IONA®

Artix™

Deploying & Managing Artix Solutions

Version 2.1, September 2004

Making Software Work Together™

Updated: 23-Sep-2005

M3203

Contents

Part I Introduction

Part II Artix Configuration and Management

Part III Using Artix Services

Part IV Integrating with Other Middleware Systems

List of Tables

List of Figures

Preface

What is Covered in this Book

Deploying and Managing Artix Solutions explains how to configure, deploy, and manage IONA Artix runtime solutions. It presents different approaches to deployment topography, and the merits of each. This book also provides detailed descriptions of the specific tasks involved in configuring and launching Artix applications and services.

This book does discuss the specifics of the different middleware and messaging products that Artix interacts with. Any discussion about the features of specific middleware products or transports relates to how Artix interacts with these features. It is assumed that you have a working knowledge of the specific middleware products and transports you are using.

Who Should Read this Book

The main audience of *Deploying and Managing Artix Solutions* is Artix system administrators. However, anyone involved in designing a large scale Artix solution will find the general discussions about Artix deployment topographies and Artix services useful.

Knowledge of specific middleware or messaging transports is not required to understand the general topics discussed in this book. However, if you are using this book as a guide to deploying runtime systems, you should have a working knowledge of the middleware transports that you intend to use in your Artix solutions.

How to Use this Book

Part I, Introduction

If you are new to Artix, Chapter 1 and Chapter 2 provide a high-level overview of using Artix to solve integration projects, and how Artix fits into a software environment.

Part II, Configuring and Managing Artix

To learn how to configure a system to run Artix solutions, read Chapter 3. This chapter describes how to set up your environment to run Artix services, and explains the Artix configuration mechanism. Chapter 5 and Chapter 4 provide detailed reference information on Artix configuration variables. They explain how to configure your Artix services for optimal performance.

If you need to use the logging features of Artix, read Chapter 4. This provides a detailed discussion of using the advanced logging features of Artix. Chapter 5 provides an overview of integrating Artix with Enterprise Management Systems (for example, IBM Tivoli and BMC Patrol).

To learn how Artix handles codeset conversions, read Chapter 6. This chapter provides a detailed discussion of which payload formats support international codesets and how each transport handles them. It also provides a description of how Artix handles codeset conversions when routing between endpoints.

Part III, Using Artix Services

If you are using any Artix services you may want to read one or more of the following:

- Chapter 7 explains how to use Artix standalone service.
- Chapter 9 explains how to use the Artix locator service.
- Chapter 10 explains how to use the Artix session manager.

> **Note:** The session manager is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports the session manager.

Part IV, Integrating with Other Middleware Systems

If you are using Artix to integrate with another middleware product you may want to read one or more of the following:

- Chapter 13 describes how to deploy Artix into a CORBA environment.
- Chapter 14 describes how to deploy Artix into a BEA Tuxedo environment.

  Note: Tuxedo integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports Tuxedo integration.

- Chapter 15 describes how to integrate Artix with a deployed J2EE system.

Online Help

While using the Artix Designer you can access contextual online help, providing:

- A description of your current Artix Designer screen.
- Detailed step-by-step instructions on how to perform tasks from this screen.
- A comprehensive index and glossary.
- A full search feature.

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

- Click the **Help** button on the Artix Designer panel, or
- Select **Contents** from the Help menu.

Finding Your Way Around the Artix Library

The Artix library contains several books that provide assistance for any of the tasks you are trying to perform. The remainder of the Artix library is listed here, with an short description of each book.

If you are new to Artix

You may be interested in reading *Learning About Artix*. This book describes the basic Artix concepts. It also walks you through an example of using Artix to solve a real world problem using code provided in the product.

To design Artix solutions

You should read *Designing Artix Solutions.* This book provides detailed information about using the Artix Designer GUI to create WSDL-based Artix contracts, Artix stub and skeleton code, and Artix deployment descriptors.

This book also provides detailed information about Artix command-line interface and the WSDL extensions used in Artix contracts. It also explains the mappings between data types and Artix bindings.

To develop applications using Artix stub and skeleton code

Depending on your development environment you should read one or more of the following:

- *Developing Artix Applications in C++*. This book discusses the technical aspects of programming applications using the Artix C++ API.
- *Developing Artix Applications in Java*. This book discusses the technical aspects of programming applications using the Artix Java API.

To configure and manage your Artix solution

You should read *Deploying and Managing Artix Solutions.* This describes how to configure and deploy Artix-enabled systems. It also discusses how to manage them when they are deployed.

In addition, if you are integrating Artix with either the IBM Tivoli or BMC Patrol Enterprise Management System, you should read:

- *IONA Tivoli Integration Guide*.
- *IONA BMC Patrol Integration Guide*.

To learn more about Artix security

You should read the *Artix Security Guide*. This outlines how to enable and configure Artix's security features. It also discusses how to integrate Artix solutions into a secure environment.

Have you got the latest version?

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs. Compare the version details provided there with the last updated date printed on the inside cover of the book you are using (at the bottom of the copyright notice).

Additional Resources for Help

The IONA Knowledge Base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about Artix and other products.

The IONA Update Center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products.

If you need help with this or any other IONA products, go to IONA Online Support (http://www.iona.com/support/index.xml).

Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical Conventions

This book uses the following typographical conventions:

`Constant width`

Constant 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 `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

Italic

Italic words in normal text represent *emphasis* and *new terms*.

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/`*`your_name`*

Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying Conventions

This book uses the following keying conventions:

No prompt

When a command's format is the same for multiple platforms, a prompt is not used.

`%`

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 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.

|

A vertical bar separates items in a list of choices enclosed in {} (braces) in format and syntax descriptions.

Part I

Introduction

In this part

This part contains the following chapters:

CHAPTER 1

Introduction to Artix

Artix enables you to deploy integration solutions that are middleware-neutral.

In this chapter

This chapter contains the following sections:

What is Artix?

Overview

Artix provides a middleware connectivity solution that minimizes invasiveness and prevents an organization from being locked into any one middleware transport. For example, Artix can be used to connect a BEA Tuxedo™ server to a CORBA client. Artix transparently handles the message mapping and transformation between them. The Tuxedo server is unaware that its client is using CORBA. For example, with Artix handling the communication, the client could be changed to an IBM WebSphere MQ™ client without modifying the server.

Scalable infrastructure

Artix also provides a great deal of configurability because it is built on IONA's Adaptive Runtime architecture (ART). All of Artix's transport and payload format support is encapsulated in individual plug-ins as are all of the services provided with Artix. This allows Artix to be scaled to fit any environment.

Artix message transporting

Artix shields applications from the details of the transports used by applications that they are communicating with, by providing on-the-wire message transformation and mapping. Unlike the approach taken by Enterprise Application Integration (EAI) products, Artix does not use an intermediate canonical format; it transforms the messages only once.

Figure 1 shows a high-level view of how a message passes through Artix. The approach taken by Artix provides a high-level of throughput by avoiding the overhead of making two transformations for each message.

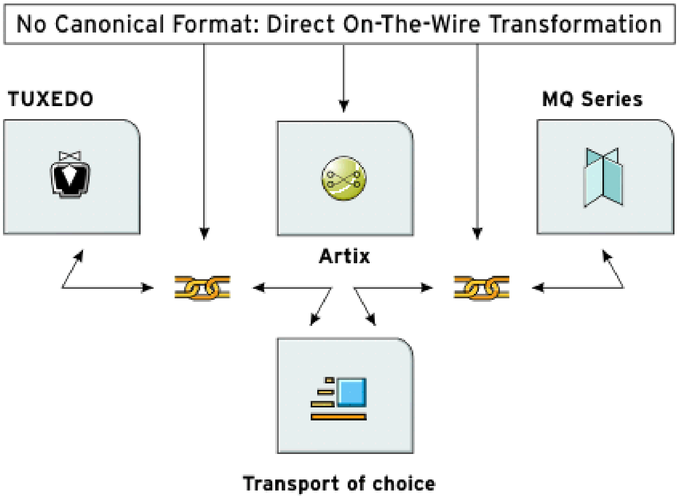

Figure 1: *Artix Message Transporting*

Supported message transports

Artix supports the following message transports:

- HTTP
- BEA Tuxedo
- IBM WebSphere MQ
- IIOP
- TIBCO Rendezvous™
- IIOP Tunnel

Supported payload formats

Artix can automatically transform between the following payload formats:

- G2++
- FML (Tuxedo format)
- GIOP (CORBA format)
- FRL (Fixed Record Length)
- VRL (Variable Record Length)
- SOAP
- TibrvMsg (TIBCO Rendezvous format)

Artix Concepts

Overview

This section explains some of the high-level concepts behind Artix. For example, Artix contracts, and their components, and Artix deployment modes. For more detailed information on Artix, see *Learning About Artix*.

Artix contracts

An Artix contract defines the interaction of a Service Access Point (SAP) or endpoint with Artix. Contracts are written using a superset of the standard Web Service Definition Language (WSDL). Following the procedure described by W3C, IONA has extended WSDL to support Artix's advanced functionality, and use of transports and formats other than HTTP and SOAP.

An Artix contract consists of two parts:

Logical

The logical portion of the contract defines the namespaces, messages, and operations that the SAP exposes. This part of the contract is independent of the underlying transports and wire formats. It fully specifies the data structures and possible operation/interaction with the interface. It consists of the WSDL tags `<message>`, `<operation>`, and `<portType>`.

Physical

The physical portion of the contract defines the transports, wire formats, and routing information used to deliver messages to and from SAPs, over the bus. This portion of the contract also defines which messages use each of the defined transports and bindings. The physical portion of the contract consists of the standard WSDL tags `<binding>`, `<port>`, and `<operation>`. It is also the portion of the contract that may contain IONA WSDL extensions.

Deployment modes

Applications that use Artix can be deployed in one of two ways:

Embedded mode

In embedded mode, an application is modified to invoke Artix functions directly and locally, as opposed to invoking a standalone Artix service. This approach is the most invasive to the application, but also provides the highest performance. Embedded mode requires linking the application with Artix-generated stubs and skeletons to connect client and server (respectively) to Artix.

Standalone mode

In standalone mode, Artix runs as a separate process invoked as a service. In this deployment mode, Artix provides a zero-touch integration solution on the application side. When designing a system, you simply generate and deploy the Artix contracts that specify each endpoint. Because a standalone switch is not linked directly with the applications that use it (as in embedded mode), a contract for standalone mode deployment must specify routing information. This is the least efficient of the two modes.

For more detailed information on Artix deployment modes, see Chapter 2.

Advanced Features

Artix also supports the following advanced functionality:

- Message routing based on the operation or port, or port characteristics.
- Transaction support over Tuxedo, WebSphere MQ, and CORBA.
- SSL and TLS support.
- Security support for Tuxedo and WebSphere MQ.
- Container-based deployment with IONA's Orbix 6.0 or higher and Tuxedo 7.1 or higher.
- Session management.
- Location services.
- Load balancing.

CHAPTER 2

Deploying Artix Solutions: An Overview

Artix can be deployed in a number of ways depending on the complexity of your project and your system architecture.

In this chapter

This chapter includes the following sections:

Artix Deployment Modes

Overview

All Artix components have two basic deployment modes:

- Embedded mode
- Standalone mode

Embedded mode

Embedded mode links Artix functionality directly into an application. The application invokes Artix functions directly and locally. Embedded mode requires linking the application with Artix-generated stubs and skeletons.

Standalone mode

Standalone mode places Artix functionality outside of the application space and runs it as a separate process invoked as a service. In standalone mode, Artix is completely described by an Artix contract that specifies which services are to be connected, what transports are in use, and how the services are linked.

These deployment modes can be combined in a number of ways to fit the needs of your applications and environment. For example, you can deploy the Artix session manager and an Artix router in standalone mode while embedding the Artix bus in your client and server applications. Or you could embed all of the Artix components into a server application.

Embedded Application

Overview

The most basic deployment of Artix is an application with Artix embedded inside. In this scenario, the application can use one of the transports supported by Artix, and the Artix bus is deployed within the application itself. The application gets all of its configuration information from the Artix configuration file and the Artix contract describing the applications interface.

Embedded configuration

Figure 2 shows an application with Artix embedded in it. Artix retrieves its configuration information from two places.

First, when the application first initializes the Artix bus, Artix pulls information about what plug-ins need to be loaded and other runtime information from one of the scopes in the Artix configuration file. Then when the application then registers a servant or instantiates a proxy object, Artix reads in the transport information from the Artix contract describing the application's interfaces.

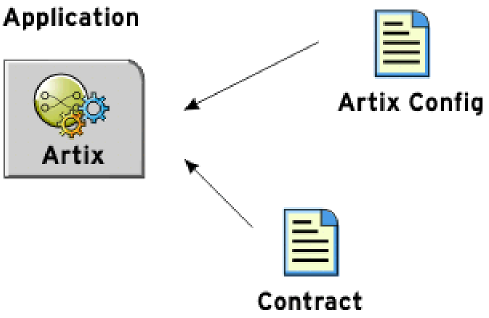

Figure 2: *Embedded Artix Deployment*

Why use this pattern

This pattern is the most common deployment pattern when deploying a Web service or developing a client application that needs to access a back-end server running on one of the transports Artix supports. Its simplicity makes it easy to configure and deploy. Also, its configuration can be easily modified.

