an implementation neutral grammar make a service highly reusable and easy to integrate into a complex system.

Using services, you can define applications based on business requirements and not worry so much about the details of how the functionality is implemented. This is SOA. For example, you may need a unified application to generate customer billing for a telecommunications company that provides VoIP, cellular, and traditional phone services to its customers. The biggest stumbling block to this is that each department has implemented their billing system using a different technology as shown in Figure 4. Because none of the technologies were designed to be interoperable and none of them expose a common interface, building a unified billing client is

a major integration headache. It can be solved using traditional means, but the solution involves either adding an expensive EAI product in the middle or developing a custom integration layer.

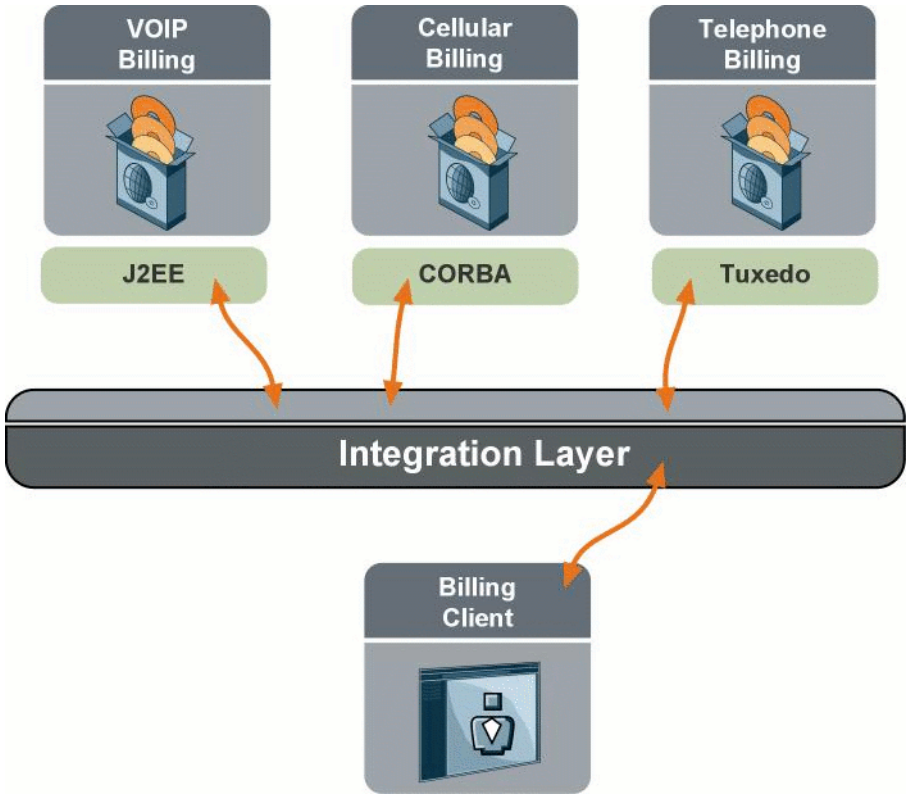

Figure 4: *Separate Billing Systems*

However, you can define a service that represents the functionality of all three billing systems as shown in Figure 5. This service only requires one message exchange: the user sends the customer's account number and the service returns the bill. You now have a common interface through which a unified billing client can access all three systems. This makes developing the client much simpler, will not require as much maintenance, and will make it easier to migrate the billing systems to newer platforms if there is a

business need. This approach is also much easier for a business level person to understand and express, thereby making it easier for an IT department to understand the requirements.

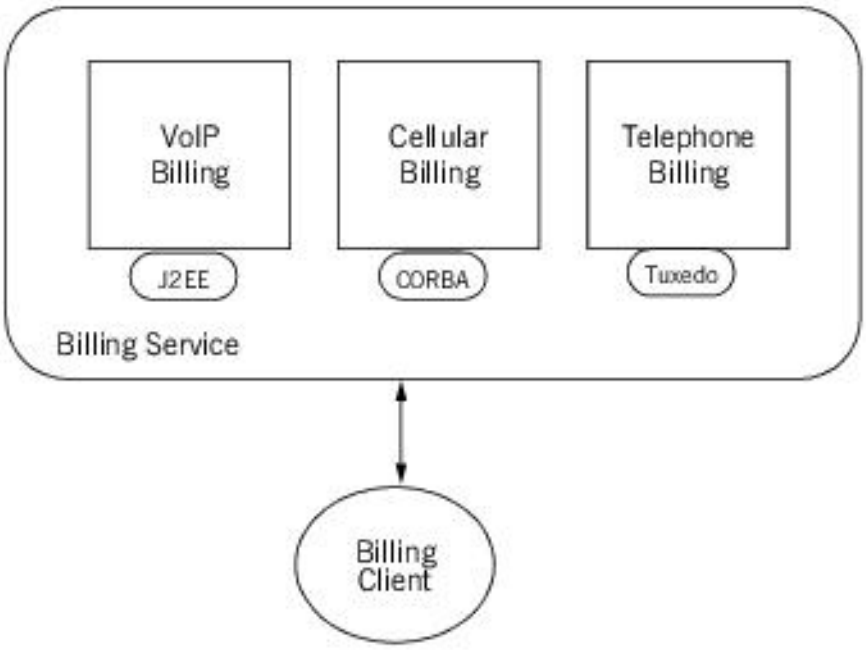

Figure 5: *Billing Systems in SOA*

Bringing a service into reality

The disconnect between SOA and real systems is that a services is just an abstraction. It is only an idealized representation of an implemented set of functionality and that implementation is still bound to the dependencies of hardware, languages, and networking protocols. Several key technologies have emerged to bridge the gap between a service and the implemented functionality that it represents. Among these are XML and HTTP.

XML is the language that allows SOA to exist. It provides the grammar used to describe services, it provides the type system used to describe the data passed by services, and it provides the most common format used to package the messages used by services.

Web Service Definition Language (WSDL) is an XML grammar standardized by W3C to describe services. Using WSDL you define all of the abstract portions of a service including the elements that make up the messages exchanged by the service. You then map the abstract messages exchanged by the service to a concrete payload format that is used on a network. You also define a physical endpoint by which the service can be accessed.

XMLSchema is the default type system for defining the messages used by a service. Because XMLSchema is a standardized XML grammar it is platform neutral and does not make any assumptions about how the messages are going to be processed. It also allows for the creation of complex messages that are built up from reusable pieces.

Simple Object Access Protocol (SOAP) is an XML-based message protocol standardized by the W3C. It defines an XML envelope for wrapping messages and a data model for encoding information in an XML document. SOAP is the most common, but not the only, concrete message format used by services. Because it is XML based, SOAP is platform independent. In addition, it is widely used.

Hypertext Transfer Protocol (HTTP) is the most common network protocol used in SOA. This is largely due to the fact that it is nearly ubiquitous. HTTP is the protocol used to connect the World Wide Web and is based on an entirely open set of standards. Its ubiquity and openness make it a perfect backbone for connecting distributed services.

What is an Enterprise Service Bus?

Overview

An *enterprise service bus* (ESB) is the layer of technology that makes SOA possible. It enables the abstraction by translating the messages defining the services into data that can be manipulated by a physical process implementing a service. An ESB also provides some QoS to the services and provides a messaging layer for services to use. Essentially, an ESB is the yarn that weaves a SOA together.

From service to endpoint

An ESB takes the concrete details defined in the `service` element of a WSDL contract and uses it to create an accessible endpoint for the service. This information includes details on how the abstract messages are mapped into data structured that can be manipulated and transmitted by the service's implementation. It also includes information about the how the service's implementation is to be exposed to the physical world. The `endpoint` is the physical representation of the abstract service defined in a WSDL contract.

As shown in Figure 6, the ESB sits between the service's implementation and any consumers that want to access the service. The ESB handles functions such as:

- publishing the endpoint's WSDL contract.
- translating the received messages into data the service's implementation can use.
- assuring that consumers have the required credentials to make requests on the service.
- directing the request to the appropriate implementation of the service.

- returning the response to the consumer.

