Artix 5.6.4

Building Service Oriented Architectures Using Artix

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

Copyright © Micro Focus 2017. All rights reserved.

MICRO FOCUS, the Micro Focus logo and Micro Focus Licensing are trademarks or registered trademarks of Micro Focus IP Development Limited or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.
All other marks are the property of their respective owners.

2017-02-23

Contents

Preface v
- What is covered in this book v
- Who Should Read This Book v
- How to Use This Book v
- The Artix ESB Documentation Library v
- Further Information and Product Support vi
  - Information We Need vi
  - Contact information vii

Service Oriented Architecture 1
- What is Service Oriented Architecture? 1
  - Service design 2
  - Reuse and integration 3
- Standards 4
- What is an ESB? 5
  - From service to endpoint 5
  - Not EAI 6
- What is a Smart Endpoint? 7
  - Distributing the ESB 7
  - ESB functionality 8
  - Benefits 9
  - Legacy endpoints 9

Artix ESB Enables SOA 11
- Overview of Artix ESB 11
- ESB Architecture 11
  - The Artix ESB bus 12
- Artix ESB C++ Runtime 13
  - Bindings 13
  - Transports 13
  - QoS features 14
- Artix ESB Java runtime 14
  - Bindings 14
  - Transports 14
  - QoS features 15
- Artix ESB in Endpoints 15
  - Artix ESB in a Service Provider 15
  - Artix ESB in a consumer 18
  - Artix ESB in an Intermediary 21
- Artix ESB for C++ Services 23
  - The Router 25

Security 26
The Locator 27
The Session Manager 29

Extending Artix ESB 33
Artix ESB Management Integration 33
Actional 33
AmberPoint 34
BMC Patrol 34
JMX 35

Preface

What is covered in this book

This book discusses the advantages of SOA to integration, what makes a service oriented architecture (SOA), and how Artix ESB 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 ESB 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 into the following chapters:

Service Oriented Architecture provides a general description of service-oriented architectures and how enterprise service buses make them possible.

Artix ESB Enables SOA provides a high-level description of Artix ESB's architecture and how it implements its ESB features. It looks at how Artix ESB connects endpoints to a network using its pluggable messaging stack.

Extending Artix ESB describes ways of extending Artix ESB's functionality through the use of other products in the Artix suite

The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see ***Using the Artix ESB Library***.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Service Oriented Architecture

Service oriented architecture is an architectural style focused on reusing existing applications and designing reusability into new applications. This is accomplished by designing your systems based on loosely-coupled, coarse grained atomic units of functionality called services. The key technology used in building a service oriented architecture is an enterprise service bus that is built using smart endpoints.

What is Service Oriented Architecture?

Service oriented architecture (SOA) is an architectural paradigm emphasizing the reusability of applications in a distributed environment and the alignment of software functionality with business processes. In technical terms SOA means designing applications around a collection of loosely coupled units of functionality with coarse-grained interfaces that are wired together using a common messaging protocol. The units of functionality are exposed by implementation agnostic interfaces that describe the operations exposed by a unit and what messages the unit accepts.

SOA principles can be applied to integrating existing applications as well as to building new applications. First you design a coarse-grained, implementation agnostic facade for the application you wish to integrate. Then you expose the legacy application to the network through the new facade using a common data format/wire protocol combination. The legacy application is now accessible to applications that do not use a proprietary messaging system.

The central concept in SOA is the *service*. A service is the basic unit of functionality in SOA. Like an object in object-oriented programming, a service is an atomic unit of functionality that performs a well-defined and closely related set of operations. They also do not rely on other services to perform the operations they perform. Unlike objects, services are defined by an implementation and language agnostic interface.

A service's interface should be as coarse-grained as possible and provide only the information needed to invoke its operations. The interface is defined as a group of operations. In order to make the interface as coarse-grained as possible, the number of operations should be kept to a minimum. This will help ensure that the amount of detail needed to invoke on a service implementing the interface is kept to a minimum.

The operations that make up the interface are defined by the messages exchanged when the operation is invoked. Messages are typically defined using XML Schema and do not necessarily match the argument list of any implementation of the operation. Ideally, the messages should be coarse-grained. One way of ensuring this is to design messages so that all of the data needed is represented as a single XML document.

Instantiated services are *endpoints*. When instantiated, endpoints add information to the service's interface. The added information includes all of the details needed to access the service. This includes details about what kind of messages (SOAP, fixed, tagged, etc.) the endpoint accepts and the transport over which the endpoint can be accessed.

Service design

Interoperability and reusability are two of the reasons for using SOA. The following guidelines help ensure that services are as interoperable and reusable as possible:

- A service should perform a specific task.

  Services, like objects, are the building blocks of an application. Each block should perform a discreet task so that it can be reused by many applications.

  Because one of the other goals of using SOA is to make it easier to align IT assets with business processes, the task performed by a service should be a business task. For example a service could process a credit card payment.

- A service should not depend on other services.

  A service should, like a toaster, be able to perform its work without any needing to invoke on other services. This does not mean that you cannot design a service that is a composite of other services. A composite service looks and acts like an atomic service to its consumers.

- A service should be stateless.

  When state is shared between two applications there is usually an implicit requirement that each application has some knowledge of the other's implementation. A service that requires its consumers to have an understanding of how it is implemented is not loosely coupled and more difficult to reuse.

- A service uses document style messages.

  Document style messages, as opposed to RPC style messages, promote the use of coarse-grained interfaces.