Deploying a simple embedded application

To deploy an application with Artix embedded in it, you would do the following:

1. Ensure that the system's runtime environment has been properly set up to run Artix applications.
2. Optionally, you can edit the Artix configuration file to create a custom configuration scope for the application. This enables you to control which plug-ins are loaded, the logging level and location, and other runtime features of the bus.
3. Edit the Artix contract for the application to ensure that the transport details are correct for your system.
4. Place the application's contract in the directory where the application will look for it. Typically, this is the same directory as the executable.
5. Run the application with the correct command line arguments.

Standalone Switching Service

Overview

When using Artix as a bridge between applications running on different transports, Artix will often be deployed as a standalone switching service, as shown in Figure 3. When deployed in this scenario, Artix will use at least two transports and route between two or more applications.

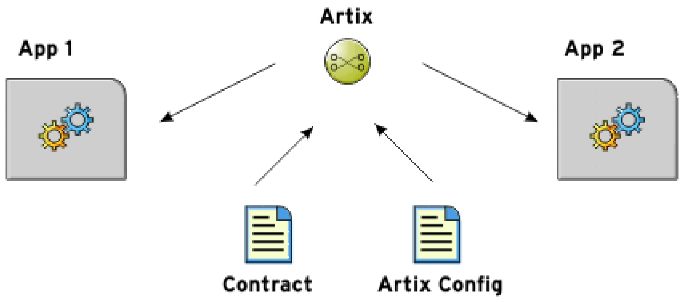

Figure 3: *Standalone Artix Deployment*

Standalone configuration

Similar to when Artix is deployed as an embedded piece of an application, Artix loads its configuration from two places. The bus gets its runtime configuration from the Artix configuration file. The transports get their configuration information from the Artix contract describing the interfaces being integrated.

However, both the configuration information and the contract describing the interfaces are more complicated. Artix needs to load more plug-ins to act as a switch. The routing plug-in also needs to be configured to load a contract with the required routing rules, and a control process will need to be configured to ensure that the switch can be shutdown gracefully. The Artix contract will have multiple transport configuration and routing information about how messages are passed through the switch.

Deploying a standalone switch

Artix is configured for the standalone switching service by default. To deploy Artix as a standalone switching service, do the following:

1. Ensure that the system's runtime environment has been properly set up to run Artix applications. See "Establishing the Host Computer Environment" on page 24.
2. If you are using the standalone service as a router, you must add the routing plug-in to the `orb_plugins` list, and configure the location of the WSDL used by the router.
3. Ensure that the Artix contract that describes the your integration contains the correct and routing extensor details (for example, the routing source and destination).
4. Place your application's contract in the directory where the application looks for it. Alternatively, your configuration must specify the location of the router's WSDL relative to where you are running the router.
5. Run the standalone Artix service.

For more detailed information, see Chapter 7.

Artix Locator

Overview

The Artix locator can be deployed as either a standalone service or an embedded service. The locator differs from Artix applications in that it does not redirect messages and it has a predefined contract.

Standalone locator

Figure 4 shows how system using the Artix locator in standalone mode would look. The locator uses its own contract to configure and advertise on which port it can be contacted. Both the application and the Artix service share a common Artix configuration file. However, they do not share a configuration scope. This style of deploying the locator is beneficial because it does not place additional load on the application. It is best suited for locators that service a number of server processes.

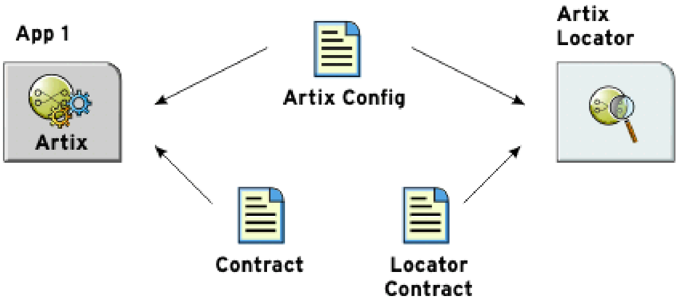

Figure 4: *Standalone Artix Locator*

Embedded locator

Figure 5 shows a system in which the Artix locator is embedded in an application. The application still requires two contracts. One for the application and one for the locator. However, when the Artix locator is embedded within an application the application and the locator share a configuration scope.

This style of deployment limits the number of separate processes that need to be deployed on a system. It is useful when the locator instance is only going to be servicing the application that it is embedded in.

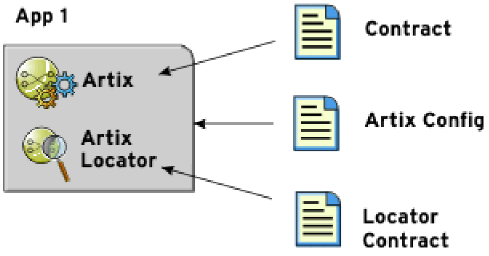

Figure 5: *Embedded Artix Locator*

Deploying a standalone Artix locator

To deploy a standalone Artix locator, complete the following steps:

1. Build a standalone Artix locator. This is discussed in *Developing Artix Applications with C++*.
2. Edit your Artix configuration file to include a configuration scope for your standalone locator.
3. In the locator's configuration scope, ensure that the locator loads the required plug-ins.
4. In the locator's configuration scope, specify the location of the contract for this instance of the locator service.

These steps are discussed in more detail in "Using the Artix Locator" on page 125.

Deploying the Artix locator embedded in an application

To deploy the Artix locator embedded in an application, complete the following steps:

1. Edit your application's configuration scope to specify that the locator plug-ins are loaded at runtime.
2. In the application's configuration scope, specify the location of the contract for locator service instance used by this application.

These steps are discussed in more detail in "Using the Artix Locator" on page 125.

Artix Session Manager

Overview

The Artix session manager enables Web services to engage in statefull communication. It can be deployed as either a standalone service or an embedded service. The session manager, like the Artix locator, has a predefined contract and service specific configuration information.

Standalone session manager

Figure 4 shows a system using the Artix session manager in standalone mode. The session manager uses its own contract to configure and advertise how it can be contacted and how its interface is configured.

In addition to the standalone session manager, your application loads an endpoint manager plug-in which also requires a contract defining the interface between the application and the session manager. Both the application and the session manager share a common Artix configuration file.

The standalone session manager instance and the application have separate configuration scopes. However, the configuration information for the endpoint manager is placed in the application's configuration scope. This style of deploying the session manager is best suited for scenarios where the session manager manages a number of endpoints.

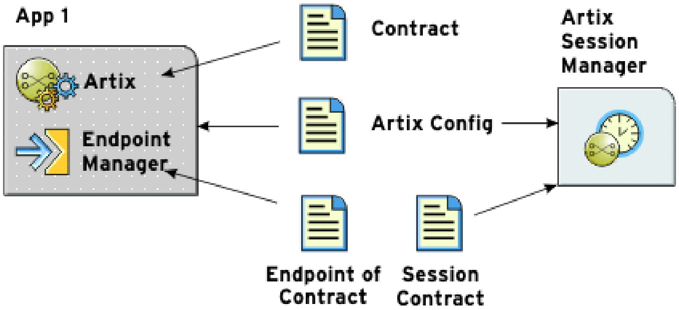

Figure 6: *Standalone Artix Session Manager*

Embedded session manager

Figure 5 shows a system in which the Artix session manager is embedded within an application. The application still requires three contracts. One for the application, one for the session manager, and one for the endpoint manager. However, when the Artix session manager is embedded within an application, the application and the session manager share a configuration scope. This style of deployment limits the number of separate processes that need to be deployed on a system and is useful when the session manger is only servicing the application in which it is embedded.

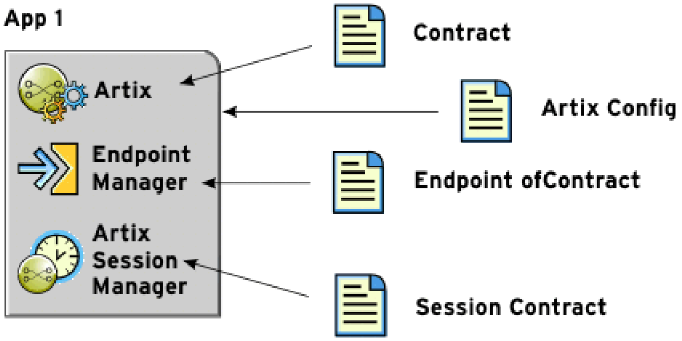

Figure 7: *Embedded Artix Session Manager*

Deploying a standalone Artix session manager

To deploy a standalone Artix session manager, complete the following steps:

1. Build a standalone Artix session manager. This is discussed in *Developing Artix Applications with C++*.
2. Edit your Artix configuration to include a configuration scope for your standalone session manager.
3. In the session manager's configuration scope, ensure that the session manager loads the required plug-ins.
4. In the session manager's configuration scope, specify the location of the contract for this instance of the session management service.

5. In the application's configuration scope, edit the orb_plugins list to include the required plug-ins for the endpoint manager.
6. In the application's configuration scope, specify the location of the contract for this instance of the endpoint manager.

These steps are discussed in more detail in "Using the Artix Session Manager" on page 141.

Deploying an embedded Artix session manager

To deploy the Artix session manager embedded in an application, complete the following steps:

1. Edit your application's configuration scope to specify that the session manager's plug-ins are loaded at runtime.
2. Edit your application's configuration scope to specify that the endpoint manager's plug-ins are loaded at runtime.
3. In the application's configuration scope, specify the location of the contract for session manager instance used by this application.
4. In the application's configuration scope, specify the location of the contract for this instance of the endpoint manager.

These steps are discussed in more detail in "Using the Artix Session Manager" on page 141.

Part II

Artix Configuration and Management

In this part

This part contains the following chapters:

CHAPTER 3

Configuring Artix

Artix's runtime configuration provides a great deal of control over how Artix systems perform. This chapter explains how to set up your system environment, and provides an overview of Artix runtime configuration.

In this chapter

This chapter discusses the following topics:

Establishing the Host Computer Environment

Overview

To use the Artix design tools and runtime environment, the host computer must have several IONA-specific environment variables set. These variables can be configured during installation, or later using the artix_env script, or configured manually.

This section includes the following:

- Running the artix_env script.
- Artix Environment variables.
- Customizing the artix_env script.

Running the artix_env script

The Artix installation process creates a script named artix_env, which captures the information required to set your host's environment variables. Running this script configures your system to use Artix. The script is located in the Artix bin directory:

```
IT_PRODUCT_DIR\artix\version\bin\artix_env
```

The artix_env script takes the following optional arguments:

Table 1: *Options to artix_env Script*

Option	Description
`-compiler vc71`	On Windows, enables support for Microsoft Visual C++ version 7.1 (Visual Studio .NET 2003). By default, Artix is enabled with support for Microsoft Visual C++ version 6.0.

Table 1: *Options to artix_env Script*

Option	Description
-preserve	Preserves the settings of any environment variables that have already been set. When this argument is specified, artix_env does not overwrite the values of variables that are already set. This option applies to the following environment variables: IT_PRODUCT_DIR IT_LICENSE_FILE IT_CONFIG_DIR IT_CONFIG_DOMAINS_DIR IT_DOMAIN_NAME IT_ART_ADMIN_PATH IT_IDL_CONFIG_FILE CLASSPATH PATH LIBPATH (AIX) LD_LIBRARY_PATH (Solaris, Linux) LD_PRELOAD (Linux) SHLIB_PATH (HP-UX) For more detailed information, see "Artix Environment variables" on page 26. **Note:** Before using the -preserve option, always ensure that the existing environment variable values are set correctly.
-verbose	artix_env outputs an audit trail of all its actions to stdout.

Artix Environment variables

This section describes the following environment variables in more detail:

- JAVA_HOME
- IT_PRODUCT_DIR
- IT_LICENSE_FILE
- IT_CONFIG_DIR
- IT_CONFIG_DOMAINS_DIR
- IT_DOMAIN_NAME
- IT_IDL_CONFIG_FILE
- IT_ART_ADMIN_PATH
- PATH

Note: You do not have to manually set your environment variables. You can configure them during installation, or set them later by running the provided `artix_env` script.

The environment variables are explained in Table 2:

Table 2: *Artix Environment Variables*

Variable	Description
JAVA_HOME	The directory path to your system's JDK is specified with the system environment variable JAVA_HOME. This must be set to use the Artix Designer GUI. This defaults to the JVM installed with Artix (`IT_PRODUCT_DIR\jre`). The Artix installer also enables you to specify a previously installed JVM.

Table 2: *Artix Environment Variables*

Variable	Description
IT_PRODUCT_DIR	IT_PRODUCT_DIR points to the top level of your IONA product installation. For example, on Windows, if you install Artix into the C:\Program Files\IONA directory, IT_PRODUCT_DIR should be set to that directory. **Note:** If you have other IONA products installed and you choose not to install them into the same directory tree, you must reset IT_PRODUCT_DIR each time you switch IONA products. You can override this variable using the -ORBproduct_dir command-line parameter when running your Artix applications.
IT_LICENSE_FILE	IT_LICENSE_FILE specifies the location of your Artix license file. The default value is IT_PRODUCT_DIR\etc\licenses.txt.
IT_CONFIG_DIR	IT_CONFIG_DIR specifies the root configuration directory. The default root configuration directory on UNIX is /etc/opt/iona, and IT_PRODUCT_DIR\artix\version\etc on Windows. You can override this variable using the -ORBconfig_dir command-line parameter.
IT_CONFIG_DOMAINS_DIR	IT_CONFIG_DOMAINS_DIR specifies the directory where Artix searches for its configuration files. The configuration domain's directory defaults to IT_CONFIG_DIR\domains. You can override it using the -ORBconfig_domains_dir command-line parameter.