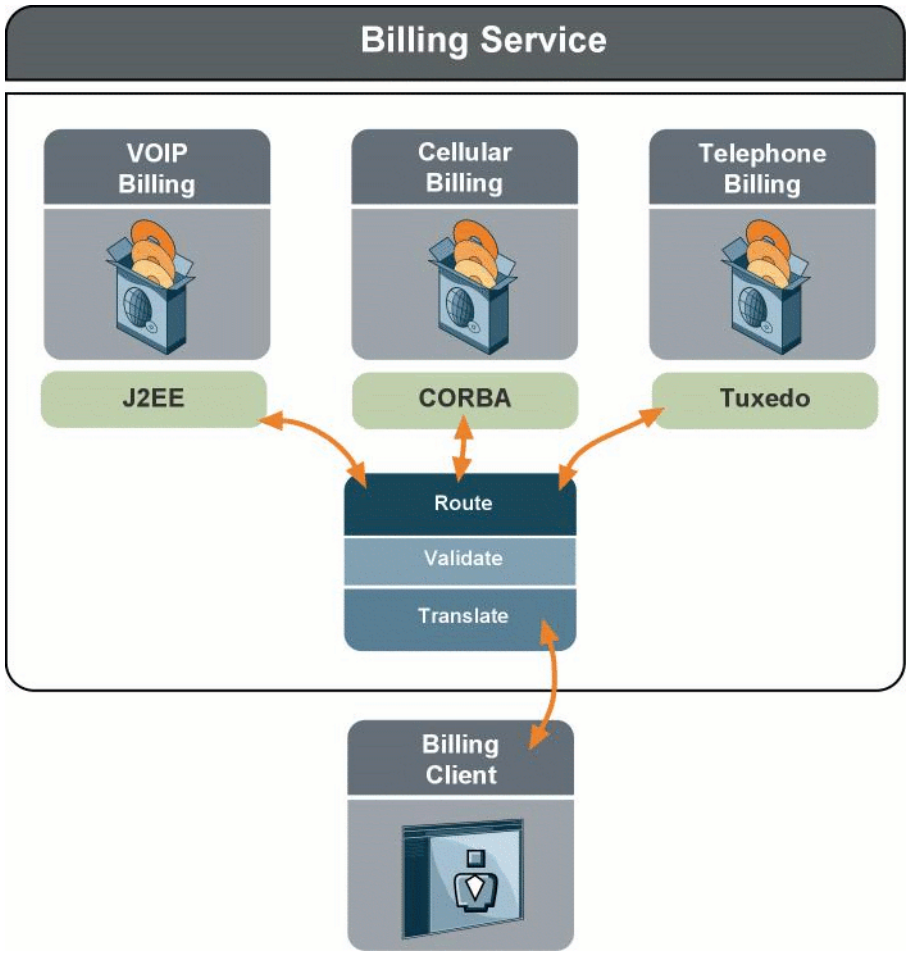

Figure 6: *Billing System SOA with an ESB*

Not EAI

A brief description of an ESB may trigger nightmares about EAIs. While the concern is warranted, ESBs have several key differentiators from the integration layers of the past:

- ESBs use industry standard WSDL contracts to define the endpoints they connect.
- ESBs use XML as a native type system.
- ESBs are distributed in nature.
- ESBs do not require the use of proprietary infrastructure.
- ESBs do not require the use of proprietary adapters.
- ESBs implement QoS based on industry standard interfaces.

The use of standardized WSDL for the interface definition language and the use of XML as a native type system make an ESB future safe and flexible. As discussed in the previous section, both are platform and implementation neutral which avoids vendor lock-in.

Strength in pieces

The most significant differentiator between ESBs and legacy EAI systems is an ESB's distributed nature. EAI systems were designed as a hub-and-spoke system. ESBs, on the other hand, are designed to be a distributed as the components they are integrating. As shown in Figure 7, an ESB distributes the load of data translation, routing, and other QoS tasks to the endpoints themselves. Because the endpoints are only responsible for translating messages that are directed to them, they can be more efficient. It also means that they can adapt to new connectivity requirements without

effecting other endpoints. The fact that routing, security, and other QoS are also distributed means that you can choose not to deploy them if they are not needed.

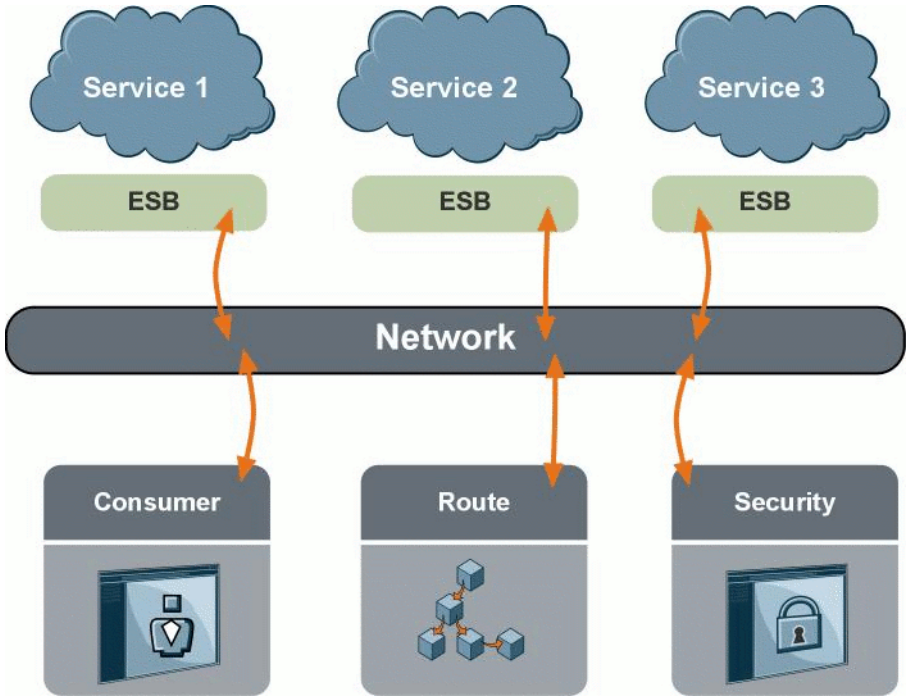

Figure 7: *Distributed Nature of an ESB*

The distributed nature of an ESB also means that you are not forced to drop all of your existing infrastructure in one big bang. You can start with a very targeted project such as service enabling a single system so that it can interact with a new AJAX based interface. As you become more comfortable with the technology, or as requirements demand, you can plug more services into the ESB without disrupting the services already deployed. As you do so, you may not even need to change any of your existing implementations because the ESB's translation services allow you to plug legacy implementations into the ESB.

The WS standards

ESBs offer a number of QoS such as transactionality, security, routing, and reliable messaging. To ensure maximum interoperabilty between implementations, ESBs base much of their QoS on a set of standards that include:

- WS-Addressing
- WS-Atomic Transactions
- WS-Coordination
- WS-Security
- WS-Reliable Messaging

These standards are all maintained by the W3C and provide a common framework on which ESBs build QoS. They were all designed around the idea that information would be passed using SOAP/HTTP. They were also designed so that services should be easily shared and accessed over the Web. Therefore they, and ESBs that implement them are built to be maximally interoperable.

How Does Artix Fit into a SOA Strategy?

Overview

Artix is IONA's ESB implementation. As such, it provides a highly distributed and easily extensible solution for implementing a SOA. Built on IONA's patented ART core, Artix comes with a number of plug-ins that support a wide range of enterprise messaging platforms and provides several enterprise QoS such as transactions and security.

How Artix is different

Many ESB solutions are merely souped up versions of the same technology that an ESB was intended to supplant. While they look like an ESB from the outside, they are really just an old fashion messaging system with some adapters thrown into the mix. They are not truly distributed and they do not help you avoid vendor lock-in. In order to use most of the features offered by these ESBs, you must make the particular ESB the backbone of your entire enterprise. Much of the QoS, routing, and translation logic is bundled into the messaging system. In essence they sell you a bunch of adapters that let you connect your systems into their plumbing.

Artix, on the other hand, gives you a bunch of adapters that lets you connect your applications into any plumbing. It does this by offering a truly distributed ESB solution. Instead of relying on a particular messaging system to provide the QoS, routing, and translation functionality, Artix moves all of the functionality to discreet endpoints. If an application needs to connect with a system that uses WebSphere MQ, the application can load the required connector and talk directly to the WebSphere MQ system. If a legacy system needs to be exposed as a Web service, you can place an endpoint on the system that can route and translate messages for it.

Extensibility

Because Artix is an extensible ESB, it has several distinct advantages:

- You can deploy SOA as it makes business sense because you can add endpoints without effecting your entire organization.
- You can chose deploy only the features you need.
- If you need to add features to an endpoint, you can do so without touching all of the deployed endpoints.
- It can easily adapt to changes in the messaging infrastructure used in an enterprise.

- Because Artix is built using a IONA's patented ART architecture, it is easy to write plug-ins that extend Artix's capabilities.

Middleware support

Artix integrates with the following middleware platforms:

- CORBA
- WebSphere MQ
- Tibco/Rendezvous
- Tuxedo
- Web services
- J2EE
- .Net

QoS

Artix provides the following qualities of service:

- Reliable messaging based on WS-ReliableMessaging
- Security including support for WS-Security headers
- Transactions based on WS-AtomicTransactions
- High availability
- Load balancing
- Location services
- Leasing

CHAPTER 2

Artix's High-Level Architecture

The Artix runtime, called a bus, connects applications to a networking infrastructure through a combination of pluggable layers.

In this chapter

This chapter discusses the following topics:

Artix as a Deployed ESB

Overview

Because Artix is an enterprise service bus it is easy to imagine Artix as a piece of plumbing that passes messages around your enterprise like a USB cable. While some ESBs are implemented in a way that makes them resemble a USB cable, Artix is more like set of caps that turn any existing networking or messaging system into a virtual bus. As shown in Figure 8, Artix lives in the endpoints that you want to connect to your system. It uses the network to do the message delivery and shields the endpoints from the details.