Service interfaces should be designed to take generic documents as opposed to a specific set of inputs. For example, a loan approval service should be designed to accept a document containing all of the possible pieces of information that could be needed to process a loan request as opposed to the subset that the current implementation requires. Doing so insulates the applications accessing the service from changes in its implementation. Adding a required piece of information to the list of required parameters does not require you to upgrade all of the applications access the service because they will already be sending a properly formed request.

- A service cannot assume that its consumers are operating in the same environment.

  To ensure maximum reusability and maximum interoperability, a service should not require its consumers to be operating in a particular environment. For example, a consumer running on a Windows system in Europe should be able to make requests on a service endpoint running on a z/OS system running in the United States. The service should be completely implementation agnostic.

Reuse and integration

Companies have millions of dollars invested in their existing IT systems and one of the main drivers for adopting a new development model is to get the most out of those existing systems. Another main driver is the desire to break out of the vendor lock-in. They are looking for a solution that allows them to reuse what they already have in new ways and ensure that future systems will have the same, if not more flexibility to be reused.

Reusability is one of the central goals of using SOA This goes beyond simply creating new services so that they are reusable and flexible enough to be recombined into new applications when needed. SOA embraces the idea that legacy systems also need to be reused and integrated with other systems to create new applications.

To achieve this reusability, you need to model your existing systems as services using the tooling provided with a SOA development platform. You may find that it is hard to model your legacy systems using coarse-grained interfaces that strictly adhere to SOA principles. This can be overcome using other features of your SOA infrastructure that can allow you further abstract the interface from your legacy system's fine-grained interfaces.

Once a legacy system is wrapped in a service interface, it will be accessible just like any other service deployed in the SOA

infrastructure. Because consumers will only see the legacy system through the service interface, they will not need to be aware of how the functionality is provided. All the consumer knows is that it sends request messages to an endpoint and reply messages are returned from the endpoint.

Standards

One of the ways that SOA achieves its goals is through the use of standardized technologies. Chief among these standards is XML. It provides the underlying grammars that SOA uses as building blocks.

One of the fundamental building blocks used in SOA is Web Service Definition Language (WSDL). WSDL is an XML based grammar that is used to define service interfaces. It breaks the definition of a service into its logical interface and the physical details used to instantiate endpoints. For more information on WSDL see *Writing Artix Contracts*.

Another fundamental building block used in SOA is XML Schema. XML Schema provides the type system used in defining service interfaces. It is used to define the abstract representation of the messages that define a service's operations. These abstract representations can then be mapped into concrete messages using WSDL.

In addition to WSDL and XML Schema, SOA takes advantage of a number of other standards that are grouped together into what is known as the WS* family of specifications. These specifications include:

- WS-Addressing
- WS-Policy
- WS-AtomicTransactions

These standards are all maintained by OASIS.

- WS-ReliableMessaging
- WS-Security

These standards are maintained by the W3C.

The standards provide a common framework on which SOA builds QoS. They were all designed around the idea that information would be passed using SOAP/HTTP, but they can be leveraged by a number of different messaging protocols. They were also designed so that services could be easily shared and accessed over the Web. Therefore, they are built to be maximally interoperable.

What is an ESB?

An enterprise service bus (ESB) is the layer of technology that makes SOA possible. It creates the necessary abstractions by translating the messages which define 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.

From service to endpoint

An ESB takes the concrete details defined in the WSDL contract and uses it to create an endpoint that implements a service. This information includes details on how the abstract messages are mapped into data 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 1 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.