Table 2: *Artix Environment Variables*

Variable	Description
IT_DOMAIN_NAME	IT_DOMAIN_NAME specifies the name of the configuration domain used by Artix to locate its configuration. This variable also specifies the name of the file in which the configuration is stored. For example, the artix domain is stored in IT_CONFIG_DIR\domains\artix.cfg. You can override this variable with the -ORBdomain_name command-line parameter.
IT_IDL_CONFIG_FILE	IT_IDL_CONFIG_FILE specifies the configuration used by the Artix IDL compiler. If this variable is not set, you will be unable to run the IDL to WSDL tools provided with Artix. This variable is required for an Artix Devopment installation.The default location is: IT_PRODUCT_DIR\artix\version\etc\idl.cfg. **Note:** Do not modify the default IDL configuration file.
IT_ART_ADMIN_PATH	IT_ART_ADMIN_PATH specifies the location of an internal configuration script used by administration tools. Defaults to IT_CONFIG_DIR\admin.

Table 2:*Artix Environment Variables*

Variable	Description

PATH	The Artix `bin` directories are prepended on the `PATH` to ensure that the proper libraries, configuration files, and utility programs (for example, the IDL compiler) are used. These settings avoid problems that might otherwise occur if Orbix and/or Tuxedo (both include IDL compilers and CORBA class libraries) are installed on the same host computer.
	The default Artix `bin` directory is:
	UNIX
	`$IT_PRODUCT_DIR/artix/`*version*`/bin`
	Windows
	`%IT_PRODUCT_DIR%\artix\`*version*`\bin`
	`%IT_PRODUCT_DIR%\bin`

Customizing the artix_env script

The `artix_env` script sets the Artix environment variables using values obtained from the Artix installer and from the script's command-line options. It checks each one of these settings in sequence, and updates them, where appropriate. The `artix_env` script is designed to suit most needs. However, if you want to customize it for your own purposes, please note the following points.

Before you begin

You can only run the `artix_env` script once in any console session. If you run this script a second time, it exits without completing. This prevents your environment from becoming bloated with duplicate information (for example, on your `PATH` and `CLASSPATH`).

In addition, if you introduce any errors when customizing the `artix_env` script, it also exits without completing. This feature is controlled by the `IT_ARTIXENV` variable, which is local to the `artix_env` script. `IT_ARTIXENV` is set to `true` the first time you run the script in a console; this causes the script to exit when run again.

Environment variables

- The JAVA_HOME environment variable defaults to the value obtained from the Artix installer. If you do not manually set this variable before running artix_env, it takes its value from the installer. The default location for the JRE supplied with Artix is IT_PRODUCT_DIR\jre.
- The following environment variables are all set with default values relative to IT_PRODUCT_DIR:
    - JAVA_HOME
    - IT_CONFIG_FILE
    - IT_IDL_CONFIG_FILE
    - IT_CONFIG_DIR
    - IT_CONFIG_DOMAINS_DIR
    - IT_LICENSE_FILE
    - IT_ART_ADMIN_PATH

  If you do not set these variables manually, artix_env sets them with default values based on IT_PRODUCT_DIR. For example, the default for IT_CONFIG_DIR on Windows is IT_PRODUCT_DIR\etc.
- The IT_IDL_CONFIG_FILE environment variable is a required only for an Artix Development installation. All other environment variables are required for both Development and Runtime installations.
- Before artix_env sets each environment variable, it checks if the -preserve command-line option was supplied when the script was run. This ensures that your preset values are not overwritten. Before using the -preserve option, always check the existing values for these variables are set correctly.

Artix Runtime Configuration

Overview

Artix is built upon IONA's Adaptive Runtime architecture (ART). Runtime behaviors are established through common and application-specific configuration settings that are applied during application startup. As a result, the same application code may be run—and may exhibit different capabilities—in different configuration environments. This section includes the following:

- Configuration domains.
- Configuration scopes.
- Specifying configuration scopes.
- Configuration namespaces.
- Configuration variables.
- Configuration data types.

Configuration domains

An Artix *configuration domain* is a collection of configuration information in an Artix runtime environment. This information consists of configuration variables and their values. A default Artix configuration is provided when Artix is installed. The default configuration file is located in:

Windows	`%IT_PRODUCT_DIR%\artix\2.1\etc\domains\artix.cfg`
UNIX	`$IT_PRODUCT_DIR/artix/2.1/etc/domains/artix.cfg`

The contents of this file may need to be changed to modify Artix logging, routing, and other behaviors.

You can also manually create new Artix configuration domains to compartmentalize your applications. These new configuration domains can import information from other configuration domains using a `#include` statement in your configuration. This provides a convenient way of compartmentalizing your application specific configuration from the global ART configuration information contained in the default domain.

Configuration scopes

An Artix configuration domain is subdivided into *configuration scopes*. These are typically organized into a hierarchy of scopes, whose fully-qualified names map directly to ORB names. By organizing configuration variables into various scopes, you can provide different settings for individual services, or common settings for groups of services.

Configuration scopes apply to a subset of services or to a specific service in an environment. Instances of the Artix standalone service can each have their own configuration scopes. A default Artix standalone service scope is automatically created when you install Artix.

Application-specific configuration variables either override default values assigned to common configuration variables, or establish new configuration variables. Configuration scopes are localized through a name tag and delimited by a set of curly braces terminated with a semicolon, for example, `(` *`scopeNameTag`* `{…}; )`.

A configuration scope may include nested configuration scopes. Configuration variables set within nested configuration scopes take precedence over values set in enclosing configuration scopes.

In the `artix.cfg` file, there are several predefined configuration scopes. For example, the `demo` configuration scope includes nested configuration scopes for some of the demo programs included with the product.

Example 1: *Demo Configuration Scope*

```
demo
{
 fml_plugin
 {
     orb_plugins = ["local_log_stream", "iiop_profile",
            "giop", "iiop", "soap", "http", "G2", "tunnel",
            "mq", "ws_orb", "fml"];
 };
```

Example 1: *Demo Configuration Scope*

```
 telco
 {
    orb_plugins = ["local_log_stream", "iiop_profile",
                   "giop”, "iiop”, "G2", "tunnel"];
    plugins:tunnel:iiop:port = "55002";
    poa:MyTunnel:direct_persistent = "true";
    poa:MyTunnel:well_known_address = "plugins:tunnel";

    server
      {
        orb_plugins = ["local_log_stream", "iiop_profile",
                      "giop", "iiop”, "ots", "soap", "http", "G2:,
                        "tunnel"];
        plugins:tunnel:poa_name = "MyTunnel";
     };
 };
 tibrv
 {
     orb_plugins = ["local_log_stream", "iiop_profile",
                    "giop", "iiop", "soap", "http", "tibrv"];

     event_log:filters = ["*=FATAL+ERROR"];
 };
};
```

Note: The `orb_plugins` list is redefined within each configuration scope.

Specifying configuration scopes

To make an Artix process run under a particular configuration scope, you specify that scope using the `-ORBname` parameter. Configuration scope names are specified using the following format

scope.subscope

For example, the scope for the telco server demo shown in Example 1 is specified as `demo.telco.server`. During process initialization, Artix searches for a configuration scope with the same name as the `-ORBname` parameter.

There are two ways of supplying the `-ORBname` parameter to an Artix process:

- Pass the argument on the command line.
- Specify the ORBname as the third parameter to `IT_Bus::init()`.

For example, to start an Artix process using the configuration specified in the `demo.tibrv` scope, you could start the process use the following syntax:

```
<processName> [application parameters] -ORBname demo.tibrv
```

Alternately, you could use the following code fragment to initialize the Artix bus:

```
IT_Bus::init (argc, argv, “demo.tibrv”);
```

If a corresponding scope is not located, the process starts under the highest level scope that matches the specified scope name. If there are no scopes that correspond to the `ORBname` parameter, the Artix process runs under the default global scope. For example, if the nested `tibrv` scope does not exist, the Artix process uses the configuration specified in the `demo` scope; if the `demo` scope does not exist, the process runs under the default global scope.

Configuration namespaces

Most configuration variables are organized within namespaces, which group related variables. Namespaces can be nested, and are delimited by colons (`:`). For example, configuration variables that control the behavior of a plug-in begin with `plugins:` followed by the name of the plug-in for which the variable is being set. For example, to specify the port on which the Artix standalone service starts, set the following variable:

```
plugins:artix_service:iiop:port
```

To set the location of the routing plug-in’s contract, set the following variable:

```
plugins:routing:wsdl_url
```

Configuration variables

Configuration data is stored in variables that are defined within each namespace. In some instances, variables in different namespaces share the same variable names.

Variables can also be reset several times within successive layers of a configuration scope. Configuration variables set in narrower configuration scopes override variable settings in wider scopes. For example, a `company.operations.orb_plugins` variable would override a `company.orb_plugins` variable. Plug-ins specified at the `company` scope would apply to all processes in that scope, except those processes that belong specifically to the `company.operations` scope and its child scopes.

Configuration data types

Each configuration variable has an associated data type that determines the variable's value.

Data types can be categorized into two types:

- Primitive types
- Constructed types

Primitive types

There are three primitive types: `boolean`, `double`, and `long`,.

Constructed types

Artix supports two constructed types: `string` and `ConfigList` (a sequence of strings).

- In an Artix configuration file, the `string` character set is ASCII.
- The `ConfigList` type is simply a sequence of `string` types. For example:

```
orb_plugins = ["local_log_stream", "iiop_profile",
    "giop","iiop"];
```

Artix Configuration Files

Overview

This section explains how to use Artix configuration files to manage applications in your environment. It includes the following:

- "Default configuration file".
- "Importing configuration settings".
- "Working with multiple installations".

Default configuration file

The Artix configuration domain file contains all the configuration settings for the domain. These settings can be changed to modify different aspects of Artix behavior (for example, logging or routing). The default Artix configuration domain file is found in the following location:

Windows	`%IT_PRODUCT_DIR%\artix\2.1\etc\domains\artix.cfg`
UNIX	`$IT_PRODUCT_DIR/artix/2.1/etc/domains/artix.cfg`

Importing configuration settings

You can manually create new Artix configuration domain files to compartmentalize your applications. These new configuration domain files can import information from other configuration domains using an `include` statement in your configuration file. This provides a convenient way of compartmentalizing your application-specific configuration from the global ART configuration information that is contained in the default configuration domain file.

Example 2 shows an `include` statement that imports the default configuration file. The include statement is typically the first line the configuration file.

Example 2: *Configuration file include statement*

```
include "../../../../../etc/domains/artix.cfg";

my_app_config {
...
}
```

For complete working examples of Artix applications that use this import mechanism, see the configuration files provided with Artix demos. These demo applications are available from the following directory:

```
<install-dir>\artix\version\demos
```

Working with multiple installations

If you are using multiple installations or versions of Artix, you can use your configuration files to help manage your applications as follows:

1. Install each version of Artix into a different directory.
2. Install your applications into their own directory.
3. Copy the `artix.cfg` file from whichever Artix release you want to use into another directory (for example, an application directory).
4. In your application's local configuration file, include the `artix.cfg` file from your copy location.

This enables you to switch between Artix versions by copying the corresponding `artix.cfg` file into a common location. This avoids having to update the directory information in your configuration file whenever you want to switch between Artix versions.

CHAPTER 4

Artix Logging

This chapter describes how to configure Artix logging. It also explains Artix support for Java log4j and SNMP (Simple Network Management Protocol).

In this chapter

This chapter includes the following sections:

Configuring Artix Logging

Overview

Logging in Artix is controlled by the `event_log:filters` configuration variable and by the log stream plug-ins (for example, `xmlfile_log_stream` and `local_log_stream`).

This section explains how to use these settings to configure logging in Artix. It includes the following:

- "Setting the event log filter".
- "Setting log stream plug-ins".
- "Using a rolling log file".

Setting the event log filter

The `event_log:filters` configuration variable can be set to provide a wide range of logging levels. You can set this variable in your Artix configuration file:

install-dir`\artix\`*version*`\etc\domains\artix.cfg`.

Displaying errors

The default `event_log:filters` setting displays errors only:

```
event_log:filters = ["*=FATAL+ERROR"];
```

Displaying warnings

The following setting displays errors and warnings only:

```
event_log:filters = ["*=FATAL+ERROR+WARNING"];
```

Displaying request/reply messages

Adding `INFO_MED` causes all of request/reply messages to be logged (for all transport buffers):

```
event_log:filters = ["*=FATAL+ERROR+WARNING+INFO_MED"];
```

Displaying trace output

The following setting displays typical trace statement output (without the raw transport buffers being printed):

```
event_log:filters = ["*=FATAL+ERROR+WARNING+INFO_HI"];
```

Displaying all logging

The following setting displays all logging:

```
event_log:filters = ["*=*"];
```

The default configuration settings enable logging of only serious errors and warnings. For more exhaustive information, select a different filter list at the default scope, or include a more expansive `event_log:filters` setting in your configuration scope.