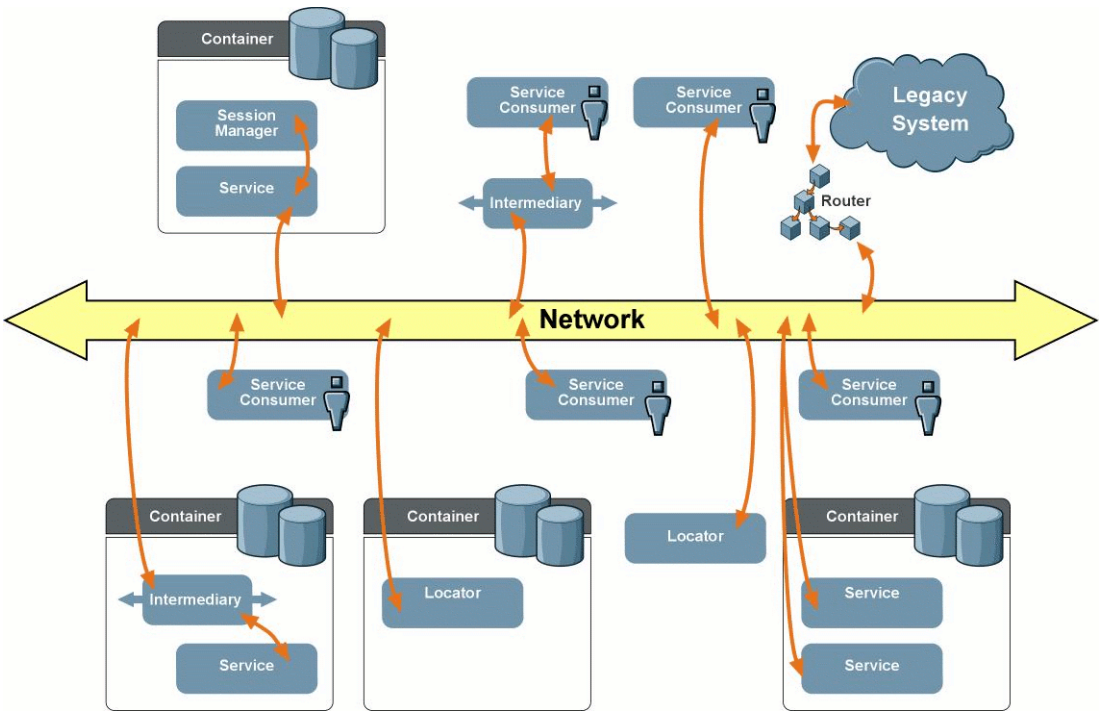

Figure 8: *Artix and the Virtual Bus*

Artix ensures that the addressing information and formats are compatible with the network infrastructure onto which the messages are placed. The network then ensures that the messages are delivered to the proper endpoints. Because Artix uses the existing network infrastructure to deliver messages, Artix can capitalize on any QoS offered by the network. For example, Artix can use the reliable messaging mechanisms offered by a JMS queue to ensure that messages are delivered.

Endpoints

Artix can be used to implement three types of endpoints in a SOA:

- *Services* are endpoints that implement the operations defined in a service contract. They are similar to servers.
- *Service Consumers*, or consumers, are endpoints that make requests on services. They are similar to clients.
- *Intermediaries* are endpoints that processes messages in a value-added way, such as converting them from one data format to another, or routing them to another service. An intermediary has characteristics of both a service and a consumer.

The Artix bus

Artix does have a bus, but it is internal. The Artix bus coordinates the passage of data from user implemented business logic to the networking system. Internally, Artix consists of the bus and a number of objects that take the data that the business logic can work with and transforms it into a message that can be sent on the network. There are also a number of objects that Artix uses to provide other features such as security and session management.

The bus is capable of coordinating and managing the messages for multiple services or service consumers. It is also responsible for loading and unloading the plug-in used by Artix. The details of how the bus coordinates messages for each type of endpoint and what components are loaded are discussed in the remaining sections of this chapter.

Artix in a Service Endpoint

Overview

A *service endpoint* is an endpoint that implements the business logic defined in a WSDL document. Using skeleton code produced by running a WSDL document through the Artix code generators, you can create a service endpoint that uses the Artix bus to connect to the network. The bus can load any plug-ins needed to provide the features you desire.

For example, you could build a service endpoint to process on-line payments from your customers. The WSDL document may specify a service that has two operations: `veiw_recent` and `make_payment`. Each operation takes the customer's account number and some additional information. Because it is going to be accessed over the Web, the WSDL document specifies that it uses SOAP/HTTPS. Using this information, Artix will generate skeleton code for the service implementation and load the proper plug-ins when the service endpoint is deployed.

What makes up a service endpoint

As shown in Figure 9, a service endpoint built with Artix has the following pieces:

- a service implementation
- a binding plug-in
- a transport plug-in

Request messages are received by the transport plug-in. Once the message is received, the bus passes the message along the messaging chain to the binding plug-in. The binding plug-in unmarshalls the data into objects that that are passed on to the service implementation. The service implementation processes the message according to the business logic it implements. If a response is generated the bus passes it back down the messaging chain so the transport can place it back onto the network.

In addition, a service endpoint can have any number of request-level and message-level interceptors that provide added functionality to the endpoint. These interceptors, which are independent of the WSDL document defining the endpoint, have access to requests before the service implementation.

They also have access to the response after the service implementation generates it. They can be used to perform functions such as encryption, validation, or header processing.

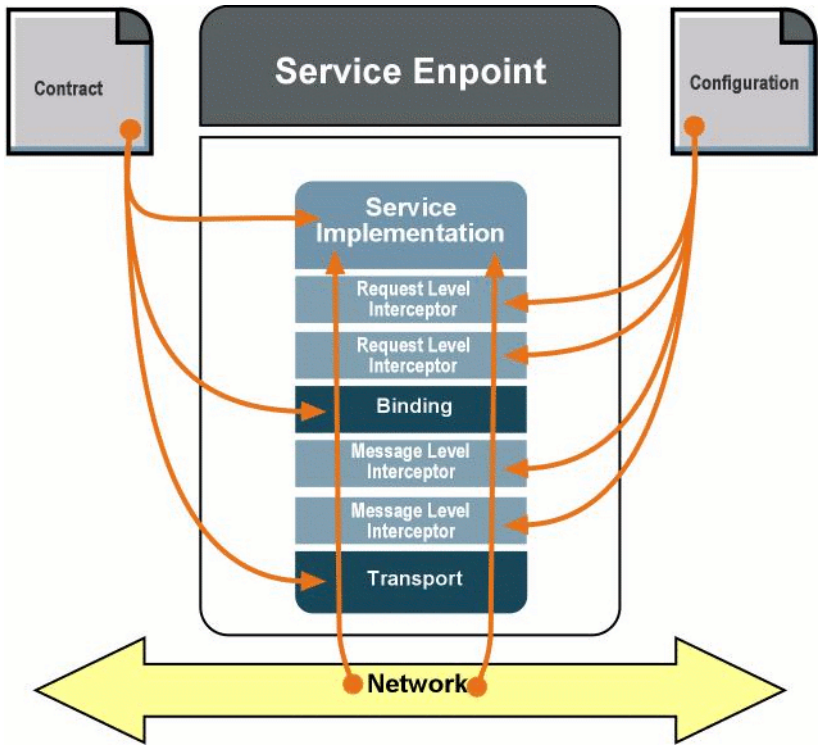

Figure 9: *High-level View of a Service Endpoint*

Service implementation

The service implementation is in Artix can be created using either C++ or Java and is based on code generated from the logical portion of the service endpoint's WSDL document. The bus loads the object that contains the logic for the service and creates a servant that wraps the implementation so that it can be managed by the bus. Depending on the threading model specified, the bus will create as many instances of the implementation object needed to process all of the requests made against the service.

The implementation does not have direct access to the request messages. It receives messages from the bus as parameters to the operations specified in the WSDL document from which it was generated. Similarly, it returns any responses to the bus as a return value. The marshalling of the data is handled by the binding plug-in. For example, the `make_payment()` method in your on-line billing endpoint would receive a string containing the account number and a float containing the amount of the payment. It would return a boolean value depending on the success or failure of the action. It has no knowledge of how the messages are packaged.

Exceptions thrown in the implementation object are also passed to the messaging chain by the bus. The lower layers of the messaging chain will handle the exception as a fault message. How the exception is returned to the consumer depends on how the service is defined in the WSDL document. For example, services that use CORBA will use the CORBA exception mechanism for reporting remote exceptions and services the use SOAP/HTTP will respond with a SOAP fault containing information about the exception.

Request-level interceptors

Request-level interceptors sit between the binding and the service implementation. They have access to the message data when it is in between the bits received off of the wire and the objects manipulated by the service implementation, so they can access the header values of the message. For example, the WS-Security specification requires that a SOAP header holding the security token be included with all requests. A request-level handler could remove this header and authorize the consumer before the request is passed to the implementation.

Request-level interceptors can also inspect and change the parameters of the operation that fulfils the request. So, if the payment is amount being passed to `make_payment()` is specified in Euros and the service endpoint process values in US dollars, a request-level handler can do the conversion before the data is passed to the implementation. Return value scan also be inspected and changed so the transaction information returned by `view_recent()` could be converted into the desired currency.

Request-level interceptors are developed as plug-ins and are loaded based on information in the Artix configuration file. They are executed in the sequential order specified in the configuration file. For instance, if the configuration file specifies that the request level interceptors are called in

the order `a b c` that is the order they will be called when a request is received. When a response is sent down the chain, the interceptors will be called in the order `c b a`.

Exceptions thrown in request-level handlers cause the message to be immediately dispatched to the binding. They are labeled as fault messages. Requests will not be passed onto the service implementation.

Binding

The binding is responsible for converting messages between the binary types used by the service implementation and the data format used on the wire. The mapping is determined by the WSDL `binding` element. The bus will load the appropriate binding plug-ins based on the `binding` elements in the contract defining the endpoint. For example, the on-line billing service endpoint would load the SOAP binding plug-in.

Because the binding plug-in is not loaded by the bus until the service endpoint is deployed, you can change the payload format used by the service endpoint without changing the service implementation. For example, if you wanted to expose the service endpoint to a COBOL application you could edit the WSDL document to include a fixed record length binding and redeploy the endpoint. The bus will then load the binding plug-in used to process fixed record length data.

Exceptions thrown in the binding are sent back down the messaging chain as a fault message. Requests will not be passed to the request-level interceptors.