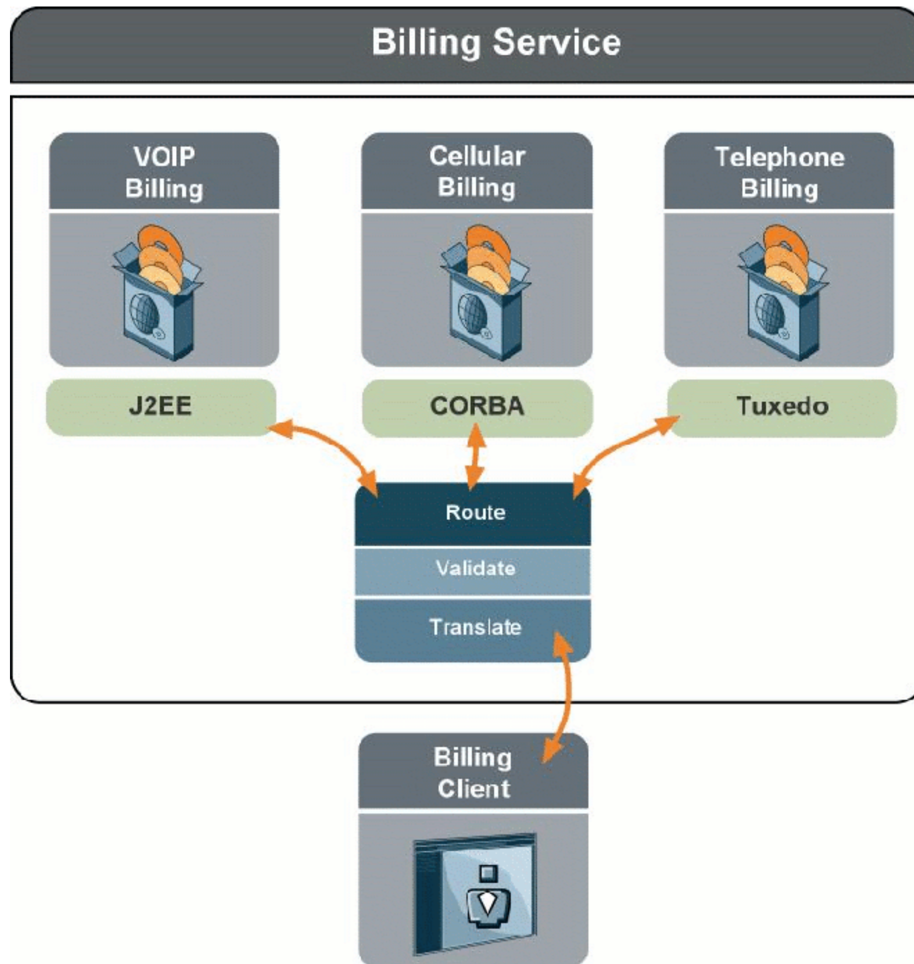

Figure 1. Billing System SOA with an ESB

Not EAI

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

- ESBs use industry standard WSDL contracts to define the endpoints they connect.
- ESBs use XML as a native type system.
- ESBs are deployed in a distributed manner.
- 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-proof and flexible. As discussed in the previous section, both are platform and implementation neutral which avoids vendor lock-in.

What is a Smart Endpoint?

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 intended to be as distributed as the components they are integrating. In Artix this is accomplished by implementing the ESB as a series of smart endpoints.

A *smart endpoint* is an endpoint that is capable of performing a number of the features of an ESB. Smart endpoints make an ESB distributed by moving its functionality out of a centralized server and putting that functionality where it is needed.

Distributing the ESB

As shown in Figure 2, an ESB distributes the work 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 affecting 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.

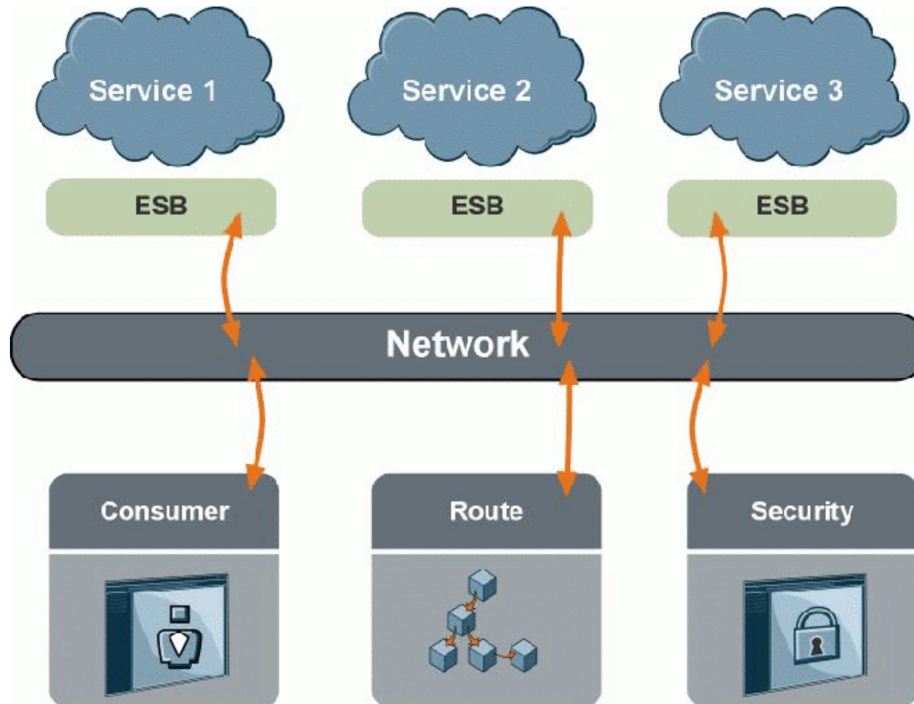

Figure 2. Distributed Nature of an ESB

The distributed nature of an ESB also means that you are not forced to drop all 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 add services 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 capabilities allow you to plug in legacy implementations.

ESB functionality

The major responsibilities of the ESB that are assumed by smart endpoints include:

- translation of requests and responses into usable data
- publication of a service's WSDL
- interactions with the transports
- message reliability
- transactions

The rest of the ESB's responsibilities are distributed across several discreet services that are also exposed as individual smart endpoints.

Benefits

Smart endpoints provide several benefits. These include:

- the flexibility to rapidly change your messaging infrastructure without reimplementing functionality.
- the ability to scale the number of endpoints implementing a service to meet demand.
- the ability to incrementally deploy services into your infrastructure without disrupting your existing systems.
- the flexibility to spread the load across your existing hardware as you need.

Legacy endpoints

It may seem impossible to expose a legacy application as a smart endpoint without re-implementing it. While it is true that legacy systems tend to be tied to a fixed messaging system, you can use a smart endpoint to expose the legacy system's functionality. This is done by using a smart endpoint to intercept requests directed at the legacy system. The endpoint will then translate the request into the appropriate format for the legacy application and pass the request over the appropriate transport.

Your legacy application will appear to be a smart endpoint to the rest of your infrastructure. This makes it easier to reuse the functionality of the legacy application. It also makes it easier to replace the legacy application with new technology when the time comes.

Artix ESB Enables SOA

Artix ESB is a fully distributed ESB. It is built around the concept that all of the endpoints in your SOA are smart. Artix ESB accomplishes this by building the ESB functionality into the runtime libraries that are loaded by deployed endpoints. Artix ESB also provides a number of services that provide features such as location independence, security, and routing.