Setting log stream plug-ins

In addition to setting the event log filter, you should configure the log stream plug-ins your `artix.cfg` file, for example:

```
//Ensure these two plug-ins exist in your orb_plugins list
orb_plugins = ["local_log_stream", "xmlfile_log_stream", ... ];

// Required name of plugin library
plugins:xmlfile_log_stream:shlib_name = "it_xmlfile";

// Optional filename (defaults to it_bus.log):
plugins:xmlfile_log_stream:filename = "artix_logfile.xml";

// Optional max size (defaults to 2MB
plugins:xmlfile_log_stream:max_file_size = "100000";

// Optional process identifier (defaults to false)
plugins:xmlfile_log_stream:use_pid = "false";
```

You must ensure that your application can detect the configuration settings for the log stream plug-in. You can either set them at the global scope, or configure a unique scope for use by your application, for example:

```
IT_Bus::init(argc, argv, "demo.myscope");
```

This enables you to place the necessary configuration in the demo.myscope scope.

Note: The xmlfile_log_stream plug-in is included in the default orb_plugins list, but not in the orb_plugins lists in many demo configuration scopes. To enable logging to an XML file for the applications that you develop, include this plug-in your orb_plugins list.

Using a rolling log file

You can specify that the logging plug-in uses a rolling file to prevent the local log from growing indefinitely. In this model, the stream appends the current date to the configured filename. This produces a complete filename, for example:

```
/var/adm/art.log.02171999
```

A new file begins with the first event of the day and ends at 23:59:59 each day. The default behavior is true. To disable rolling file behavior, set the following configuration variable to false:

```
plugins:xmlfile_log_stream:rolling_file = "false";
```

Configuring Log4J Logging

Overview

For Artix Java applications, you also have the option of using log4J, which is the standard Java logging tool. This enables you to control Artix logging with the same logging tool used by Java applications. This section includes the following:

- "Specifying the log4j plug-in".
- "Setting the log4 properties file".

Note: log4j logging overrides Artix logging. Settings in the `Log4jConfig.properties` file completely override settings in the `artix.cfg` file.

Specifying the log4j plug-in

You must first add the `log4j_log_stream` plug-in to your Artix `orb_plugins` list. For example:

```
orb_plugins = ["log4j_log_stream", "iiop_profile", "giop",
   "iiop"];
```

The `log4j_log_stream` plug-in reroutes all Artix logging to log4j.

Setting the log4 properties file

When using log4j with Artix, the `Log4jConfig.properties` file controls your Artix logging settings. This file is located in the following directory:

`install_dir``/artix/`*`version`*`/etc`

To enable log4j logging, delete the comment symbol (#) in the following line:

```
#log4j.logger.com.iona=DEBUG
```

In this file, all Artix logging is set to a root logger named `com.iona`. In addition, you cannot specify to log only `DEBUG` level messages like you can Artix logging. Instead, specifying a logging level means to log all messages with that level or higher. For example, setting the log level to `DEBUG` means to log all `DEBUG`, `WARNING`, `ERROR`, and `FATAL` messages.

Using log4j with your Java applications

If you wish to combine the log4J logging in your Java application with log4j logging in Artix, you must initialize log4j with the `LogConfig.properties` file in your Java application code.

However, you can still use your own properties file to initialize log4j, and you do no have to use `Log4jConfig.properties`.

Further information

For more information about using log4j, see the Apache documentation at:

`http://logging.apache.org/log4j/docs/documentation.html`

Configuring SNMP Logging

SNMP

*Simple Network Management Protocol (*SNMP) is the Internet standard protocol for managing nodes on an IP network. SNMP can be used to manage and monitor all sorts of equipment (for example, network servers, routers, bridges, and hubs).

The Artix SNMP `LogStream` plug-in uses the open source library `net-snmp` (v.5.0.7) to emit SNMP v1/v2 traps. For more information on this implementation, see http://sourceforge.net/projects/net-snmp/. To obtain a freeware SNMP Trap Receiver, visit http://www.ncomtech.com.

Artix Management Information Base (MIB)

A *MIB file* is a database of objects that can be managed using SNMP. It has a hierarchical structure, similar to a DOS or UNIX directory tree. It contains both pre-defined values and values that can be customized. The Artix MIB is shown below:

Example 3: *Artix MIB*

```
IONA-ARTIX-MIB DEFINITIONS  ::= BEGIN

 IMPORTS
        MODULE-IDENTITY, OBJECT-TYPE,
       Integer32, Counter32,
       Unsigned32,
       NOTIFICATION-TYPE              FROM   SNMPv2-SMI
       DisplayString                  FROM   RFC1213-MIB
;

-- v2 s/current/current


 iona OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) 3027 }

 ionaMib MODULE-IDENTITY
 LAST-UPDATED "200303210000Z"

 ORGANIZATION "IONA Technologies PLC"
```

Example 3: *Artix MIB*

```
CONTACT-INFO
            "
               Corporate Headquarters
               Dublin Office
               The IONA Building
               Shelbourne Road
               Ballsbridge
               Dublin 4 Ireland
               Phone: 353-1-662-5255
               Fax: 353-1-662-5244

               US Headquarters
               Waltham Office
               200 West Street 4th Floor
               Waltham, MA 02451
               Phone: 781-902-8000
               Fax: 781-902-8001

               Asia-Pacific Headquarters
               IONA Technologies Japan, Ltd
               Akasaka Sanchome Bldg.
               7F 3-21-16 Akasaka, Minato-ku,
               Tokyo, Japan 107-0052
               Tel: +81 3 3560 5611
               Fax: +81 3 3560 5612
               E-mail: support@iona.com
            "
DESCRIPTION
       "This MIB module defines the objects used and format of SNMP traps that are generated
        from the Event Log for Artix based systems from IONA Technologies"

::= { iona 1 }
```

Example 3: *Artix MIB*

```
--                      iona(3027)

--                        |
--                     ionaMib(1)
--                        |
--              ________________________________________
--              |                 |                    |
--           orbix3(2)       IONAAdmin (3)          Artix (4)
-                                                    |
--                                              --------------------
--                                              |                  |
--                               ArtixEventLogMibObjects(0)  ArtixEventLogMibTraps (1)
--                                   |                                   |
--              ----------------------------------------      -----------------------
--                              |- eventSource (1)                    |- ArtixbaseTrapDef (1)
--                              |- eventId (2)
--                              |- eventPriority (3)
--                              |- timeStamp (4)
--                              |- eventDescription (5)



 Artix                          OBJECT IDENTIFIER  ::= { ionaMib 4 }
 ArtixEventLogMibObjects        OBJECT IDENTIFIER  ::= { Artix 0 }
 ArtixEventLogMibTraps          OBJECT IDENTIFIER  ::= { Artix 1 }
 ArtixBaseTrapDef               OBJECT IDENTIFIER  ::= { ArtixEventLogMibTraps 1 }


-- MIB variables used as varbinds
 eventSource         OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "The component or subsystem which generated the event."
     ::= { ArtixEventLogMibObjects 1 }
```

Example 3: *Artix MIB*

```
 eventId        OBJECT-TYPE
    SYNTAX      INTEGER
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "The event id for the subsystem which generated the event."

    ::= { ArtixEventLogMibObjects 2 }

 eventPriority      OBJECT-TYPE
    SYNTAX          INTEGER
    MAX-ACCESS      not-accessible
    STATUS          current
    DESCRIPTION
        "The severity level of this event.  This maps to IT_Logging::EventPriority types.  All
         priority types map to four general types: INFO (I), WARN (W), ERROR (E), FATAL_ERROR (F)"

    ::= { ArtixEventLogMibObjects 3 }

 timeStamp       OBJECT-TYPE
    SYNTAX      DisplayString (SIZE(0..255))
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "The time when this event occurred."

    ::= { ArtixEventLogMibObjects 4 }

 eventDescription       OBJECT-TYPE
    SYNTAX           DisplayString (SIZE(0..255))
    MAX-ACCESS       not-accessible
    STATUS           current
    DESCRIPTION
        "The component/application description data included with event."

    ::= { ArtixEventLogMibObjects 5 }

-- SNMPv1 TRAP definitions
-- ArtixEventLogBaseTraps   TRAP-TYPE
--    OBJECTS {
--       eventSource,
--       eventId,
--       eventPriority,
```

Example 3: *Artix MIB*

```
--        timestamp,
--        eventDescription
--     }

--     STATUS current
--     ENTERPRISE iona
--     VARIABLES { ArtixEventLogMibObjects }
--     DESCRIPTION  "The generic trap generated from an Artix Event Log."
--     ::= { ArtixBaseTrapDef 1 }

-- SNMPv2 Notification type

 ArtixEventLogNotif   NOTIFICATION-TYPE
    OBJECTS {
       eventSource,
       eventId,
       eventPriority,
       timestamp,
       eventDescription
    }

    STATUS current
    ENTERPRISE iona
    DESCRIPTION  "The generic trap generated from an Artix Event Log."
    ::= { ArtixBaseTrapDef 1 }

END
```

IONA SNMP integration

Events received from various Artix components are converted into SNMP management information. This information is sent to designated hosts as SNMP traps, which can be received by any SNMP managers listening on the hosts. In this way, Artix enables SNMP managers to monitor Artix-based systems.

Artix supports SNMP version 1 and 2 traps only.

Artix provides a log stream plug-in called `snmp_log_stream`. The shared library name of the SNMP plug-in found in the `artix.cfg` file is:

```
plugins:snmp_log_stream:shlib_name = "it_snmp"
```

Configuring the SNMP plug-in

The SNMP plug-in has five configuration variables, whose defaults can be overridden by the user. The availability of these variables is subject to change. The variables and defaults are:

```
plugins:snmp_log_stream:community = "public";

plugins:snmp_log_stream:server    = "localhost";

plugins:snmp_log_stream:port      = "162";

plugins:snmp_log_stream:trap_type = "6";

plugins:snmp_log_stream:oid       = "<your IANA number in dotted decimal notation>"
```

Configuring the Enterprise Object Identifier

The last plug-in described, `oid`, is the Enterprise Object Identifier. This is assigned to specific enterprises by the Internet Assigned Numbers Authority (IANA). The first six numbers correspond to the prefix: `iso.org.dod.internet.private.enterprise` (`1.3.6.1.4.1`). Each enterprise is assigned a unique number, and can provide additional numbers to further specify the enterprise and product.

For example, the `oid` for IONA is `3027`. IONA has added `1.4.1.0` for Artix. Therefore the complete OID for IONA's Artix is `1.3.6.1.4.1.3027.1.4.1.0`. To find the number for your enterprise, visit the IANA website at http://www.iana.org.

The SNMP plug-in implements the `IT_Logging::LogStream` interface and therefore acts like the `local_log_stream` plug-in.

Using Artix TRACE Macros

Artix Trace levels

When the event log filter and log stream are properly configured, the Artix logging output from the TRACE macros is sent to the event log.

When using TRACE macros, the most important concept is the trace level, which is an enum that lets you filter events. Trace levels are defined in the *install-dir*/artix/*version*/include/it_bus/logging_support.h file:

```
const IT_TraceLevel IT_TRACE_FATAL = 64;           //FATAL
const IT_TraceLevel IT_TRACE_ERROR = 32;           //ERROR
const IT_TraceLevel IT_TRACE_WARNING = 16;         //WARNING
const IT_TraceLevel IT_TRACE = 4;                  //INFO_HIGH
const IT_TraceLevel IT_TRACE_BUFFER = 2;           //INFO_MED
const IT_TraceLevel IT_TRACE_METHODS = 1;          //INFO_LOW
const IT_TraceLevel IT_TRACE_METHODS_INTERNAL = 1; //INFO_LOW
```

The simplest trace statement emits a constant string at level IT_TRACE. For example:

```
TRACELOG("Hello world");
```

Passing in arguments

Several versions of the macro allow using a C printf format string, and passing in some arguments. Because you cannot have variable argument lists for macros, there are several defined according to how many arguments are allowed:

```
TRACELOG1("My name is: %s", "Slim Shady");
TRACELOG2("At state number %d, this happened: %s", 44, "connection failure");
```

Both the zero argument and the multi argument versions have a setting that allows a trace level to be passed in, instead of level IT_TRACE. For example:

```
TRACELOG_WITH_LEVEL(IT_METHODS, "MyClass::MyClass()");
TRACELOG_WITH_LEVEL1(IT_TRACE_METHODS_INTERNAL, "Value of my_name_field was %s", my_name_field);
```

Creating your own output

If you must create your own output using iostreams or another expensive process that is not supported by the macro, use the trace guard block. This ensures that the trace level test prevents your trace creation code from running when it does not produce output. For example:

```
BEGIN_TRACE(IT_TRACE)
        String trace_message = "data elements: ";
        for(i = 0; i < data_count; i++)
        {
                trace_message = trace_message + data_item[i] + "
   ";
        }
        TRACELOG(trace_message.c_str());
END_TRACE
```

To create binary output (for instance, a hex dump of the buffer), use TRACELOGBUFFER. For example:

```
TRACELOGBUFFER(vvMQMessageData, vvMQMessageData.GetSize())
```

If the trace statement issues at a level less than or equal to the process trace level, the entry is written to disk. The default log file name is it_bus.log.

Orbix TRACE Macros

Overview

The *install-dir*/artix/*version*/include/orbix/logging_support.h file defines the following Orbix-style logging macros:

- "IT_LOG_MESSAGE() Macro".
- "IT_LOG_MESSAGE_1() Macro".

IT_LOG_MESSAGE() Macro

```
// C++
#define IT_LOG_MESSAGE( \
    event_log,  \
    subsystem, \
    id, \
    severity, \
    desc \
) ...
```

A macro to use for reporting a log message.

Parameters

event_log	The log (EventLog) where the message is to be reported.
subsystem	The SubsystemId.
id	The EventId.
severity	The EventPriority.
desc	A string description of the event.