Message-level interceptors

Message-level interceptors sit between the binding and the transport. When a request comes in, interceptors have access to the binary stream holding the message pulled off the wire. At this point, they can perform actions such as decompression or decryption. When a response is being returned, interceptors have access to the binary stream holding the newly packaged message. At this point they can perform actions such as compression or encryption.

Like request-level interceptors, message level interceptors are developed as plug-ins and are deployed based on information in the Artix configuration file. They are also called in the order specified in the configuration.

Exceptions in message-level handlers result in unpredictable behavior. It is recommended that your code does not throw exceptions at this level.

Transport

The transport is responsible for pulling requests off of the network and placing responses back on the network. The transport plug-ins to be loaded and their configuration are determined by the WSDL `port` elements included in the contract defining the endpoint. For example, the on-line billing service endpoint would load the HTTPS plug-in.

Because the transport plug-in is not loaded until the service endpoint is deployed, you can change the transport used by the service endpoint without making any change to the service implementation. For example, if you decided that the on-line billing service needed to be accessible to systems that used CORBA or Tibco Rendezvous, you could simply edit the service endpoint's contract and redeploy it. The bus will then load the plug-ins needed for the new connections.

Artix in a Consumer Endpoint

Overview

A *consumer* endpoint is an endpoint that invokes on a service endpoint to make use of the business logic implemented by the service endpoint. Using stub code produced by running a WSDL document through the Artix code generators, you can create a consumer endpoint that uses the Artix bus to load a service proxy for the service defined in the WSDL and connect to one of the endpoints that implements that service. The bus can also load any plug-ins needed to provide the features you desire.

For example, you could build a consumer endpoint to access an on-line payment service. The WSDL document defining the payment service may specify two operations: `veiw_recent` and `make_payment`. Each operation takes the customer's account number and some additional information. The WSDL document specifies that the service uses SOAP/HTTPS for communicating with consumers. Using this information, Artix will generate stub code for the service and load the proper plug-ins when the consumer endpoint is deployed.

What makes up a consumer endpoint

As shown in Figure 10, a consumer endpoint using Artix has the following pieces:

- the consumer implementation
- a service proxy
- a binding plug-in
- a transport plug-in

Requests are generated by the service proxy when it is invoked by the consumer implementation. The request is then passed to the binding plug-in where it is marshalled into the data format specified in the service's WSDL document. From the binding, the request is passed to the transport plug-in where it placed onto the wire. If a response is expected, the transport plug-in waits until the response arrives. When the response arrives, the transport passes it back up the messaging chain to the binding where it is unmarshalled. The binding passes the unmarshalled data to the service proxy and the service proxy passes it back to the consumer implementaiton.

In addition, a consumer endpoint can have any number of request-level and message-level interceptors that provide added functionality to the endpoint. These interceptors, which are independent of the a WSDL document, have access to requests after the service proxy. They also have access to the response before the service proxy. They can be used to perform functions such as encryption, validation, or header processing.

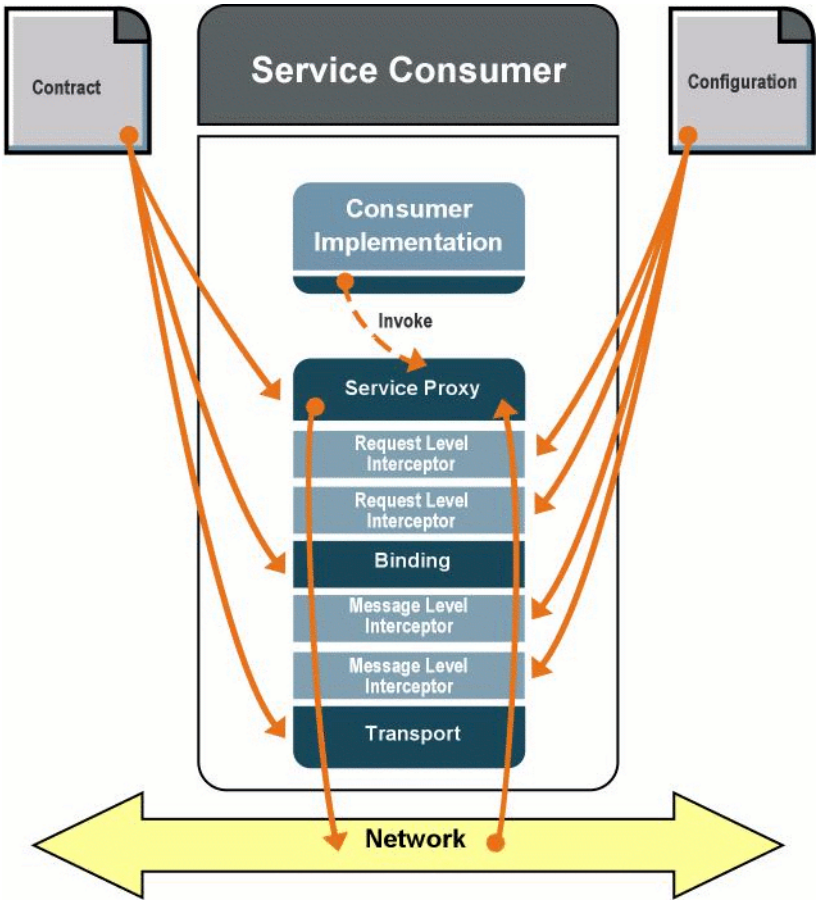

Figure 10: *High-level View of a Consumer Endpoint*

Consumer implementation

The consumer implementation is user-written code that provides the business logic for the consumer. As part of the consumer implementation you need to instantiate and register service proxies for any service endpoint upon which the consumer will make requests. For example, the on-line payment consumer will need to instantiate and register a proxy for the on-line payment service endpoint upon which it will ultimately make requests.

Service proxy

The service proxy is a stub generated from logical portion of a WSDL document defining the service upon which the consumer endpoint will make requests. It allows a consumer endpoint to invoke the operations offered by the service. The service proxy offers all of the operations defined in the service's WSDL document.

When instantiated, a service proxy provides a connection to the a service endpoint that implements the service defined in the WSDL document from which it was generated. As part of their instantiation, service proxies are registered with the Artix bus so that the invocations made on the service proxy can be properly passed along the message chain and delivered to the proper service endpoint.

Request-level interceptors

Request-level interceptors sit between the service proxy and the binding. They have access to the parameters of the invoked operation. They can inspect the parameters and take action based on their values. They can also alter the value of any of the parameters. For example, when `make_payment()` is invoked a request-level interceptor could be used to check the user's bank account balance to ensure they have the funds to make the payment specified. If there are not enough funds, the interceptor could also change the amount of the payment to a value that the user can afford.

While they can change the values of the operation's parameters, request-level handlers cannot add or remove parameters to the operation. For example, you could not use a request-level interceptor to split a single parameter that contains the user's full name into two parameters: one for the first name and one for the last name.

Request-level handlers also have access to the message headers that are included with the message. When requests are made, they can add a SOAP header to the message. For example, you could write a request-level handler

to add a WS-Security header to all out-going requests. When a response is received, request-level handlers can inspect the message headers before the message is passed back into the consumer implementation. For example, a request-level handler could check a message header to validate the data returned in response to `view_recent()`.

Request-level interceptors are developed as plug-ins and are loaded based on information in the Artix configuration file. They are executed in the sequential order specified in the configuration file. For instance, if the configuration file specifies that the request-level interceptors are called in the order `a b c`, that is the order they will be called when a request is passed down the message chain. When a response comes up the chain, the interceptors will be called in the order `c b a`.

Exceptions generated in a request-level interceptor are immediately returned to the consumer implementation. If the exception is thrown while processing a request, the request is not sent. The client implementation is responsible for properly handling the exception.

Binding

The binding is responsible for converting messages between the binary types used by the client implementation and the data format used on the wire. The mapping is determined by the WSDL `binding` element. The Artix bus will load the appropriate binding plug-ins based on the binding elements in the contract defining the service to which the client is making requests. For example, the bus would load the SOAP binding plug-in for the on-line payment consumer.

Because the binding plug-in is not loaded by the bus until the consumer endpoint is deployed, you can change the payload format used by the endpoint without changing any of the endpoint's code. For example, if your on-line billing service endpoint is an application that uses Tuxedo's FML buffers you could edit the WSDL document to include an FML binding and redeploy the endpoint. The bus will then load the binding plug-in used to process FML.

Exceptions in the binding are sent back up the messaging chain as a fault message. Requests will not be passed to the message-level interceptors.

Message-level interceptors

Message-level interceptors sit between the binding and the transport. When a request is made, they have access to the binary data stream that contains the newly packaged message before it is placed onto the wire. At this point they can perform actions such as compression or encryption of the outgoing

request. When a response is received, the interceptors have access to the binary stream that represents the message pulled off of the wire. At this point, they can perform operations such as decompress the data or decrypt it.

Like request-level interceptors, message level-interceptors are developed as plug-ins and are deployed based on information in the Artix configuration file. They are also called in the order specified in the configuration.

Message level interceptors return exceptions directly to the consumer implementation. If the exception is thrown wile processing a request, the request is not sent. If the exception is thrown when processing a response, the message is not passed to the rest of the messaging chain.

Transport

The transport is responsible for placing requests on the network and pulling responses back off of the network. The transport plug-ins to be loaded and their configuration are determined by the WSDL `port` elements in the WSDL document that defines the service endpoint on which the consumer endpoint is invoking. For example, the bus would load the HTTP transport plug-in for the on-line billing consumer endpoint.