Overview of Artix ESB

The Artix ESB products, Artix ESB for Java, and Artix ESB for C++, provide the following functionality:

- data and transport abstraction
- C++ runtime
- Java runtime
- message routing
- security
- transactions
- reliable messaging
- location resolution
- high availability
- design time tooling

In addition, Artix ESB can be supplemented to include robust orchestration tools and mainframe connectivity.

ESB Architecture

Because Artix ESB is an enterprise service bus, it is easy to picture it as a pipe, or wire, that transports data between endpoints. While there are a number of ESB implementations that are designed like a data pipe, Artix ESB is designed as a set of caps that allow the endpoints to connect to a number of different pipes. In essence, it turns whatever messaging infrastructure you have deployed into a virtual ESB.

As shown in Figure 3, the Artix ESB runtime components are embedded into the endpoints deployed as part of your SOA. Artix ESB enabled endpoints are smart and are capable of handling all of the data and transport abstraction needed to connect to the network, regardless of the messaging infrastructure in use.

Because of the pluggable nature of the Artix ESB runtime components, the endpoints only load the pieces of the runtime needed to connect to the specified messaging infrastructure.

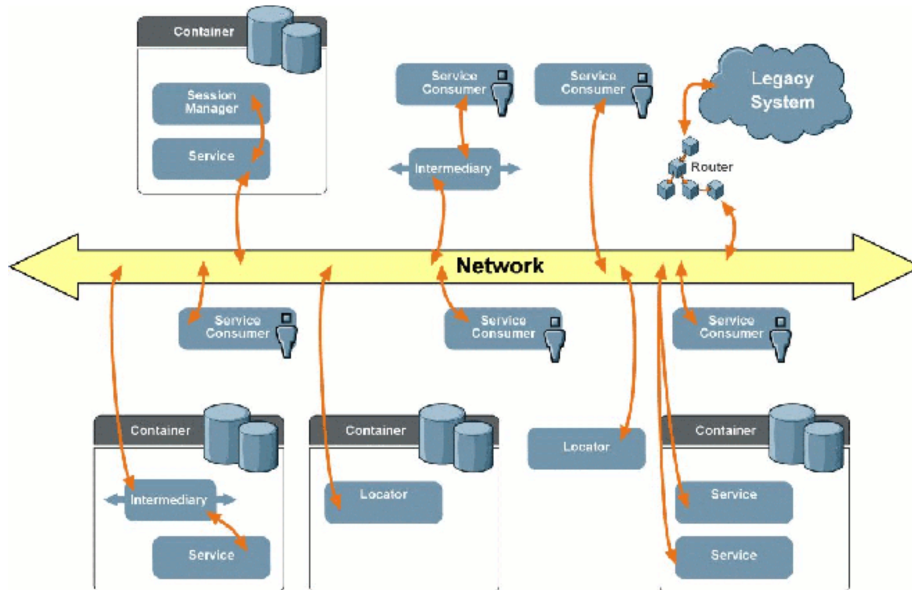

Figure 3. Artix ESB and the Virtual Bus

Because the endpoints do the work of negotiating the transport and message format details independent of each other, the ESB functionality is distributed across your entire deployment. The endpoints also have some of the logic needed for transaction management, security, and location resolution embedded into them.

Features like routing, transaction management, security, location resolution, and high-availability use components that are also deployed as smart endpoints. They can be spread across resources as needed.

The Artix ESB bus

Artix ESB does have a bus, but it is internal. The bus coordinates the passage of data from the user implemented business logic to the networking system. Internally, Artix ESB consists of the bus and a number of objects that take the data that the business logic manipulates and transforms it into a message that is sent on the network. There are also a number of objects that Artix ESB 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-ins used by Artix ESB. 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.

Capitalizing on the existing infrastructure

Artix ESB 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 ESB uses the existing network infrastructure to deliver messages, it can capitalize on any QoS offered by the network. For example, Artix ESB can use the reliable messaging mechanisms offered by a JMS queue to ensure that messages are delivered.

Artix ESB C++ Runtime

Artix ESB C++ Runtime provides developers with a C++ API with which to implement services. It is built on top of the ART runtime. Artix ESB C++ Runtime has a C++ core that provides a fast and stable platform for building applications.

Bindings

Artix ESB C++ Runtime supports the following message format bindings:

- • SOAP (1.1 and 1.2)
- CORBA
- Pure XML
- Fixed length records
- Tagged data
- FML buffers

Transports

Artix ESB C++ Runtime supports the following transports:

- HTTP
- JMS
- IIOP
- FTP
- WebSphere MQ
- Tuxedo

QoS features

Artix ESB C++ Runtime supports the following QoS features:

- message routing
- security
- transactions
- reliable messaging
- high-availability
- load balancing
- location resolution
- statefulness

Artix ESB Java runtime

Artix ESB Java Runtime provides the developer with both a JAX-WS 2.0 API and a JavaScript API with which to implement services. It is based on Apache CXF and provides a fast, modular, and extensible platform for implementing services that is built purely in Java.

Bindings

Artix ESB Java Runtime supports the following message format bindings:

- SOAP (1.1 and 1.2)
- MTOM/XOP
- RESTful
- CORBA
- Pure XML

Transports

Artix ESB Java Runtime supports the following transports:

- HTTP
- JMS
- FTP

- WebSphere MQ

QoS features

Artix ESB Java Runtime supports the following QoS features:

- message routing
- security
- reliable messaging
- high-availability
- load balancing
- location resolution

Artix ESB in Endpoints

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

- Service providers are endpoints that implement the operations defined in a service contract. They are similar to servers.
- Consumers are endpoints that make requests on services. They are similar to clients.
- Intermediaries are endpoints that process 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 provider and a consumer.