Examples

The following is a simple example of usage:

```
...
IT_LOG_MESSAGE(
    event_log,
    IT_IIOP_Logging::SUBSYSTEM,
    IT_IIOP_Logging::SOCKET_CREATE_FAILED,
    IT_Logging::LOG_ERROR,
    SOCKET_CREATE_FAILED_MSG
);
```

IT_LOG_MESSAGE_1() Macro

```
// C++
#define IT_LOG_MESSAGE_1( \
    event_log, \
    subsystem, \
    id, \
    severity,  \
    desc, \
    param0 \
) ...
```

A macro to use for reporting a log message with one event parameter.

Parameters

`event_log`	The log (`EventLog`) where the message is to be reported.
`subsystem`	The `SubsystemId`.
`id`	The `EventId`.
`severity`	The `EventPriority`.
`desc`	A string description of the event.
`param0`	A single parameter for an `EventParameters` sequence.

In addition, the `IT_LOG_MESSAGE_2()`, `IT_LOG_MESSAGE_3()`, `IT_LOG_MESSAGE_4()`, and `IT_LOG_MESSAGE_5()` macros, are provided for reporting log messages with two, three, four, and five parameters, respectively.

logging_support.h

Overview

This section shows the contents of an example `logging_support.h` file. This file is located in the following directory:

install-dir`/artix/`*version*`/include/it_bus/logging_support.h`

Example 4: *Artix logging_support.h*

```
#if !defined(_IT_BUS_LOGGING_)
#define _IT_BUS_LOGGING_
#include <stdio.h>
#include <stdarg.h>

#include <it_bus/API_Defines.h>

#define MAX_STACK_ALLOCATION 256
#define MAX_TRACE_SIZE 16384

typedef IT_UShort IT_TraceLevel;

//These are now equal to ART logging values, and are for backward compatibility.
                                                    //value to put in event_log:filters
const IT_TraceLevel IT_TRACE_FATAL = 64;            //FATAL
const IT_TraceLevel IT_TRACE_ERROR = 32;            //ERROR
const IT_TraceLevel IT_TRACE_WARNING = 16;          //WARNING
const IT_TraceLevel IT_TRACE = 4;                   //INFO_HIGH
const IT_TraceLevel IT_TRACE_BUFFER = 2;            //INFO_MED
const IT_TraceLevel IT_TRACE_METHODS = 1;           //INFO_LOW
const IT_TraceLevel IT_TRACE_METHODS_INTERNAL = 1;  //INFO_LOW

extern IT_AFC_API IT_TraceLevel g_log_filter;

namespace CORBA
{
class ORB;
};

namespace IT_Logging
{
    class EventLog;
}
```

Example 4: *Artix logging_support.h*

```
extern "C"
{
    void IT_AFC_API set_global_log_filter(IT_TraceLevel trace_level);
    void IT_AFC_API set_logging_default_ORB(CORBA::ORB* orb);

    void IT_AFC_API write_log_record(IT_Logging::EventLog* event_log, IT_TraceLevel trace_level,
   const char* description, ...);
    void IT_AFC_API write_log_record_with_CDATA(IT_Logging::EventLog* event_log, IT_TraceLevel
   trace_level, const char* description, const char* data_buffer, long buffer_size);
    void IT_AFC_API write_log_record_with_binary(IT_Logging::EventLog* event_log, IT_TraceLevel
   trace_level, const char* description, const char* data_buffer, long buffer_size);
}

//These are for writing data buffers. Binary buffers are written in a hex dump format.
//To see output from these, include INFO_MED in your event_log:filters.
#define IT_LOG_BUFFER(event_log, Entry, Length) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
       write_log_record_with_binary(event_log, IT_TRACE_BUFFER, "Buffer Output", Entry, Length);
   \
    }

#define IT_LOG_CDATA(event_log, description, Entry) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(event_log, IT_TRACE_BUFFER, description, Entry, 0); \
    }

#define IT_LOG_CDATA_SIZE(event_log, description, Entry, Size) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(event_log, IT_TRACE_BUFFER, description, Entry, Size); \
    }

#define IT_LOG_CDATA_BINARY_BUFFER(event_log, description, bbData) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_binary(event_log, IT_TRACE_BUFFER, description,
   bbData.get_const_pointer(), bbData.get_size()); \
    }
```

Example 4: *Artix logging_support.h*

```
//These are used for controlled tracing operations.  Description is a printf format string
//They allow specifying the trace level so callers can control visibility.
#define IT_LOG_GUARDED0(event_log, trace_level, description) \
    if ((g_log_filter & trace_level) != 0) \
        write_log_record(event_log, trace_level, description);

#define IT_LOG_GUARDED(event_log, trace_level, description) \
    IT_LOG_GUARDED0(event_log, trace_level, description)

#define IT_LOG_GUARDED1(event_log, trace_level, description, Arg1) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1); \
    }

#define IT_LOG_GUARDED2(event_log, trace_level, description, Arg1, Arg2) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1, Arg2); \
    }

#define IT_LOG_GUARDED3(event_log, trace_level, description, Arg1, Arg2, Arg3) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1, Arg2, Arg3); \
    }

#define IT_LOG_GUARDED4(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4) \
    if ((g_log_filter & trace_level) != 0) \
    { \
            write_log_record(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4); \
    }

#define IT_LOG_GUARDED5(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5) \
    if ((g_log_filter & trace_level) != 0) \
    { \
           write_log_record(event_log, trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5); \
    }
```

Example 4: *Artix logging_support.h*

```
//These are used to guard a code block from executing when the purpose of the code
//block is solely for formatting a trace statement.  It prevents the code from
//executing when the trace_level is filtered out and would not be used anyway.
#define BEGIN_TRACE(trace_level)                                         \
    if ((g_log_filter & trace_level) != 0)                                 \
    {

#define END_TRACE                                                                         \
    }


//All the macros that follow are just short hand for the previous ones, but they
//default the event_log to 0, which uses the first one that was loaded (usually
//the only one unless you are using multiple ORB names in your cfg file.

//These are for writing data buffers. Binary buffers are written in a hex dump format.
//To see output from these, include INFO_MED in your event_log:filters
#define TRACELOGBUFFER(Entry, Length) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_binary(0, IT_TRACE_BUFFER, "Buffer Output", Entry, Length); \
    }

#define TRACELOG_CDATA(description, Entry) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(0, IT_TRACE_BUFFER, description, Entry, 0); \
    }

#define TRACELOG_CDATA_SIZE(description, Entry, Size) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
        write_log_record_with_CDATA(0, IT_TRACE_BUFFER, description, Entry, Size); \
    }

#define TRACELOG_CDATA_BINARY_BUFFER(description, bbData) \
    if ((g_log_filter & IT_TRACE_BUFFER) != 0) \
    { \
       write_log_record_with_binary(0, IT_TRACE_BUFFER, description, bbData.get_const_pointer(),
   bbData.get_size()); \
    }
```

Example 4: *Artix logging_support.h*

```
//These are used for method level tracing.
//To see output from these, include INFO_LOW in your event_log:filters.
#define BEGIN_INTERNAL_METHOD(Name) \
    const char *FuncName = Name; \
    if ((g_log_filter & IT_TRACE_METHODS_INTERNAL) != 0) \
        write_log_record(0, IT_TRACE_METHODS_INTERNAL, FuncName);

#define END_INTERNAL_METHOD

#define BEGIN_METHOD(Name) \
    const char *FuncName = Name; \
    if ((g_log_filter & IT_TRACE_METHODS_INTERNAL) != 0) \
        write_log_record(0, IT_TRACE_METHODS, FuncName);

#define END_METHOD


//These are used for controlled tracing operations.  Description is a printf format string.
//They allow specifying the trace level so callers can control visibility.
#define TRACELOG_WITH_LEVEL0(trace_level, description) \
    IT_LOG_GUARDED(0, trace_level, description)

#define TRACELOG_WITH_LEVEL(trace_level, description) \
    IT_LOG_GUARDED(0, trace_level, description)

#define TRACELOG_WITH_LEVEL1(trace_level, description, Arg1) \
    IT_LOG_GUARDED1(0, trace_level, description, Arg1)

#define TRACELOG_WITH_LEVEL2(trace_level, description, Arg1, Arg2) \
    IT_LOG_GUARDED2(0, trace_level, description, Arg1, Arg2)

#define TRACELOG_WITH_LEVEL3(trace_level, description, Arg1, Arg2, Arg3) \
    IT_LOG_GUARDED3(0, trace_level, description, Arg1, Arg2, Arg3)

#define TRACELOG_WITH_LEVEL4(trace_level, description, Arg1, Arg2, Arg3, Arg4) \
    IT_LOG_GUARDED4(0, trace_level, description, Arg1, Arg2, Arg3, Arg4)

#define TRACELOG_WITH_LEVEL5(trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5) \
    IT_LOG_GUARDED5(0, trace_level, description, Arg1, Arg2, Arg3, Arg4, Arg5)
```

Example 4: *Artix logging_support.h*

```
//These are used for normal tracing operations.  Description is a printf format string.
//They default the trace level to IT_TRACE. To use another level, see the previous set.
#define TRACELOG(description) \
    IT_LOG_GUARDED(0, IT_TRACE, description)

#define TRACELOG0(description) \
    IT_LOG_GUARDED(0, IT_TRACE, description)

#define TRACELOG1(description, Arg1) \
    IT_LOG_GUARDED1(0, IT_TRACE, description, Arg1)

#define TRACELOG2(description, Arg1, Arg2) \
    IT_LOG_GUARDED2(0, IT_TRACE, description, Arg1, Arg2)

#define TRACELOG3(description, Arg1, Arg2, Arg3) \
    IT_LOG_GUARDED3(0, IT_TRACE, description, Arg1, Arg2, Arg3)

#define TRACELOG4(description, Arg1, Arg2, Arg3, Arg4) \
    IT_LOG_GUARDED4(0, IT_TRACE, description, Arg1, Arg2, Arg3, Arg4)

#define TRACELOG5(description, Arg1, Arg2, Arg3, Arg4, Arg5) \
    IT_LOG_GUARDED5(0, IT_TRACE, description, Arg1, Arg2, Arg3, Arg4, Arg5)

#endif
```

IT_Logging Module

Overview

The IT_Logging module is the centralized point for programmatic control of all logging. The LogStream interface controls how and where events are received.

The IT_Logging module also uses the following common data types, static method, and macros.

Table 3:*IT_Logging Common Data Types, Methods, and Macros*

Common Data Types	Methods and Macros

ApplicationId
EventId
EventParameters
EventPriority
SubsystemId
Timestamp
	format_message()
	
	IT_LOG_MESSAGE()
	IT_LOG_MESSAGE_1()
	IT_LOG_MESSAGE_2()
	IT_LOG_MESSAGE_3()
	IT_LOG_MESSAGE_4()
	IT_LOG_MESSAGE_5()

IT_Logging::ApplicationId Data Type

```
//IDL
typedef string ApplicationId;
```

An identifying string representing the application that logged the event.

For example, a UNIX and Windows ApplicationId contains the host name and process ID (PID) of the reporting process. Because this value can differ from platform to platform, streams should only use it as informational text, and should not attempt to interpret it.

IT_Logging::EventId Data Type

```
//IDL
typedef unsigned long EventId;
```

An identifier for the particular event.

IT_Logging::EventParameters Data Type

```
//IDL
typedef CORBA::AnySeq EventParameters;
```

A sequence of locale-independent parameters encoded as a sequence of `Any` values.

IT_Logging::EventPriority Data Type

```
//IDL
typedef unsigned short EventPriority;
```

Specifies the priority of a logged event. These can be divided into the following categories of priority.

Information A significant non-error event has occurred. Examples include server startup/shutdown, object creation/deletion, and information about administrative actions. Informational messages provide a history of events that can be invaluable in diagnosing problems.

Warning The subsystem has encountered an anomalous condition, but can ignore it and continue functioning. Examples include encountering an invalid parameter, but ignoring it in favor of a default value.

Error An error has occurred. The subsystem will attempt to recover, but may abandon the task at hand. Examples include finding a resource (such as memory) temporarily unavailable, or being unable to process a particular request due to errors in the request.

Fatal Error An unrecoverable error has occurred. The subsystem or process will terminate.

The possible values for an EventPriority consist of the following:

```
LOG_NO_EVENTS
LOG_ALL_EVENTS
LOG_INFO_LOW
LOG_INFO_MED
LOG_INFO_HIGH
LOG_INFO  (LOG_INFO_LOW)
LOG_ALL_INFO
LOG_WARNING
LOG_ERROR
LOG_FATAL_ERROR
```

A single value is used for EventLog operations that report events or LogStream operations that receive events. In filtering operations such as set_filter(), these values can be combined as a filter mask to control which events are logged at runtime.

IT_Logging::format_message()

```
// C++
static char* format_message(
    const char* description,
    const IT_Logging::EventParameters& params
);
```

Returns a formatted message based on a format description and a sequence of parameters.

Parameters

Messages are reported in two pieces for internationalization:

description	A locale-dependent string that describes of how to use the sequence of parameters in params.
params	A sequence of locale-dependent parameters.

format_message() copies the description into an output string, interprets each event parameter, and inserts the event parameters into the output string where appropriate. Event parameters that are primitive and SystemException parameters are converted to strings before insertion. For all other types, question marks (?) are inserted.

IT_Logging::SubsystemId Data Type

```
//IDL
typedef string SubsystemId;
```

An identifying string representing the subsystem from which the event originated. The constant `_DEFAULT` may be used to enable all subsystems.