Because the transport plug-in is not loaded until the consumer endpoint is deployed, you can change the transport by simply editing the WSDL document used to define the endpoint. For example, if you decided that the on-line billing service endpoints were to moved to WebSphere MQ, you could simply edit the consumer endpoint's WSDL document and redeploy it. The bus will then load the plug-ins needed to connect to WebSphere MQ.

Artix in an Intermediary

Overview

An *intermediary* is a special case of a service endpoint. It is a service endpoint whose primary function is intercept messages, perform some value-added processing, and possibly pass the message on to its intended destination. Intermediaries have some of the characteristics of a service endpoint and also of a consumer endpoint. They are typically defined by a WSDL document defining all of the interfaces required by the intermediary and that has been extended to contain the rules for how the intermediary is to process messages. Using the extended WSDL document, you can generate skeleton code and stub code for the endpoints with which the intermediary will interact. Alternatively, intermediaries can use generic interfaces that are created at runtime based on the information provided in the contract. The bus will use the information in the contract to load the plug-ins needed to connect the intermediary to the network.

For example, you could build an intermediary that collected statistics about how long it took a service endpoint to process requests, the average payment amount, how many times a particular operation was invoked, or how many requests are processed by all of the service endpoints on your SOA. Artix uses an intermediary to service-enable legacy systems by performing transport and binding switching. Other uses of intermediaries are message routing and message transformation. For more information about the intermediaries provided with Artix see "The Artix Router" on page 38 and "The Artix Transformer" on page 54.

What makes up an intermediary

As shown in Figure 11, an intermediary built using Artix has the following pieces:

- a service-side transport plug-in
- a service-side binding plug-in
- a service implementation
- a service proxy
- a consumer-side binding plug-in
- a consumer-side transport plug-in

Requests are picked up from the network by the service-side transport plug-in. The bus passes the request up the service-side message chain to the service implementation. The service implementation performs any message processing that is required. The service implementation invokes a service proxy if it is appropriate. The request is then passed through the consumer-side messaging chain to the network. When the response arrives, the consumer-side transport plug-in passes it back up the consumer side messaging chain to the service implementation. The service implementation performs any message processing that is required and then passes the response to the bus. The bus passes the response down the service-side messaging chain to the network.

In addition, an intermediary can have any number of request-level and message-level interceptors that provide added functionality to the endpoint. These interceptors can be used to perform functions such as encryption, validation, or header processing.

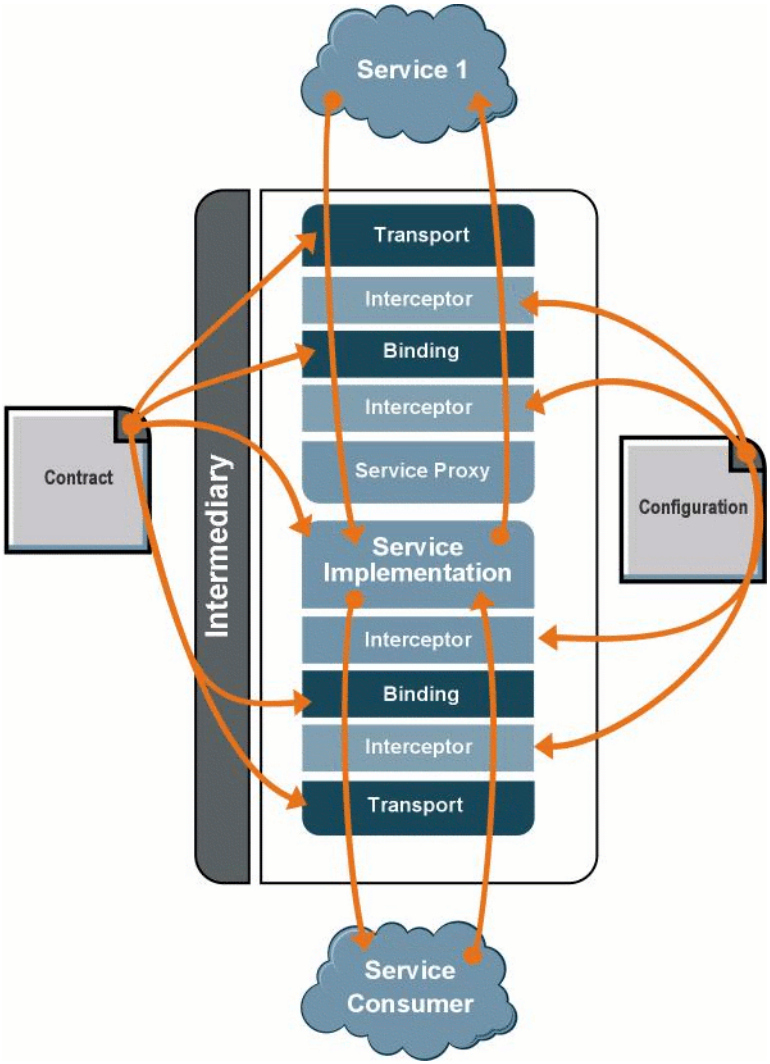

Figure 11: *High-level View of an Intermediary*

Service-side messaging chain

An intermediary's service-side messaging chain functions identically to the messaging chain of a service endpoint. It is made up of a transport, message-level handlers, a binding, and request-level handlers. The binding and transport are specified by the part of the intermediary's contract that defines the service(s) that the intermediary can interact with. The handlers in the chain are specified in the intermediary's configuration.

For more information see "Artix in a Service Endpoint" on page 20.

Service implementation

An intermediary's service implementation determines the functionality of the intermediary. For example, it may inspect the account number of a payee and use it to route the request to a regional payment center.

The only requirement for an intermediary's service implementation is that it continues the invocation chain for the messages it receives. For example, if the intermediary is placed in front of a teller service, the intermediary must pass along all incoming requests to an instance of the teller service for which the request was intended.

Service proxies

An intermediary has a service proxy for any service to which it must pass messages. In some cases this may be a single service, but an intermediary can also pass messages along to a number of services. For example, the Artix router can redirect a message to any number of services.

Consumer-side messaging chain

An intermediary's consumer-side messaging chain functions identically to the messaging chain of a consumer endpoint. It is made up of request-level handlers, a binding, message-level handlers, and a transport. The binding and transport are specified by the part of the intermediary's contract that defines the service(s) that the intermediary can interact with. The handlers in the chain are specified in the intermediary's configuration.

For more information see "Artix in a Consumer Endpoint" on page 25.

CHAPTER 3

Services Provided with Artix

Artix provides a number of services that add value and reliability to a SOA.

In this chapter

This chapter discusses the following topics:

The Artix Container

Overview

One of the key features of SOA is that its endpoints are highly dynamic. The Artix container provides a number of features that make Artix enabled endpoints more dynamic including:

- remote deployment
- suspension of an endpoint
- automatic reloading of an endpoint
- dynamic endpoint configuration
- monitoring of endpoint performance metrics

The container does this by hosting a light-weight administrative service along side the endpoints hosted in the container as shown in Figure 12.

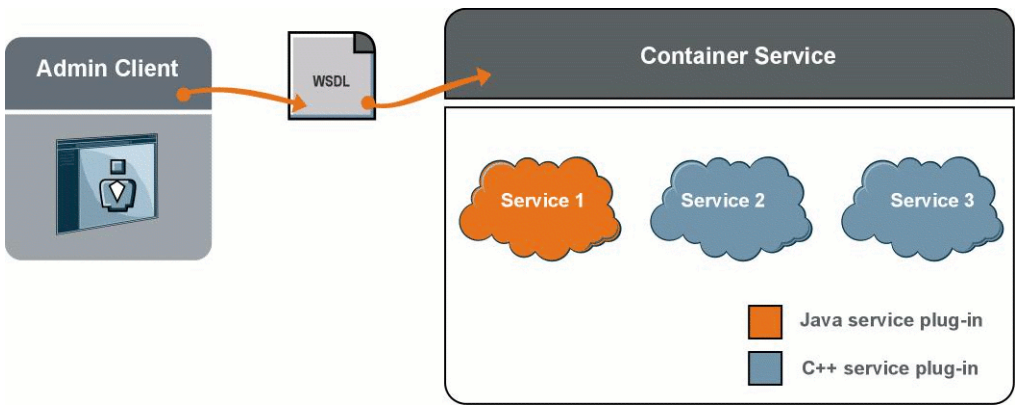

Figure 12: *Overview of the Artix Container*

Container server

The container server is a light weight process that can host a number of Artix enabled endpoints. It contains an Artix bus that instantiates service implementation objects, loads the binding and transport plug-ins specified in the WSDL documents of the endpoints it is hosting, and exposes the

endpoints to the network. The container's bus coordinates the flow of messages so that messages are placed on the proper message chains and delivered to the appropriate service implementations.

in addition to the endpoints you deploy into a container, Artix containers always load an instance of the container administrative service.

Administrative service

The container's administrative service is an Artix service that allows you to manage the endpoints deployed in a container. Like all services in SOA, the administrative service is defined by a WSDL document. By default it is exposed as an endpoint using SOAP/HTTP and can be accessed by any consumer endpoint that instantiates an administrative service proxy. You can alter the networking properties of an administrative service endpoint such that it uses any of the binding/transport combinations supported by Artix.