Artix ESB in a Service Provider

A *service provider* 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 ESB code generators, you can create a service endpoint that uses Artix to connect to the network. Artix ESB can load any components needed to provide the desired features.

What makes up a service endpoint

As shown in Figure 4, a service provider built with Artix ESB has the following pieces:

- a service implementation
- a binding layer

- a transport layer

In addition, a service provider can have any number of request-level and message-level interceptors that provide added functionality. These interceptors, which are independent of the service provider's contract, 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.

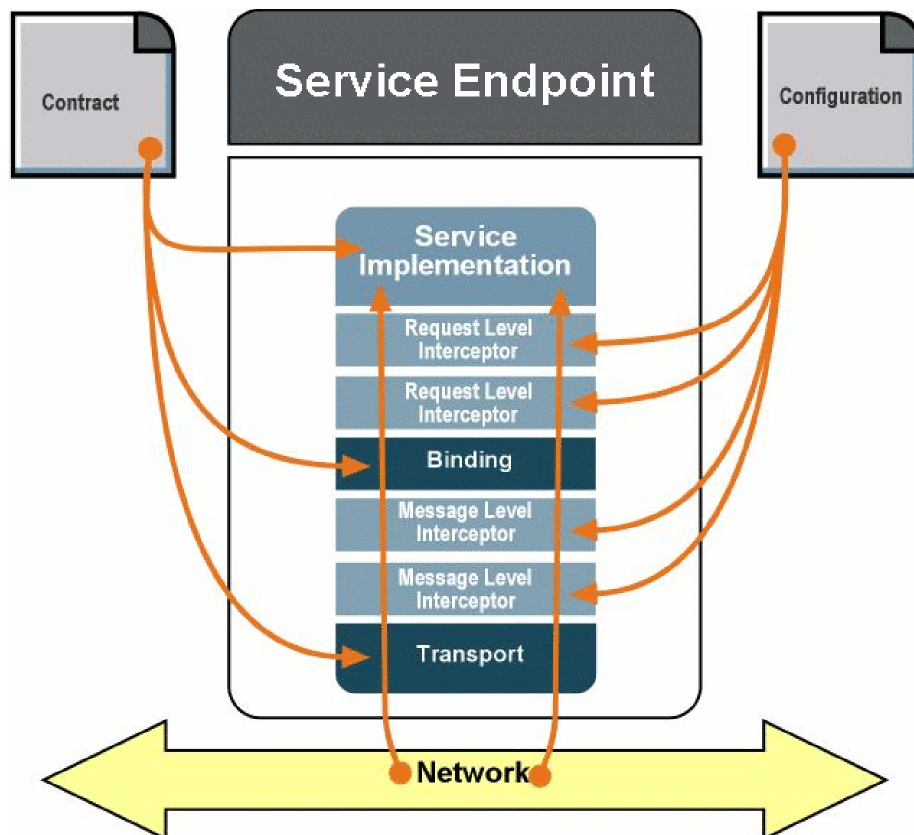

Figure 4. High-level View of a Service Provider

Service Implementation

The service implementation in Artix ESB for C++ can be created using C++. The service implementation in Artix ESB for Java can be created using Java and is based on code generated from the logical portion of the service endpoint's contract. Artix ESB 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 runtime.

The implementation does not have direct access to the request messages. It receives messages from the Artix ESB runtime as parameters to the operations specified in the contract from

which it was generated. Similarly, it returns any responses to the bus as a return value. The marshaling of the data is handled by the binding plug-in. The service implementation has no knowledge of how the messages are packaged.

Exceptions thrown in the implementation object are also passed to the messaging chain. 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 contract. 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. For example, if a payment being passed to a `make_payment()` operation 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 values can also be inspected and changed.

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.

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. Artix will load the appropriate binding based on the `binding` elements in the contract defining the endpoint.

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, message-level 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.

Transport

The transport is responsible for pulling requests off of the network and placing responses back on the network. The transport to be loaded and their configuration are determined by the WSDL `port` elements included in the contract defining the endpoint.

Artix ESB in a consumer

A *consumer* is an endpoint that makes requests on a service provider. Using stub code produced by running a contract through the Artix ESB code generators, you can create a consumer that uses Artix ESB to load a service proxy for the service defined by the contract and connect to one of the service providers implementing that service. The bus can also load any components needed to provide the features you desire.

What makes up a consumer

As shown in Figure 5, a consumer built with Artix ESB has the following pieces:

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

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 WSDL document defining the service's interface, 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.