IT_Logging::Timestamp Data Type

```
//IDL
typedef unsigned long  Timestamp;
```

The time of the logged event in seconds since January 1, 1970.

IT_Logging::LogStream Interface

Overview

Each of the Artix logging plug-ins implements the IT_Logging::LogStream interface. The LogStream interface allows an application to intercept events and write them to some concrete location via a stream.

IT_Logging::EventLog objects maintain a list of LogStream objects. You register a LogStream object from an EventLog using register_stream().

The complete LogStream interface is as follows:

```
// IDL in module IT_Logging
interface LogStream {
    void report_event(
        in ApplicationId   application,
        in SubsystemId     subsystem,
        in EventId         event,
        in EventPriority   priority,
        in Timestamp       event_time,
        in any             event_data
    );

    void report_message(
        in ApplicationId   application,
        in SubsystemId     subsystem,
        in EventId         event,
        in EventPriority   priority,
        in Timestamp       event_time,
        in string          description,
        in EventParameters parameters
    );
};
```

These operations are described in detail as follows:

LogStream::report_event()

```
// IDL
void report_event(
    in ApplicationId   application,
    in SubsystemId     subsystem,
    in EventId         event,
    in EventPriority   priority,
    in Timestamp       event_time,
    in any             event_data
);
```

Reports an event and its event-specific data to the log stream.

Parameters

`application`	An ID representing the reporting application.
`subsystem`	The name of the subsystem reporting the event.
`event`	A unique ID defining the event.
`priority`	The event priority.
`event_time`	The time when the event occurred.
`event_data`	Event-specific data.

See also

`IT_Logging::EventLog::report_event()`

`IT_Logging::LogStream::report_message()`

LogStream::report_message()

```
// IDL
void report_message(
    in ApplicationId   application,
    in SubsystemId     subsystem,
    in EventId         event,
    in EventPriority   priority,
    in Timestamp       event_time,
    in string          description,
    in EventParameters parameters
);
```

Reports an event and message to the log stream.

Parameters

`application`	An ID representing the reporting application.
`subsystem`	The name of the subsystem reporting the event.
`event`	The unique ID defining the event.
`priority`	The event priority.
`event_time`	The time when the event occurred.
`description`	A string describing the format of `parameters`.
`parameters`	A sequence of parameters for the log.

See also

`IT_Logging::EventLog::report_message()`

`IT_Logging::LogStream::report_event()`

CHAPTER 5

Enterprise Performance Logging

IONA's performance logging plug-ins enable Artix to integrate effectively with third-party Enterprise Management Systems (EMS).

In this chapter

This chapter contains the following sections:

Enterprise Management Integration

Overview

IONA's performance logging plug-ins enable both Artix and Orbix to integrate effectively with *Enterprise Management Systems* (EMS), such as IBM Tivoli™, HP OpenView™, CA Unicenter™, or BMC Patrol™. The performance logging plug-ins can also be used in isolation or as part of a bespoke solution.

Enterprise Management Systems enable system administrators and production operators to monitor enterprise-critical applications from a single management console. This enables them to quickly recognize the root cause of problems that may occur, and take remedial action (for example, if a machine is running out of disk space).

Performance logging

When performance logging is configured, you can see how each Artix server is responding to load. The performance logging plug-ins log this data to file or `syslog`. Your EMS (for example, IBM Tivoli) can read the performance data from these logs, and use it to initiate appropriate actions, (for example, issue a restart to a server that has become unresponsive, or start a new replica for an overloaded cluster).

Example EMS integration

Figure 8 shows an overview of the IONA and IBM Tivoli integration at work. In this example, a restart command is issued to an unresponsive server.

In Figure 8, the performance log files indicate a problem. The IONA Tivoli Provider uses the log file interpreter to read the logs. The provider sees when a threshold is exceeded and fires an event. The event causes a task to be activated in the Tivoli Task Library. This task restarts the appropriate server.

This chapter explains how to manually configure the performance logging plug-ins. It also explains the format of the performance logging messages.

For details on how to integrate your EMS environment with Artix, see the IONA guide for your EMS. For example, the *IONA Tivoli Integration Guide* or *IONA BMC Patrol Integration Guide*.

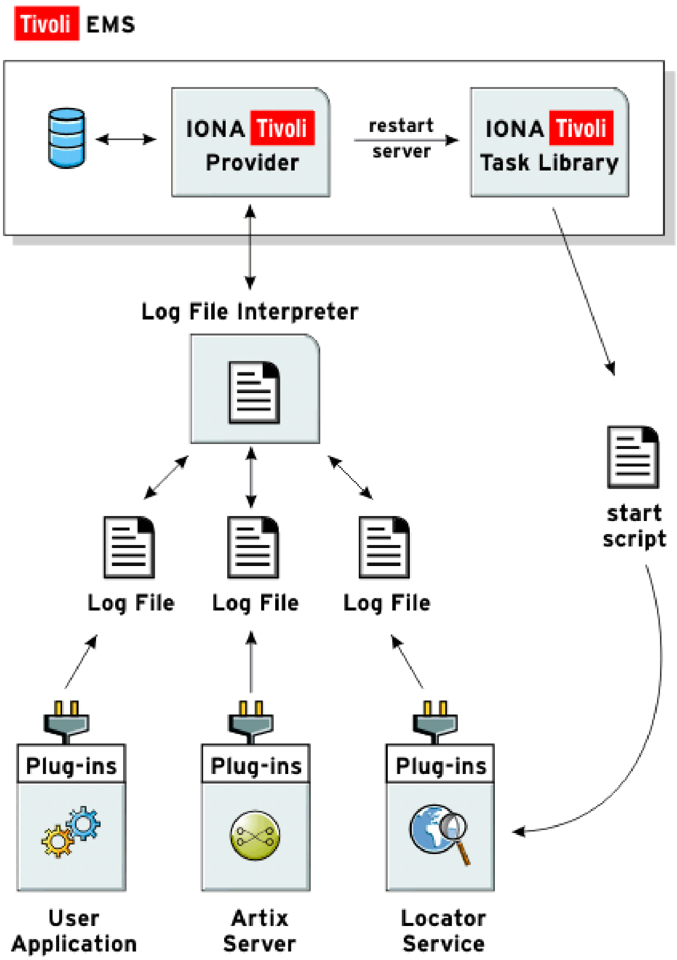

Figure 8: *Overview of an Artix and IBM Tivoli Integration*

Configuring Performance Logging

Overview

This section explains how to manually configure performance logging. This section includes the following:

- "Performance logging plug-ins".
- "Monitoring Artix requests".
- "Logging to a file or syslog".
- "Logging to a syslog daemon".
- "Monitoring clusters".
- "Configuring a server ID".
- "Configuring a client ID".
- "Configuring with the GUI".

Note: You can also use the **Artix Designer** GUI tool to configure performance logging automatically. However, manual configuration gives you more fine-grained control.

Performance logging plug-ins

The performance logging component includes the following plug-ins:

Table 4: *Performance Logging Plug-ins*

Plug-in	Description
Response monitor	Monitors response times of requests as they pass through the Artix binding chains. Performs the same function for Artix as the response time logger does for Orbix.
Collector	Periodically collects data from the response monitor plug-in and logs the results.

Monitoring Artix requests

You can use performance logging to monitor Artix server and client requests.

To monitor both client and server requests, add the bus_response_monitor plug-in to the orb_plugins list in the global configuration scope. For example:

```
orb_plugins = ["xmlfile_log_stream", "soap", "http",
   "bus_response_monitor"];
```

To configure performance logging on the client side only, specify this setting in a client scope only.

Logging to a file or syslog

You can configure the collector plug-in to log data either to a file or to syslog. The configuration settings depends on whether your application is written in C++ or Java.

C++ configuration

The following example configuration for a C++ application results in performance data being logged to /var/log/my_app/perf_logs/treasury_app.log every 90 seconds:

```
plugins:it_response_time_collector:period = "90";
plugins:it_response_time_collector:filename =
"/var/log/my_app/perf_logs/treasury_app.log";
```

If you do not specify the response time period, it defaults to 60 seconds.

Java configuration

Configuring the Java collector plug-in is slightly different from the C++ collector) because the Java collector plug-in makes use of Apache Log4J. Instead of setting plugins:it_response_time_collector:filename, you set the plugins:it_response_time_collector:log_properties to use Log4J, for example:

```
plugins:it_response_time_collector:log_properties = ["log4j.rootCategory=INFO, A1",
"log4j.appender.A1=com.iona.management.logging.log4jappender.TimeBasedRollingFileAppender",
"log4j.appender.A1.File="/var/log/my_app/perf_logs/treasury_app.log",
"log4j.appender.A1.MaxFileSize=512KB",
"log4j.appender.A1.layout=org.apache.log4j.PatternLayout",
"log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} %-80m %n"
];
```

Logging to a syslog daemon

You can configure the collector to log to a syslog daemon or Windows event log, as follows:

```
plugins:it_response_time_collector:system_logging_enabled = "true";
plugins:it_response_time_collector:syslog_appID = "treasury";
```

The `syslog_appid` enables you to specify your application name that is prepended to all syslog messages. If you do not specify this, it defaults to `iona`.

Monitoring clusters

You can configure your EMS to monitor a cluster of servers. You can do this by configuring multiple servers to log to the same file. If the servers are running on different hosts, the log file location must be on an NFS mounted or shared directory.

Alternatively, you can use `syslogd` as a mechanism for monitoring a cluster. You can do this by choosing one `syslogd` to act as the central logging server for the cluster. For example, say you decide to use a host named `teddy` as your central log server. You must edit the `/etc/syslog.conf` file on each host that is running a server replica, and add a line such as the following:

```
# Substitute the name of your log server
  user.info                                         @teddy
```

Some syslog daemons will not accept log messages from other hosts by default. In this case, it may be necessary to restart the `syslogd` on `teddy` with a special flag to allow remote log messages.

You should consult the `man` pages on your system to determine if this is necessary and what flags to use.

Configuring a server ID

You can configure a server ID that will be reported in your log messages. This server ID is particularly useful in the case where the server is a replica that forms part of a cluster.

In a cluster, the server ID enables management tools to recognize log messages from different replica instances. You can configure a server ID as follows:

```
plugins:it_response_time_collector:server-id = "Locator-1";
```

This setting is optional; and if omitted, the server ID defaults to the ORB name of the server. In a cluster, each replica must have this value set to a unique value to enable sensible analysis of the generated performance logs.

Configuring a client ID

You can also configure a client ID that will be reported in your log messages. Specify this using the `client-id` configuration variable, for example:

```
plugins:it_response_time_collector:client-id = "my_client_app";
```

This setting enables management tools to recognize log messages from client applications. This setting is optional; and if omitted, it is assumed that that a server is being monitored.

Configuration example

The following simple example configuration file is from the management demo supplied in your Artix installation:

```
include "../../../../../etc/domains/artix.cfg";

demos {

  management

  {

    orb_plugins = ["xmlfile_log_stream", "soap", "http",
                     "bus_response_monitor"];

    plugins:it_response_time_collector:period = "5";

   client {

    plugins:it_response_time_collector:client-id=
      "management-demo-client";

    plugins:it_response_time_collector:filename=
      "management_demo_client.log";
   };

   server {

     plugins:it_response_time_collector:server-id=
       "management-demo-server";

     plugins:it_response_time_collector:filename=
       "management_demo_server.log";
   };
  };
};
```

In this example, the `bus_response_monitor` plug-in and `plugins:it_response_time_collector:period` are set in the global scope. This specifies these settings for both the client and server applications.

Configuring with the GUI

The **Artix Designer** GUI tool automatically generates performance logging configuration for the Artix services. The generated `server-id` defaults to the following format:

`domain-name_service-name_hostname` (for example, `artix_locator_myhost`)

For details on how to automatically generate performance logging, see the *IONA Tivoli Integration Guide* or *IONA BMC Patrol Integration Guide*.

Logging Message Formats

Overview

This section describes the logging message formats used by IONA products. It includes the following:

- "Artix log message format".
- "Orbix log message format".
- "Simple life cycle message formats".

Artix log message format

Performance data is logged in a well-defined format. For Artix applications, this format is as follows:

```
YYYY-MM-DD HH:MM:SS server=serverID [namespace=nnn service=sss
   port=ppp operation=name] count=n avg=n max=n min=n int=n oph=n
```

Table 5: *Artix log message arguments*

Argument	Description
server	The server ID of the process that is logging the message.
namespace	The Artix namespace.
service	The Artix service.
port	The Artix port.
operation	The name of the operation for CORBA invocations or the URI for requests on servlets.
count	The number of operations of invoked (IIOP). or The number of times this operation or URI was logged during the last interval (HTTP).
avg	The average response time (milliseconds) for this operation or URI during the last interval.

Table 5: *Artix log message arguments*

Argument	Description
`max`	The longest response time (milliseconds) for this operation or URI during the last interval.
`min`	The shortest response time (milliseconds) for this operation or URI during the last interval.
`int`	The number of milliseconds taken to gather the statistics in this log file.
`oph`	Operations per hour.

The combination of namespace, service and port above denote a unique Artix endpoint.

Orbix log message format

The format for Orbix log messages is as follows:

```
YYYY-MM-DD HH:MM:SS server=serverID [operation=name] count=n
   avg=n max=n min=n int=n oph=n
```

Table 6: *Orbix log message arguments*

Argument	Description
`server`	The server ID of the process that is logging the message.
`operation`	The name of the operation for CORBA invocations or the URI for requests on servlets.
`count`	The number of operations of invoked (IIOP). or The number of times this operation or URI was logged during the last interval (HTTP).
`avg`	The average response time (milliseconds) for this operation or URI during the last interval.
`max`	The longest response time (milliseconds) for this operation or URI during the last interval.