The administrative service provides the following operations:

- List all endpoints deployed in the container
- Stop a running endpoint
- Start a dormant endpoint
- Remove an endpoint
- Deploy a new endpoint
- Get a reference to an endpoint
- Get the WSDL for an endpoint
- Get the URL to an endpoint's WSDL document
- Retrieve performance metrics for an endpoint
- Shut down the container

The Artix Router

Overview

The Artix router is an intermediary whose primary role is to redirect messages based on rules defined in its contract. As shown in Figure 13, an Artix router has a service-side interface that receives requests from consumer endpoints. It also has one or more consumer-side service proxies that forward the request to service implementations on the backend of the router.

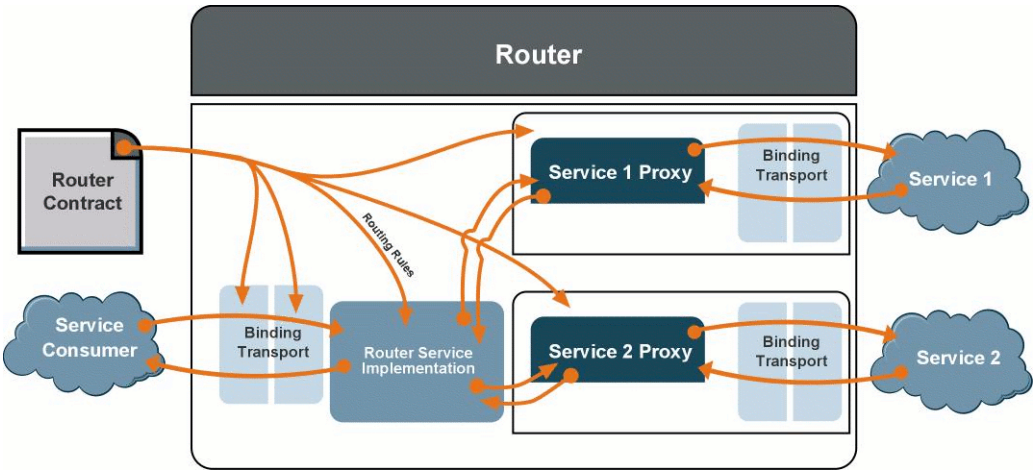

Figure 13: *Overview of the Artix Router*

The service-side messaging chain and consumer-side messaging chain are defined by separate parts of the router's WSDL document. They do not necessarily have a common binding or transport.

Service-side

The service-side of a router looks like a service endpoint to the other endpoints on your network. It is responsible for receiving requests from consumers that make requests on the service or services behind the router.

Its interface and messaging chain is determined by a service definition in the router's WSDL document. It has a transport plug-in and a binding plug-in and any optional interceptors required for QoS.

For example, a router could be used to direct messages from a .Net client that uses SOAP/HTTP to a backend service that is implemented using SOAP/JMS. The router would load the HTTP transport plug-in and the SOAP binding plug-in on the service-side messaging chain. This way the router makes the backend service look like a SOAP/HTTP endpoint.

Consumer-side

The consumer-side of a router looks like a consumer endpoint to the rest of the endpoints on your network. It consists of one or more service proxies and their associated message chains and is responsible for forwarding requests to the services on the backend of the router. The proxies, and their messaging chains, are defined in the router's WSDL document. However, they are not instantiated until they are needed by the router. So, if one of the destinations in the router's WSDL document never receives a message, no consumer-side artifacts will be created for it.

The consumer-side proxies can all have a different combination of bindings and transports in its messaging chains. They also can have a different combination from the service-side of the router. For example, if you wanted to build an AJAX based client that needed to make requests on two backend servers, you could deploy a router that presents a consolidated service facade. The router's service-side interface would look like a SOAP/HTTP service endpoint that offered all of the operations of both backend services. Its consumer-side, however, would consist of two consumer endpoints that pass the requests along to the appropriate backend server. For example, if one server is a CORBA server that offers a `buildRobosnake` operation and the other server was a Tuxedo based server that offers a `buildRobopenguin` operation, the router's consumer-side would consist of one CORBA consumer endpoint and one Tuxedo consumer endpoint.

Features

A router provides a number of features:

- message routing
- payload format translation
- transport switching
- load balancing
- message broadcasting

- endpoint fail-over

More information

For more information about the Artix container see Configuring and Deploying Artix Solutions.

Security

Overview

Artix's security architecture is designed to be easily deployable and easily connect to any existing security infrastructure already in use. As shown in Figure 16, it consists of two main components:

- the Artix security plug-in
- The Artix security server

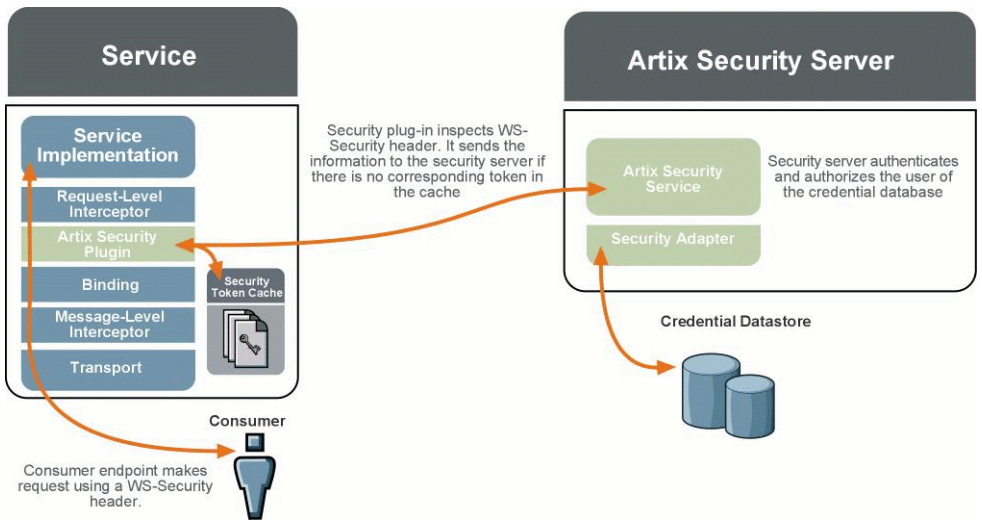

Figure 14: *Overview of the Artix Security Architecture*

The security plug-in is responsible for getting the credentials from incoming messages to a service endpoint and sending them to the security server. The security server takes the credentials performs authentication and authorization using user data stored in a credential datastore.

Security plug-in

The Artix security plug-in is deployed into the message chain of any service endpoint that uses the Artix security service. It checks incoming requests for security credentials. Before allowing the request to be forwarded to the

service implementation, it checks with the Artix security server to validate the user and ensure that they are authorized to access the service. The security plug-in uses mutually authenticated and encrypted channel to communicate with the security service.

For optimization, the security plug-in has a token cache that holds on to authorization tokens from the security server. Before sending the credentials to the security server, the plug-in will check its cache for a valid token that matches the credentials from the request. If a valid token is stored in the plug-in's cache, the plug-in will use it. If not, it will request one from the security service.

Security server

The Artix security server is a standalone server that provides the authentication and authorization functionality for Artix service endpoints. It is designed to use pluggable adapters that connect to a variety of credential datastores. For example, if you are already using LDAP on your systems, the Artix security server can leverage that data to perform it's functions.

To ensure that the Artix security server has the following enterprise features:

- high-availability through clustering
- token federation

More information

For more information about Artix security see the Artix Security Guide.

The Artix Locator

Overview

The Artix locator is a lightweight registry of deployed Artix endpoints. Artix enabled service endpoints register their endpoint information with a locator instance and consumer endpoints can use a locator instance to get references to an endpoint that implements a given service. It uses WS-Addressing compliant endpoint references to provide addressing information to consumers.

As shown in Figure 15, the locator consists of three components:

- The locator service is deployed into your network as a service endpoint.
- The locator endpoint plug-in is deployed with Artix service endpoints that want to register with a locator instance.

- The locator client plug-in is deployed with Artix consumer endpoints that want to use the locator service to get the addressing information.

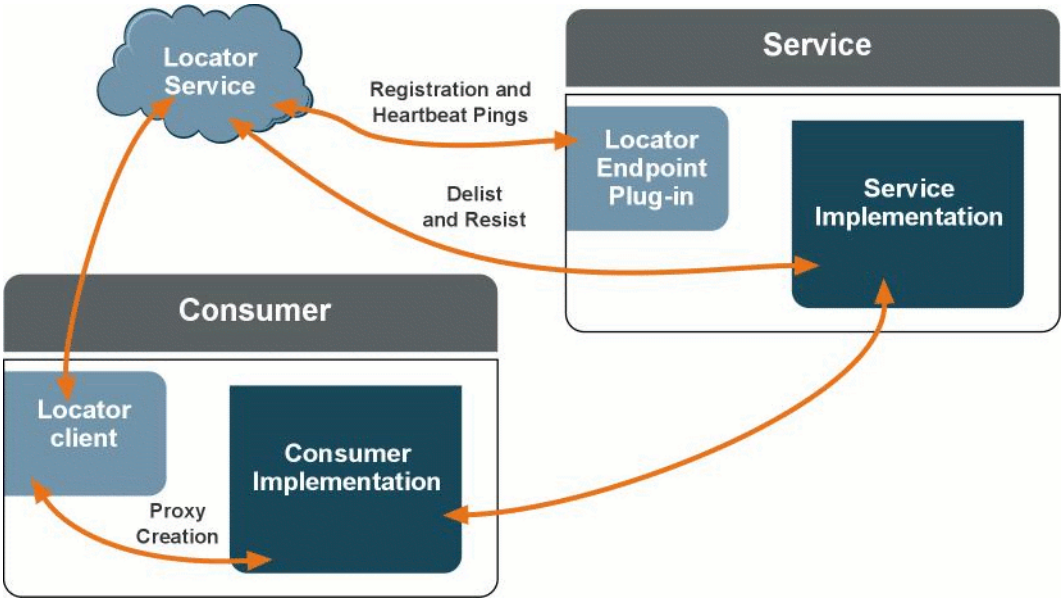

Figure 15: *Overview of the Artix Locator*

Locator service

The locator service, like all services in SOA, is defined by a WSDL document. Artix contains a service implementation using skeleton code generated from this WSDL document. You can deploy an instance of the locator service into an Artix container to create a locator service endpoint that can respond to the following types of requests:

- service registration
- service deregistration
- service endpoint look-up
- service endpoint query

The WSDL document supplied with Artix defines a locator service endpoint using SOAP/HTTP. You should not modify this because the peer manager that is used to interact with the locator cannot work with other transports.