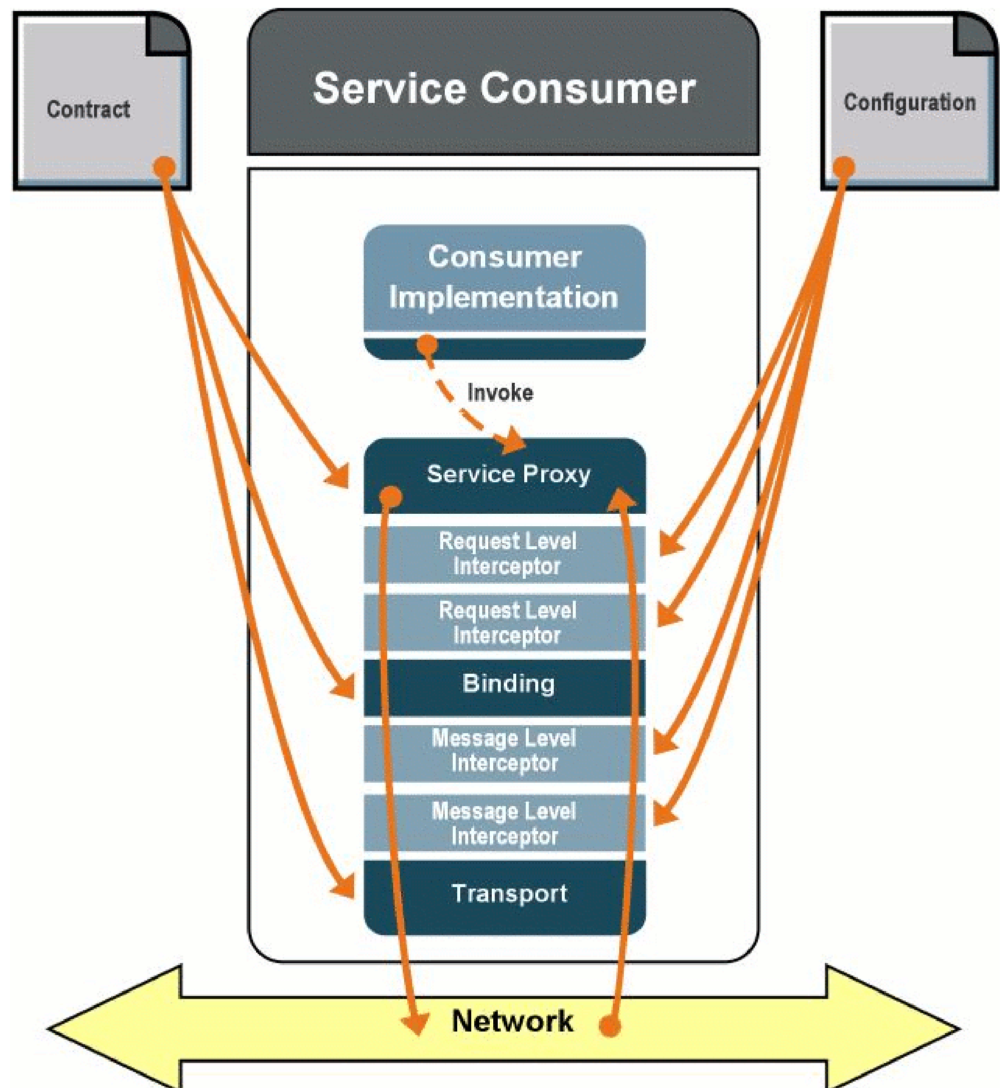

Figure 5. High-level View of a Consumer

Consumer implementation

The consumer implementation 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.

Service proxy

The service proxy is a stub generated from the logical portion of a contract defining the service upon which the consumer will make requests. It allows a consumer to invoke the operations offered by a service provider.

When instantiated, a service proxy provides a connection to a service provider that implements the service defined in the

contract from which it was generated. As part of their instantiation, service proxies are registered with the runtime so that the invocations made on the service proxy can be properly delivered to the desired service provider.

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.

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.

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 consumer implementation is responsible for properly handling the exception.

Binding

The binding is responsible for converting messages between the binary types used by the consumer implementation and the data format used on the wire. The mapping is determined by the WSDL `binding` element. Artix loads the appropriate bindings based on the `binding` elements in the contract defining the service to which the client is making requests.

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.

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 transports and their configuration are determined by the WSDL `port` elements in the contract defining the service endpoint on which the consumer endpoint is invoking.

Artix ESB in an Intermediary

An *intermediary* is a special case of a service provider. It is a service provider 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 provider and some of the characteristics of a consumer. They are typically defined by a contract 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 contract, 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. Artix ESB will use the information in the contract to load the components needed to connect the intermediary to the network.

Artix ESB 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 Router.

What makes up an intermediary

As shown in Figure 6, an intermediary built using Artix ESB has the following pieces:

- a service-side transport

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

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.

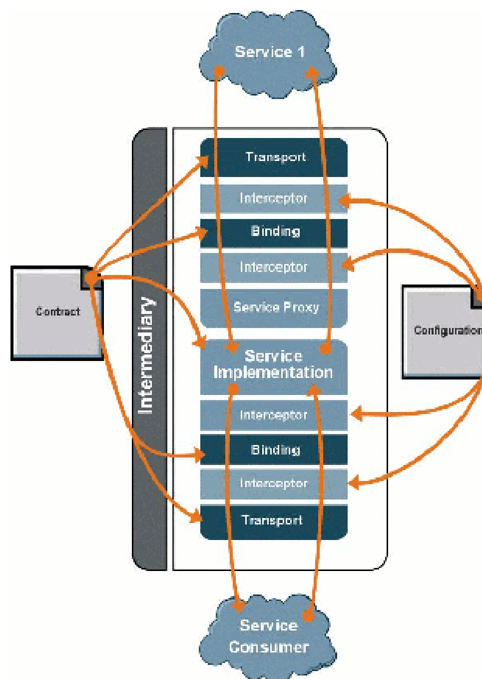

Figure 6. High-level View of an Intermediary

Service-side messaging chain

An intermediary's service-side messaging chain functions identically to the messaging chain of a service provider. 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 ESB in a Service Provider.

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 ESB 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. 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 ESB in a Consumer.

Artix ESB for C++ Services

Features such as location independence, message routing, and security require functionality that cannot be built into a smart endpoint. To address this Artix ESB provides a number of service providers that you deploy into your SOA.