Table 6: *Orbix log message arguments*

Argument	Description
`min`	The shortest response time (milliseconds) for this operation or URI during the last interval.
`int`	The number of milliseconds taken to gather the statistics in this log file.
`oph`	Operations per hour.

Simple life cycle message formats

The server will also log simple life cycle messages. All servers share the following common format.

```
YYYY-MM-DD HH:MM:SS server=serverID status=current_status
```

Table 7: *Simple life cycle message formats arguments*

Argument	Description
`server`	The server ID of the process that is logging the message.
`status`	A text string describing the last known status of the server (for example, `starting_up`, `running`, `shutting_down`).

CHAPTER 6

Using Artix with International Codesets

The Artix SOAP and CORBA bindings enable you to transmit and receive messages in a range of codesets.

In this chapter

This chapter includes the following:

Introduction to International Codesets

Overview

A *coded character set*, or *codeset* for short, is a mapping between integer values and characters that they represent. The best known codeset is ASCII (American Standard Code for Information Interchange). ASCII defines 94 graphic characters and 34 control characters using the 7-bit integer range.

European languages

The 94 characters defined by the ASCII codeset are sufficient for English, but they are not sufficient for European languages, such as French, Spanish, and German.

To remedy the situation, an 8-bit codeset, ISO 8859-1, also known as Latin-1, was invented. The lower 7-bit portion is identical to ASCII. The extra characters in the upper 8-bit range cover those languages used widely in Western Europe.

Many other codesets are defined under ISO 8859 framework. These cover languages in other regions of Europe, as well as Russian, Arabic and Hebrew. The most recent addition is ISO 8859-15, which is a revision of ISO 8859-1. This adds the Euro currency symbol and other letters while removing less used characters.

For further information about ISO-8859-x encoding, see the following web site: “The ISO 8859 Alphabet Soup” (http://wwwwbs.cs.tu-berlin.de/user/czyborra/charsets/).

Ideograms

Asian countries that use ideograms in their writing systems need more characters than fit in an 8-bit integer. Therefore, they invented double-byte codesets, where a character is represented by a bit pattern of 2 bytes.

These languages also needed to mix the double-byte codeset with ASCII in a single text file. So, *character encoding schemas*, or simply *encodings*, were invented as a way to mix characters of multiple codesets.

Some of the popular encodings used in Japan include:

- Shift JIS
- Japanese EUC
- Japanese ISO 2022

Unicode

Unicode is a new codeset that is gaining popularity. It aims to assign a unique number, or code point, to every character that exists (and even once existed) in all languages. To accomplish this, Unicode, which began as a double-byte codeset, has been expanded into a quadruple-byte codeset.

Unicode, in pure form, can be difficult to use within existing computer architectures, because many APIs are byte-oriented and assume that the byte value `0` means the end of the string.

For this reason, Unicode Transformation Format for 8-bit channel, or UTF-8, is frequently used. When browsers list “Unicode” in its encoding selection menu, they usually mean UTF-8, rather than the pure form of Unicode.

For more information about Unicode and its variants, visit Unicode (http://www.unicode.org/).

Charset names

To address the need for computer networks to connect different types of computers that use different encodings, the Internet Assigned Number Authority, or IANA, has a registry of encodings at http://www.iana.org/assignments/character-sets.

IANA names are used by many Internet standards including MIME, HTML, and XML.

Table 8 lists IANA names for some popular charsets.

Table 8: *IANA Charset Names*

IANA Name	Description
US-ASCII	7-bit ASCII for US English
ISO-8859-1	Western European languages
UTF-8	Byte oriented transformation of Unicode
UTF-16	Double-byte oriented transformation of Unicode
Shift_JIS	Japanese DOS & Windows
EUC-JP	Japanese adaptation of generic EUC scheme, used in UNIX

Table 8: *IANA Charset Names*

IANA Name	Description
ISO-2022-JP	Japanese adaptation of generic ISO 2022 encoding scheme

Note: IANA names are case insensitive. For example, US-ASCII can be spelled as us-ascii or US-ascii.

CORBA names

In CORBA, codesets are identified by numerical values registered with the Open Group's registry, OSF Codeset Registry: ftp://ftp.opengroup.org/pub/code_set_registry/code_set_registry1.2g.txt.

Java names

Java has its own names for charsets. For example, ISO-8859-1 is named `ISO8859_1`, Shift_JIS is named `SJIS`, and UTF-8 is named `UTF8`.

Java is transitioning to IANA charset names, to be aligned with MIME. JDK 1.3 and above recognizes both names.

Note: Artix uses IANA charset names even for CORBA codesets.

Working with Codesets using SOAP

Overview

Because SOAP messages are XML based, they are composed primarily of character data that can be encoded using any of the existing codesets. If the applications in a system are using different codesets, they can not interpret the messages passing between them. The Artix SOAP plug-in uses the XML prologue of SOAP messages to ensure that it stays in sync with the applications that it interacts with.

Making requests

When making requests or broadcasting a message, the SOAP plug-in determines the codeset to use from its Artix configuration scope. You can set the SOAP plug-in's character encoding using the `plugins:soap:encoding` configuration variable. This takes the IANA name of the desired codeset. The default value is `UTF-8`.

For more information on this configuration variable, see "SOAP Plug-in" on page 65. For general information on configuring Artix applications, see "Configuring Artix" on page 23.

Responding to SOAP requests

When an Artix server receives a SOAP message, it checks the XML prologue to see what encoding codeset the message uses. If the XML prologue specifies the message's codeset, Artix uses the specified codeset to read the message and to write out its response to the request. For example, an Artix server that receives a request with the XML prologue shown in Example 5 decodes the message using `UTF-16` and encodes its response using `UTF-16`.

Example 5: *XML Prologue*

```
<?xml version="1.0" encoding="UTF-16"?>
```

If an Artix server receives a SOAP message where the XML prologue does not include the `encoding` attribute, the server will use whatever default codeset is specified in its configuration to decode the message and encode the response.

Working with Codesets using CORBA

Overview

The Artix CORBA plug-in supports both wide characters and narrow characters to accommodate an array of codesets. It also supports *codeset negotiation*. Codeset negotiation is the process by which two CORBA processes which use different *native codeset*s determine which codeset to use as a *transmission codeset*. Occasionally, the process requires the selection of a *conversion codeset* to transmit data between the two processes. The algorithm is defined in section 13.10.2.6 of the CORBA specification (http://www.omg.org/cgi-bin/apps/doc?formal/02-12-06.pdf).

Note: For CORBA programing in Java, you can specify a codeset other than the true native codeset.

Native codeset

A native codeset (NCS) is a codeset that a CORBA program speaks natively.

For Java, this is UTF-8 (`0x05010001`) for `char` and `String`, and UTF-16 (`0x00010109`) for `wchar` and `wstring`.

For C and C++, this is the encoding that is set by `setlocale()`, which in turn depends on the `LANG` and `LC_`*xxxx* environment variables.

You can configure the Artix CORBA plug-in's native codesets using the configuration variables listed in Table 9.

Table 9: *Configuration Variables for CORBA Native Codeset*

Configuration Variable	Description
`plugins:codeset:char:ncs`	Specifies the native codeset for narrow character and string data.
`plugins:codeset:wchar:ncs`	Specifies the native codeset for wide character and string data.

Conversion codeset

A conversion codeset (CCS) is an alternative codeset that the application registers with the ORB. More than one CCS can be registered for each of the narrow and wide interfaces. CCS should be chosen so that the expected input data can be converted to and from the native codeset without data loss. For example, Windows code page 1252 (`0x100204e4`) can be a conversion codeset for ISO-8859-1 (`0x00010001`), assuming only the common characters between the two codesets are used in the data.

You can configure the Artix CORBA plug-in's list of conversion codesets using the configuration variables listed in Table 10.

Table 10: *Configuration Variables for CORBA Conversion Codesets*

Configuration Variable	Description
`plugins:codeset:char:ccs`	Specifies the list of conversion codesets for narrow character and string data.
`plugins:codeset:wchar:ccs`	Specifies the list of conversion codesets for wide character and string data.

Transmission codeset

A transmission codeset (TCS) is the codeset agreed upon after the codeset negotiation. The data on the wire uses this codeset. It is either the native codeset, one of the conversion codesets, or UTF-8 for the narrow interface and UTF-16 for the wide interface.

Negotiation algorithm

Codeset negotiation uses the following algorithm to determine which codeset to use in transferring data between client and server:

1. If the client and server are using the same native codeset, no translation is required.
2. If the client has a converter to the server's codeset, the server's native codeset is used as the transmission codeset.
3. If the client does not have an appropriate converter and the server does have a converter to the client's codeset, the client's native codeset is used as the transmission codeset.

4. If neither the client nor the server has an appropriate converter, the server ORB tries to find a conversion codeset that both server and client can convert to and from without loss of data. The selected conversion codeset is used as the transmission codeset.
5. If no conversion codeset can be found, the server ORB determines if using UTF-8 (narrow characters) or UTF-16 (wide characters) will allow communication between the client and server without loss of data. If UTF-8 or UTF-16 is acceptable, it is used as the transmission codeset. If not, a `CODESET_INCOMPATIBLE` exception is raised.

Codeset compatibility

The final steps involve a compatibility test, but the CORBA specification does not define when a codeset is compatible with another. The compatibility test algorithm employed in Orbix is outlined below:

1. ISO 8859 Latin-*n* codesets are compatible.
2. UCS-2 (double-byte Unicode), UCS-4 (four-byte Unicode), and UTF-x are compatible.
3. All other codesets are not compatible with any other codesets.

This compatibility algorithm is subject to change without notice in future releases. Therefore, it is best to configure the codeset variables as explicitly as possible to reduce dependency on the compatibility algorithm.

Working with Codesets using Fixed Length Records

Overview

Artix fixed record length support enables Artix to interact with mainframe systems using COBOL. For example, many COBOL applications send fixed length record data over WebSphere MQ.

Artix provides a fixed binding that maps logical messages to concrete fixed record length messages. This binding enables you to specify attributes such as encoding style, justification, and padding character.

Encoding attribute

The Artix fixed binding provides an optional `encoding` attribute for both its `<fixed:binding>` and `<fixed:body>` elements. The `encoding` attribute specifies the codeset used to encode the text data. Valid values are any IANA codeset name. See http://www.iana.org/assignments/character-sets for details.

The `encoding` attribute for the `<fixed:binding>` element is a global setting; while the `<fixed:body>` attribute is per operation. Both settings are optional. If you do not set either, the default value is `UTF-8`.

For more details, see `fixed-binding.xsd`, available in *`install-dir`*`\iona\artix\`*`version`*`\schemas`.

Fixed binding example

The following WSDL example shows a fixed binding with `encoding` attributes for `<fixed:body>` elements. This binding includes two operations, `echoVoid` and `echoString`.

Example 6: *Fixed Length Record Binding*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:fixed="http://schemas.iona.com/bindings/fixed"
   xmlns:http="http://schemas.iona.com/transports/http"
   xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
   xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
   xmlns:mq="http://schemas.iona.com/transports/mq"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

Example 6: *Fixed Length Record Binding*

```
 xmlns:tns="http://www.iona.com/artix/test/I18nBase/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://www.iona.com/artix/test/I18nBase" name="I18nBaseService"
targetNamespace="http://www.iona.com/artix/test/I18nBase/"

 <message name="echoString">
     <part name="stringParam0" type="xsd:string"/>
 </message>

 <message name="echoStringResponse">
     <part name="return" type="xsd:string"/>
 </message>

 <message name="echoVoid"/>
 <message name="echoVoidResponse"/>

 <portType name="I18nBasePortType">
     <operation name="echoString">
         <input message="tns:echoString" name="echoString"/>
         <output message="tns:echoStringResponse" name="echoStringResponse"/>
     </operation>
     <operation name="echoVoid">
         <input message="tns:echoVoid" name="echoVoid"/>
         <output message="tns:echoVoidResponse" name="echoVoidResponse"/>
     </operation>
 </portType>

 <binding name="I18nFIXEDBinding" type="tns:I18nBasePortType">
     <fixed:binding/>
     <operation name="echoString">
         <fixed:operation discriminator="discriminator"/>
         <input name="echoString">
             <fixed:body encoding="ISO-8859-1">
                 <fixed:field bindingOnly="true" fixedValue="01" name="discriminator"/>
                 <fixed:field name="stringParam0" size="50"/>
             </fixed:body>
         </input>
         <output name="echoStringResponse">
             <fixed:body encoding="ISO-8859-1">
                 <fixed:field name="return" size="50"/>
             </fixed:body>
         </output>
     </operation>
```

Example 6: *Fixed Length Record Binding*

```
 <operation name="echoVoid">
            <fixed:operation discriminator="discriminator"/>
            <input name="echoVoid">
                <fixed:body>
                    <fixed:field name="discriminator" fixedValue="02" bindingOnly="true"/>
                </fixed:body>
            </input>
            <output name="echoVoidResponse">
                <fixed:body/>
            </output>
        </operation>
    </binding>
</definitions>
```

Further information

For more details on the Artix fixed length binding, see *Designing Artix Solutions*.

Working with Codesets using Message Interceptors

Overview

Artix provides support for codeset conversion for transports that do not have their own concept of headers. For example, IBM Websphere MQ, BEA Tuxedo, and Tibco Rendezvous. This generic support is implemented using an Artix message interceptor and WSDL port extensors.