Locator endpoint plug-in

The locator endpoint plug-in is loaded into the process space of any Artix service endpoint that intends to register with an instance of the locator. It is responsible for registering the service with a locator instance when the service endpoint starts up. It is also responsible for loading a peer manager that is responsible for monitoring the health of the locator service endpoint with which it is registered. If the associated locator service endpoint goes down, the peer manager reregisters the service endpoint when it returns. If the service endpoint goes down, the locator instance unregisters it.

Locator client plug-in

The locator client plug-in is loaded into the process space of any Artix consumer endpoint that wants to use the locator to get addressing information when creating a service proxy. When it is loaded, a consumer endpoint can automatically perform look-ups on a locator service endpoint without creating a service proxy for the locator. The plug-in has its own locator service proxy that is used by the Artix initial reference resolving mechanism. The plug-in does not, however, support service endpoint queries.

To use the locator service's service endpoint query mechanism or to access the locator service from a non-Artix consumer endpoint, you can create a service proxy for the locator service. Using the proxy, consumer endpoints can access all of the features of the locator service regardless of the platform used to implement them.

Features

The locator has the following features:

- provides references to deployed service endpoints
- load balancing among endpoints that implement the same service
- highly available

More information

For more information on the Artix locator see the Artix Locator Guide.

The Artix Session Manager

Overview

The Artix session manager is a versatile service that provides the following features:

- Limiting the amount of time a consumer endpoint can access a service endpoint
- Limiting the number of concurrent consumer connections to a service endpoint
- Stateful service endpoints

Components

As shown in Figure 16, the session manager is implemented in a modular fashion. It consists of the following components:

- the session manager service implementation
- a policy plug-in that is collocated with the service implementation
- an endpoint manager plug-in that is collocated with all managed endpoints

- a session token interceptor that sits in the messaging chain of all managed endpoints

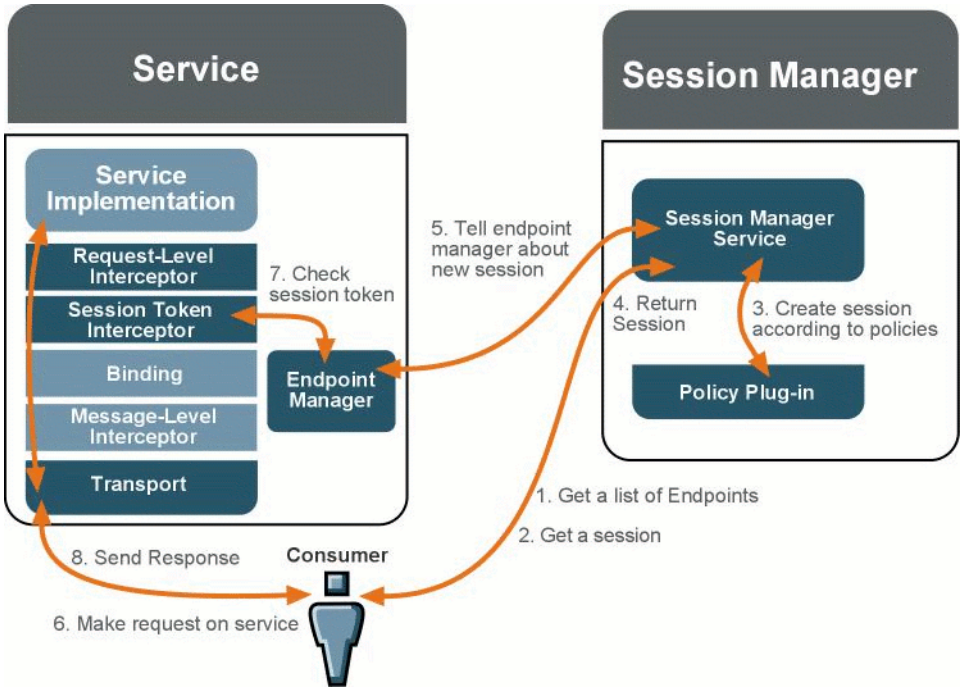

Figure 16: *Overview of the Artix Session Manager*

Session manager service

The session manager service is defined by a WSDL document and is implemented by a library shipped with Artix. You deploy instances of the session manager service implementation into an Artix container to make session manager service endpoints. These endpoints can be accessed by any consumer endpoints that can instantiate a proxy for the session manager service and communicate using SOAP/HTTP.

In general, consumers will request lists of registered service groups from the session manager. The consumer will then invoke on the session manager to request a session for one of the returned service groups. In addition, consumers can request extensions to their sessions and request that a

session be ended. The other session manager components also have specific operations that they invoke on the session manager service to provide the service-side functionality.

Policy plug-in

The session policy plug-in is deployed into the same process space as the session manager service endpoint. It is responsible for defining rules about the duration of sessions, rules about the number of concurrent sessions allowed per group, and other rules about how sessions are granted. Before the session manager grants a session to a consumer, it checks with the policy plug-in.

Artix comes with a default policy plug-in called `sm_simple_policy`. This plug-in uses information from the Artix configuration file to determine length of sessions and the maximum number of concurrent sessions allowed. If you need more detailed session rules, you can write your own policy plug-in.

Endpoint manager

The endpoint manager plug-in is loaded into the process space of any Artix enabled service endpoints that intends to register with a session manager service endpoint. The endpoint manager plug-ins are in constant communication with the session manager service endpoint to report on the endpoint's health, to receive information on new sessions that have been granted to the managed service endpoints, and to check on the health of the session manager service endpoint.

Session token interceptor

The session token interceptor is placed in a service endpoint's messaging chain when it is configured to use managed sessions. It looks for the session token that is attached to a request. If no session token is found, the interceptor rejects the request. If the session token is found, the token is sent to the endpoint manager for verification. If the session token is invalid, the interceptor rejects the request. If the session is valid, the request is passed up the message chain.

More information

For more information on the Artix session manager see the Artix Session Manager Guide.

Reliable Messaging

Overview

When being used in conjunction with a reliable transport, Artix uses the transport to provide reliable message delivery. Artix can also use the local transaction mechanism in JMS to ensure that messages are received without error.

Not all transports, however, have built-in reliable messaging capabilities. To provide reliable messaging across all transports, Artix includes an implementation of WS-RelaibleMessaging (WS-RM) specification. WS-RM defines a mechanism by which messages are transmitted in sequence and both the sender and the receiver use SOAP headers to communicate about the status of the messages that have been transmitted. Message are stored for retransmission until the receiver confirms that it has been received.

Note: Using Artix's WS-RM implementation requires that endpoints use SOAP as their payload format.

Components

As shown in Figure 17, WS-RM is implemented by plug-ins that sit in the messaging chain just before the SOAP binding. In addition to the plug-in, the WS-RM implementation uses an in-memory datastore to hold messages until their successful transmission has been confirmed.

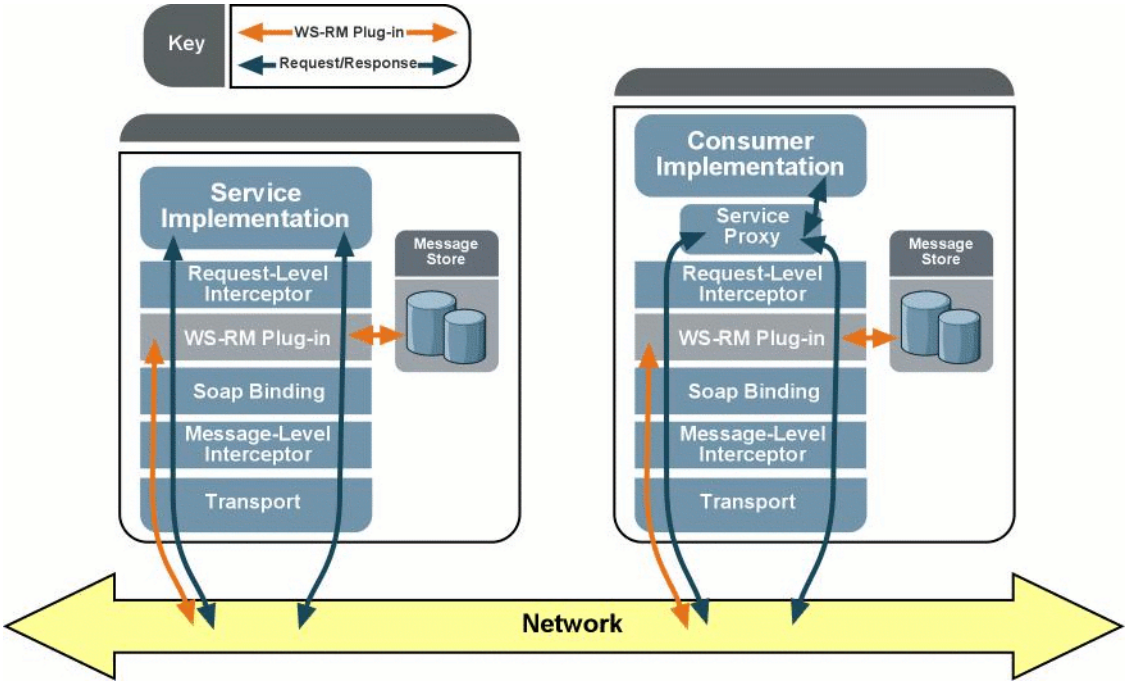

Figure 17: *Overview of WS-RM Architecture*

To use reliable messaging, both endpoints in a request/response sequence must be configured to load the WS-RM plug-in. This information is not part of the contract used to define an endpoint. It is placed in the Artix configuration for each endpoint.