The Artix ESB C++ Runtime Container

One of the key features of SOA is that its endpoints are highly dynamic. The Artix ESB C++ Runtime container provides a number of features that make 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 alongside the endpoints hosted in the container.

Container server

The container server is a light weight process that can host a number of Artix enabled endpoints. It instantiates service implementation objects, loads the bindings and transports specified in the contracts of the endpoints the container is hosting, and exposes the endpoints to the network. The container coordinates the flow of messages so that messages are delivered to the appropriate service implementations.

In addition to the endpoints you deploy into a container, Artix ESB C++ Runtime containers always load an instance of the container administrative service.

Administrative service

The container's administrative service allows you to manage the endpoints deployed in a container. Like all services in SOA, the administrative service is defined by a contract. By default the administrative service is exposed as a SOAP/HTTP endpoint 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 ESB C++ Runtime.

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 contract for an endpoint
- Get the URL to an endpoint's contract document
- Retrieve performance metrics for an endpoint
- Shut down the container

The Router

The router is an intermediary whose primary role is to redirect messages based on rules defined in its contract. As shown in Figure 7, a 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.

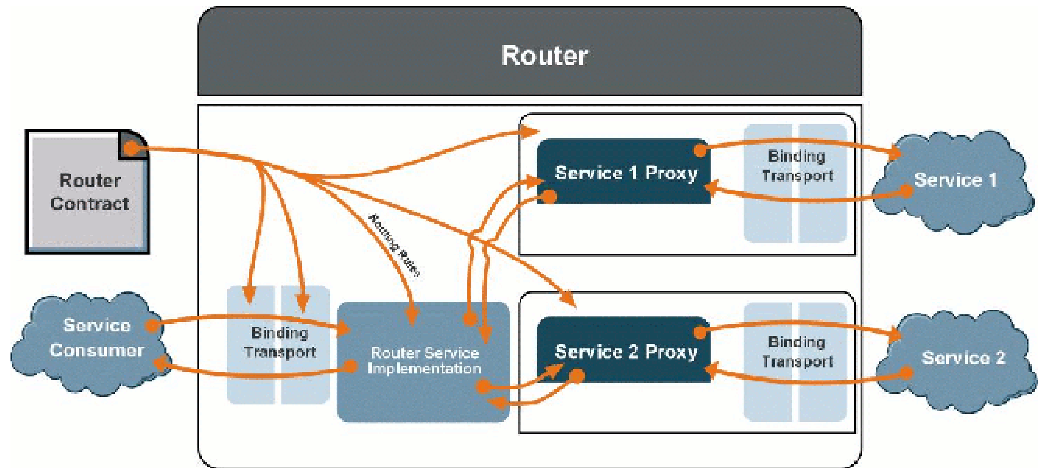

Figure 7. Overview of the Artix Router

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

Features

A router provides a number of features:

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

Service-side

The service-side of a router looks like a service provider to the other endpoints on your network. It is responsible for receiving requests from consumers that make requests on the service provider, or service providers, behind the router. Its interface and messaging chain is determined by a service definition in the router's contract.

Consumer-side

The consumer-side of a router looks like a consumer 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 service providers on the backend of the router. The proxies, and their messaging chains, are defined in the router's contract. However, they are not instantiated until they are needed by the router. So, if one of the destinations in the router's contract 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.

More information

For more information about the router see the ***Artix Router Guide***.

Security

Artix ESB's security architecture is designed to be easily deployable and easily connected to any existing security infrastructure already in use. It consists of two main components:

- the security plug-in
- the Artix security service

Security plug-in

The security plug-in is deployed into the message chain of any service provider 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 service

The Artix security service provides the authentication and authorization functionality for Artix service providers. 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 its functions.

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 Locator

The locator is a lightweight registry of deployed service endpoints. Service endpoints register 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 8, the locator consists of three components:

- the locator service
- the locator endpoint plug-in
- the locator client plug-in

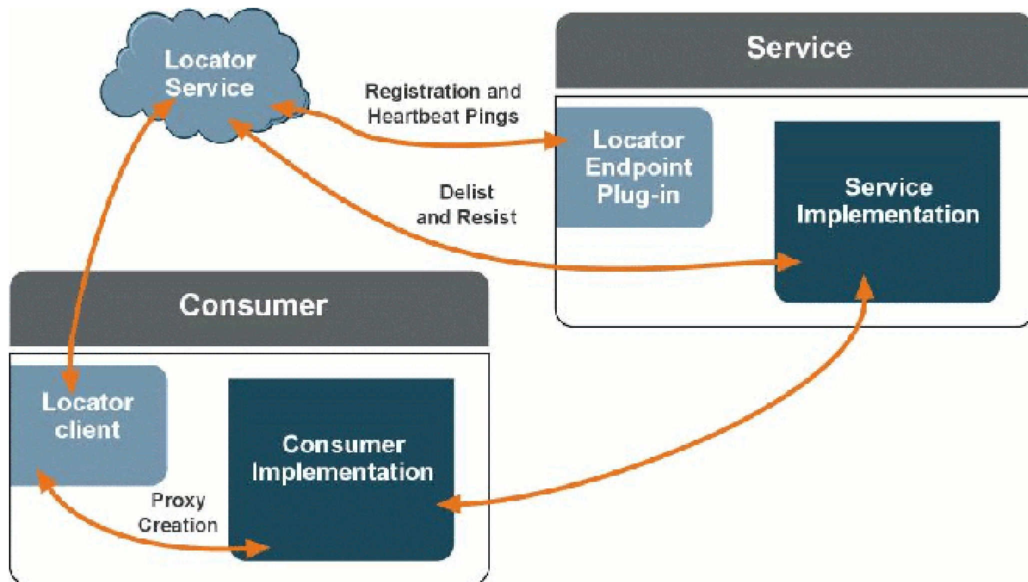

Figure 8. Overview of the Locator

Features

The locator has the following features:

- look up of references to deployed service endpoints
- load balancing among endpoints that implement the same service
- high availability

Locator service

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

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

The locator contract 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.

Because the locator service is defined by a standard contract and deployed as a SOAP/HTTP endpoint, it can be used by any endpoint in your SOA that communicates using SOAP/HTTP. For instance if you have .NET clients that want to use the locator to find service instances, it is not a problem. You could also register Axis based services with an instance of the locator service. All a non-Artix ESB client needs to do is generate a service proxy for making requests against the locator service.

Locator endpoint plug-in

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

Locator client plug-in

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

More information

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

The Session Manager

The 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

The session manager is implemented in a modular fashion. It consists of the following components:

- the session manager service
- a policy plug-in that is collocated with each instance of the service
- an endpoint manager plug-in that is collocated with all managed service providers
- a session token interceptor that sits in the messaging chain of all managed service providers

Session manager service

The session manager service is defined by a WSDL document and is implemented by a library shipped with Artix ESB. You deploy instances of the session manager service implementation into an Artix ESB C++ Runtime container to create session manager service providers. These service providers can be accessed by any consumer 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 a session manager service instance. 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 ESB includes with a default policy plug-in called `sm_simple_policy`. This plug-in uses information from the session manager's 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 an Artix service providers that wants to register with a session manager instance. The endpoint managers are in constant communication with the session manager instance to report on the endpoint's health, to receive information on new sessions that have been granted to the managed service providers, and to check on the health of the session manager instance.

Session token interceptor

The session token interceptor is placed in a service provider'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 session manager see the Session Manager.

Extending Artix ESB

In addition to Artix ESB, you can add other packages from the Artix suite to extend your SOA infrastructure. These packages offer features like mainframe connectivity, orchestration, .NET integration, and repository functionality.

Artix ESB Management Integration

Artix ESB enables you to use integrate Artix ESB C++ Runtime and Artix ESB Java Runtime services with a number of enterprise management systems. These include:

- Actional
- AmberPoint
- BMC Patrol
- Java Management Extensions (JMX)

Actional

Integration between Artix ESB and Actional enables Artix ESB services to be monitored by Actional SOA management products. For example, you can use Actional SOA management tools to perform monitoring, auditing, and reporting on Artix ESB services. You can also correlate and track messages through your network to perform dependency mapping and root cause analysis.

The Artix–Actional integration is deployed on Artix ESB service endpoints to enable reporting of management data back to the Actional server. The data reported back to Actional includes system administration metrics such as response time, fault location, auditing, and alerts based on policies and rules.

The integration relies on two components to monitor your services and report the data back to the Actional SOA management tools:

Actional Agents

An Actional agent is run on each service endpoint that you wish to manage. Actional agents are used to provide instrumentation data back to the Actional server. Actional agents are provisioned from the Actional server to establish initial contact and send configuration to the Actional agent. There is one Actional agent per service endpoint.

Artix ESB Interceptors

Interceptors are added to an endpoint's messaging chain that send the instrumentation data to the Actional agent using an Actional-specific API. These interceptors essentially push events to the Actional agent. The data is analyzed and stored in the Actional agent for retrieval later by the Actional server. However, any alerts triggered at the Actional agent are sent immediately to the Actional server.

AmberPoint

The Artix ESB AmberPoint Agent can be deployed with Artix ESB endpoints that use SOAP over HTTP to enable reporting of performance metrics back to AmberPoint.

The agent enables the use of the following AmberPoint features:

- Dynamic discovery of Artix ESB clients and services using SOAP over HTTP.
- Monitoring of Artix ESB client and service invocations, and reporting them back to AmberPoint.
- Mapping Qualities of Service to customer Service Level Agreements (SLAs).
- Monitoring of Artix ESB invocation flow dependencies, which enables AmberPoint to draw Web service dependency diagrams.
- Centralized logging and performance statistics.

BMC Patrol

The Artix ESB BMC Patrol integration performs the following key enterprise management tasks:

- Posting an event when a server crashes. This enables programmed recovery actions to be taken.
- Tracking key server metrics (for example, server response times). Alarms are triggered when these go out of bounds.

The server metrics tracked by the BMC Patrol integration include the number of invocations received, and the average, maximum and minimum response times. The BMC Patrol integration also enables you to track these metrics for individual operations. Events can be generated when any of these parameters go out of bounds. You can also perform a number of actions on servers including stopping, starting and restarting.

In the BMC Patrol integration, key server metrics are logged by performance logging plugins. The BMC Patrol integration provides Artix ESB Knowledge Modules, which conform to standard BMC Knowledge Module design and operation. These modules tell the BMC Patrol console how to interpret the logging data.

JMX

Artix ESB endpoints are instrumented to allow runtime components to be exposed as JMX Managed Beans (MBeans). This enables an endpoint to be monitored and managed either in process or remotely with the help of the JMX Remote API. In addition to providing instrumented runtime components, Artix ESB allows you to build and register custom MBeans so you can monitor metrics that are specific to an application.

For Artix ESB C++ Runtime endpoints you can monitor the following runtime components out of the box:

- Bus
- Service
- Port

For Artix ESB Java Runtime endpoints you can monitor the following runtime components out of the box:

- bus
- Service endpoint