WS-RM sequences

The WS-ReliableMessaging specification defines the notion of reliable message sequences. Each message sent to between sender and receiver are part of a numbered sequence that are tracked using a SOAP header. Using the sequence numbers the receiver can track which messages it has received and, if needed, request that a message be retransmitted.

In Artix, sequences span the lifetime of a service proxy. When a service proxy is created a new message sequence is created and it is terminated when the proxy is destroyed. So, if a proxy makes 50 requests against a service endpoint, the sequence will consist of 50 messages. You can also configure a maximum number of messages in a sequence.

Outgoing messages

When a message, either a request or a response, is passed down the messaging chain, the WS-RM plug-in intercepts the message before it gets to the SOAP binding. Before the message is passed down the rest of the messaging chain, the WS-RM plug-in makes a copy of the message and stores it in memory. The plug-in then attaches a WS-RM header to the message that contains the sequence number of the message. It then passes it down the message chain.

When the recipient confirms that the message arrived, the WS-RM plug-in discards the message. If, after a configured interval, the recipient has not confirmed receipt of a message, the WS-RM plug-in will retransmit the message. This process continues until the recipient confirms receipt of the message.

Incoming messages

The WS-RM plug-in inspects all messages that are received from the network. If it intercepts a message informing it that a message is being sent using WS-RM, it checks its sequence number and informs the sender that it has received the message. Using the sequence number, the plug-in then determines if the message should be passed to the implementation code or stored for later.

The WS-RM plug-in uses the `ExactlyOnceInOrder` policy to determine when a message is passed to the implementation code. This means that only one copy of each message is passed to the implementation code and they are delivered in the order that they were sent. For example, if a consumer makes six requests on a service endpoint the message sequence will consist of six messages numbered 1 through 6. If the receiver gets message 4 before it gets messages 2 and 3, it will store message 4 and wait for

messages 2 and 3. Once it has message 2, it will pass it to the service implementation. If message 3 has already arrived, the WS-RM plug-in will then pass it along. If not, the plug-in will continue to store message 4 until it arrives. When message 3 arrives, the plug-in will pass it to the service implementation. The plug-in will then pass message 4 along and remove it from the message store.

More information

For more information on using Artix's reliable messaging capabilities see:

- Configuring and Deploying Artix Solutions.
- http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-ReliableMessaging.pdf

Transactions

More information

For more information on working with transactions in Artix see the Artix Transactions Guide, C++ or the Artix Transactions Guide, Java.

The Artix Transformer

Overview

The Artix transformer is an XSLT service. It transforms request messages based on directions from an XSLT script and returns the results as the response message. As shown in Figure 18, it consists of a transformer service implementation that when deployed into an Artix container becomes a service endpoint.

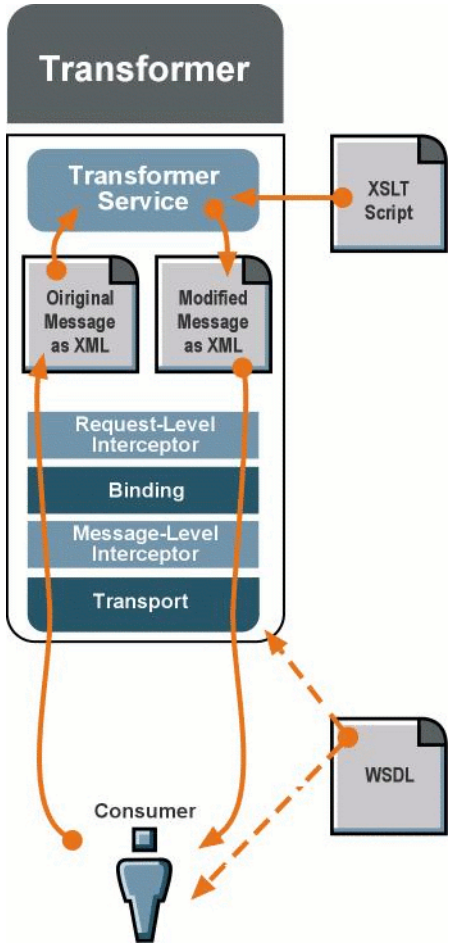

Figure 18: *Overview of the Artix Transformer*

Transformer service WSDL

The transformer service is a dynamic service. Unlike other services in SOA, it does not have a single WSDL document that defines it. Instead, the transformer service configures its interface based on a user supplied WSDL document. The WSDL document defining an instance of the transformer service should have one logical operation for each type of transformation the

service can perform. Each operation's input message should define the XMLSchema used to define the XML data that the transformer service will manipulate. Each operations' output message should define the XMLSchema defining the results of the XSLT script executed when the operation is invoked.

Transformer service processing

Internally, the transformer service receives messages from the messaging layer as XML documents that are constructed using the XMLSchema definitions from the WSDL document. It then uses the XLAN XSLT engine to process the XML document based on an XSLT script. The results of the XLAN engine are placed back onto the messaging chain as the service's response.

More information

For more information on the Artix transformer see Understanding Artix Contracts and Configuring and Deploying Artix Solutions.

The Artix Chain Builder

Overview

The Artix chain builder is an intermediary that allows you to create composite services by linking together two or more services. As shown in Figure 19, the chain builder has a service-side interface that is defined by a WSDL document that defines the input and output of the composite operations it provides. The chain builder's consumer-side consists of one service proxy for each of the backend services it links together to form the composite service.

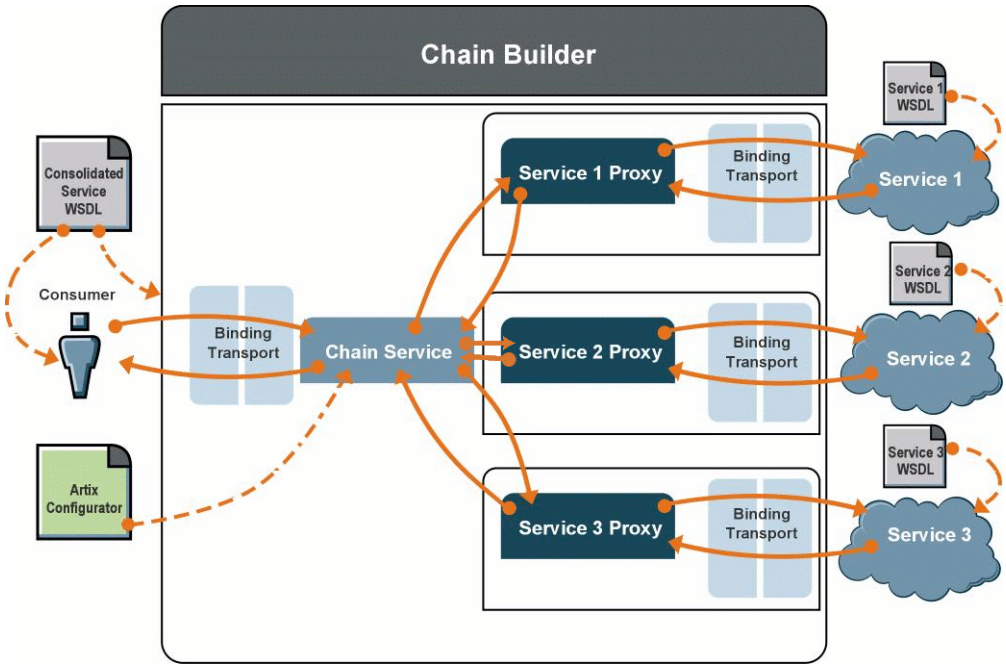

Figure 19: *Overview of the Artix Chain Builder*

Composite WSDL document

A deployed chain builder uses a composite WSDL document to create its service-side interface and consumer-side proxies. The service-side interface is defined by a logical interface that contains at least one operation. The logical operation's input message must correspond to the input message of one of the logical operations defined for the first service in the chain. The logical operation's output message must correspond to the output message of one of the logical operations defined for the last service in the chain. In addition to the logical interface, the composite WSDL document must also contain the information required to expose the composite service as an endpoint.

To deploy the service proxies needed by the consumer-side of the chain builder, the composite WSDL needs to contain endpoint definitions for each service that will be used in the chain.

Chain service

Using directions entered into an Artix configuration file, the chain builder directs the request through the chain. The request received by the chain builder is forwarded to the first service in the chain. The response from the first service is forwarded to the second service in the chain. The response from the 2nd service is forwarded to the next service in the chain. This process is repeated until the last service in the chain is reached. The chain builder returns the response from the last service in the chain to the consumer endpoint that made the request.

More information

For more information on the Artix chain builder see Configuring and Deploying Artix Solutions.

Index