For example, an Artix C++ client could use Artix Mainframe to access a mainframe system, using a binding for fixed length record over MQ. In this scenario, an Artix message interceptor can be configured to enable codeset conversion between ASCII and EBCDIC (Extended Binary Coded Decimal Interchange Code).

You can enable this codeset conversion simply by editing your WSDL file, or by using accessor methods in your application code. This section explains how to use both of these approaches.

Note: Codeset conversion set in application code takes precedence over the same settings in a WSDL file.

Codeset conversion attributes

This generic support for codeset conversion is implemented using a message interceptor. This message interceptor manipulates the following codeset conversion attributes:

`LocalCodeSet`	Specifies the codeset used locally by a client or server application.
`OutboundCodeSet`	Specifies the codeset used by the application for outgoing messages.
`InboundCodeSet`	Specifies the codeset used by the application for incoming messages.

You can specify these attributes to convert client-side requests and server-side responses. All three attributes are optional.

Configuring codeset conversion in a WSDL file

You can configure codeset conversion by setting the codeset conversion attributes in a WSDL file. Example 7 shows the contents of the Artix internationalization schema (`i18n-context.xsd`).

Example 7: *Artix i18n Schema*

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
           targetNamespace="http://schemas.iona.com/bus/i18n/context"
           xmlns:i18n-context="http://schemas.iona.com/bus/i18n/context"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">

  <xs:import namespace = "http://schemas.xmlsoap.org/wsdl/"
   schemaLocation="wsdl.xsd"/>

  <xs:element name="client" type="i18n-context:ClientConfiguration" />

  <xs:complexType name="ClientConfiguration">

   <xs:annotation>
       <xs:documentation> I18n Client Context Information
       </xs:documentation>
   </xs:annotation>

    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibilityElement" >
        <xs:attribute name="LocalCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="OutboundCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="InboundCodeSet" type="xs:string" use="optional" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

Example 7: *Artix i18n Schema*

```
 <xs:element name="server" type="i18n-context:ServerConfiguration"/>

  <xs:complexType name="ServerConfiguration" >
   <xs:annotation>
       <xs:documentation> I18n Server Context Information
       </xs:documentation>
   </xs:annotation>

    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibilityElement" >
        <xs:attribute name="LocalCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="OutboundCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="InboundCodeSet" type="xs:string" use="optional" />
      </xs:extension>
    </xs:complexContent>

  </xs:complexType>

</xs:schema>
```

The Artix internationalization message interceptor uses this schema as a port extensor. This enables you to configure codeset conversion attributes in a WSDL file.

Client/server WSDL example

The following example shows codeset conversion settings for a client and a server application specified in a sample WSDL file:

Example 8: *i18n Specified in a WDSL File*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="I18nBaseService"
   targetNamespace="http://www.iona.com/artix/test/I18nBase/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/artix/test/I18nBase/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:mq="http://schemas.iona.com/transports/mq"
    xmlns:http="http://schemas.iona.com/transports/http"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:fixed="http://schemas.iona.com/bindings/fixed"
    xmlns:i18n-context="http://schemas.iona.com/bus/i18n/context"
    xmlns:xsd1="http://www.iona.com/artix/test/I18nBase">
```

Example 8: *i18n Specified in a WDSL File*

```
  <import namespace="http://www.iona.com/artix/test/I18nBase"
   location="./I18nServiceBindings.wsdl"/>

    <service name="I18nService">

       <port binding="tns:I18nFIXEDBinding" name="I18nFIXED_HTTPPort">
             <http:address location="http://localhost:0"/>
        <i18n-context:client LocalCodeSet="ISO-8859-1" InboundCodeSet="UTF-8"/>
        <i18n-context:server LocalCodeSet="UTF-8" OutboundCodeSet="ISO-8859-1"/>
       </port>

     <port binding="tns:I18nFIXEDBinding" name="I18nFIXED_MQPort">

       <mq:client QueueManager="MY_DEF_QM" QueueName="MY_FIRST_Q" AccessMode="send"
         ReplyQueueManager="MY_DEF_QM" ReplyQueueName="REPLY_Q"
        CorrelationStyle="messageId copy" />

       <mq:server QueueManager="MY_DEF_QM" QueueName="MY_FIRST_Q"
       ReplyQueueManager="MY_DEF_QM" ReplyQueueName="REPLY_Q" AccessMode="receive"
        CorrelationStyle="messageId copy" />
       <i18n-context:client LocalCodeSet="UTF-8" InboundCodeSet=""/>
       <i18n-context:server LocalCodeSet="ISO-8859-1"/>
     </port>

  </service>

</definitions>
```

This sample WSDL file shows a single service named `I18nService`, with two bindings and two ports named `I18nFIXED_HTTPPort` and `I18nFIXED_MQPort`. The binding in both cases is fixed length record, each with a single operation.

Enabling codeset conversion in application code

You can also enable codeset conversion attributes by calling the following accessor methods in your C++ application code:

```
void setLocalCodeSet(const IT_Bus::String * val);
void setLocalCodeSet(const IT_Bus::String & val);

void setOutboundCodeSet(const IT_Bus::String * val);
void setOutboundCodeSet(const IT_Bus::String & val);

void setInboundCodeSet(const IT_Bus::String * val);
void setInboundCodeSet(const IT_Bus::String & val);
```

An Artix `ContextContainer` in the message interceptor, and the WSDL configuration are checked for each attribute. This is performed during the client's `intercept_invoke()` method and the server's `intercept_dispatch()` method. The client request buffer or server response buffer can be converted to another encoding as needed. This conversion can occur on the outbound or inbound intercept points.

The interceptor refers to the current context on a per-thread basis. For detailed information on Artix contexts, see the *Artix C++ Programmer's Guide*.

Linking with the context library

The message interceptor uses a common type library of Artix context attributes. The application must be linked with this common library, and with any transports that use this context to set or get attributes. The generated header files for this common library are available in the following directory:

`install-dir\artix\`*version*`\include\it_bus_pdk\context_attrs`

You must ensure that your application links with the context library that contains the generated stub code for `i18n-context.xsd`.

Client code example

Example 9 shows an example of the code that you need to add to your C++ client application:

Example 9: *Accessing i18n in C++ Client Code*

```
void
I18nTest::echoString(
    I18nBaseClient* client, const String& instr)
{
    String outstr;
    try
    {

    // Set the i18n request context to match the fixed binding encoding setting

    IT_Bus::Bus_var bus = client->get_bus();
    ContextRegistry * reg = bus->get_context_registry();

    ContextCurrent & cur = reg->get_current();
    ContextContainer * registered_ctx = cur.request_contexts();

    AnyType & i18n_ctx_info =
    registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_CLIENT_QNAME, true);
    ClientConfiguration & i18n_ctx_cfg = dynamic_cast<ClientConfiguration&> (i18n_ctx_info);

    // Set the Inbound codeset to match the binding encoding

    static const String LOCAL_CODE_SET = "ISO-8859-1";
    i18n_ctx_cfg.setLocalCodeSet(LOCAL_CODE_SET);

    const String & local_codeset = (*i18n_ctx_cfg.getLocalCodeSet());

    client->echoString(instr, outstr);

    // Read the i18n reply context

    registered_ctx = cur.reply_contexts();

    AnyType & i18n_ctx_reply_info =
     registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_CLIENT_QNAME, true);

    const ClientConfiguration & i18n_ctx_reply_cfg =
      dynamic_cast<const ClientConfiguration&> (i18n_ctx_reply_info);
```

Example 9: *Accessing i18n in C++ Client Code*

```
    const String * local_codeset_reply = i18n_ctx_reply_cfg.getLocalCodeSet();
    const String * outbound_codeset_reply = i18n_ctx_reply_cfg.getOutboundCodeSet();
    const String * inbound_codeset_reply = i18n_ctx_reply_cfg.getInboundCodeSet();

    if(local_codeset_reply)
     cout << "client LocalCodeSet reply context:" << local_codeset_reply->c_str() << endl;
    if(outbound_codeset_reply)
     cout << "client OutboundCodeSet reply context:"<< outbound_codeset_reply->c_str << endl;
    if(inbound_codeset_reply)
     cout << "client InboundCodeSet reply context" << inbound_codeset_reply->c_str() << endl;
    }

    catch (IT_Bus::ContextException& ce)
    {
    ...
    }
    catch (IT_Bus::Exception& ex)
    {
    ...
    }
    catch (...)
    {
    ...
    }
}
```

Server code example

Example 9 shows example of the code that you need to add to your C++ servant application.

Example 10: *Accessing i18n in C++ Server Code*

```
void
I18nServiceImpl::echoString(
    const String& stringParam0,
    String & var_return) IT_THROW_DECL((IT_Bus::Exception))
{

    var_return = stringParam0;

```

Example 10: *Accessing i18n in C++ Server Code*

```
try
{
// Read the i18n reply context

ContextRegistry * reg = m_bus->get_context_registry();

ContextCurrent & cur = reg->get_current();
ContextContainer * registered_ctx = cur.request_contexts();

AnyType & i18n_ctx_info =
registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_SERVER_QNAME, false);
const ServerConfiguration & i18n_ctx_cfg =
dynamic_cast<const ServerConfiguration&> (i18n_ctx_info);

const String * local_codeset = i18n_ctx_cfg.getLocalCodeSet();
const String * outbound_codeset = i18n_ctx_cfg.getOutboundCodeSet();
const String * inbound_codeset = i18n_ctx_cfg.getInboundCodeSet();

if(local_codeset)
  cout << "server LocalCodeSet request context:" << local_codeset->c_str() << endl;
if(outbound_codeset)
  cout << "server OutboundCodeSet request context:" << outbound_codeset->c_str() << endl;
if(inbound_codeset)
  cout << "server InboundCodeSet request context:" << inbound_codeset->c_str() << endl;

// Add code to change the reply context

registered_ctx = cur.reply_contexts();

AnyType & i18n_reply_ctx =
registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_SERVER_QNAME, true);

ServerConfiguration & i18n_reply_ctx_cfg =
 dynamic_cast<ServerConfiguration&> (i18n_reply_ctx);

// Set the local codeset to match the binding encoding

static const String LOCAL_CODE_SET = "ISO-8859-1";
i18n_reply_ctx_cfg.setLocalCodeSet(LOCAL_CODE_SET);

String & set_local_context = (*i18n_reply_ctx_cfg.getLocalCodeSet());

assert(set_local_context == LOCAL_CODE_SET);
}
```

Example 10: *Accessing i18n in C++ Server Code*

```
   catch (IT_Bus::ContextException& ex)
    {
     cout << "Error with server context" << ex.message() << endl;
    }
  catch (IT_Bus::Exception& ex)
    {
     cout << "Error with server context" << ex.message() << endl;
    }
  catch (...)
    {
     cout << "Unknown Error with server context" << endl;
    }
}
```

Artix configuration settings

Finally, you must also enable the i18n message interceptor in your Artix configuration file (`artix.cfg`). Example 11 shows the required settings:

Example 11: *Artix Configuration File Settings*

```
// Add to a demo/application scope.
interceptor{
    binding:artix:client_message_interceptor_list = "i18n-context:I18nInterceptorFactory";

    binding:artix:server_message_interceptor_list = "i18n-context:I18nInterceptorFactory";

    orb_plugins = ["xmlfile_log_stream", "i18n_interceptor"];

    event_log:filters = ["*=WARN+ERROR+FATAL"];
};
```

Further information

For more information details on writing Artix C++ applications and on Artix contexts, see the *Artix C++ Programmer's Guide*.

Routing with International Codesets

Overview

When routing between applications, Artix attempts to correctly map between different codesets. If both endpoints use bindings that support internationalization (i18n), Artix uses codeset conversion. If only one of the endpoints supports internationalization, the Artix endpoint supporting internationalization attempts to use codeset conversion on the messages.

The following bindings do not support internationalization:

- Tagged
- G2++
- XML

Routing between internationalized endpoints

When Artix is routing between internationalized endpoints, the receiving endpoint and the sending endpoint both behave independently of each other.

For example, if one endpoint of a router receives a request in Shift_JIS and the router is configured to use ISO-8859-1, the Shift_JIS request is properly decoded by the router.

However, when the request is passed on by the router, it is passed on in ISO-8859-1. If the two codesets are not compatible, there is a good chance that data will be lost in the conversion and the request will not be properly handled.

Note: If the codesets are not compatible, and data is lost in the router, Artix does not generate a warning.

Routing from non-internationalized to internationalized bindings

When Artix is routing from a non-internationalized endpoint to an internationalized endpoint, it uses the default codeset specified in the router's configuration for writing messages to internationalized endpoints. If the Artix router is configured to encode messages using a codeset that is different from the one used by the endpoint, you will lose data.

For example, if a Tibco application makes a request on a Web service through a router, the router receives non-internationalized data from the Tibco application. And the router then writes the SOAP message using the codeset specified in its configuration. If the Web service and the router are both configured to write in us-dk, the operation proceeds without a problem. The router receives the encoded response from the server and passes it back to the Tibco binding.

However, if the Web service is configured to accept data using us-dk, and the router is configured to encode data using Chinese, data may be lost between the router and the Web service due to codeset incompatibility.

Routing from internationalized to non-internationalized bindings

When Artix is routing SOAP messages to a non-SOAP endpoint, such as a Tuxedo server on a mainframe using the fixed plug-in, Artix handles the message transformations so that the SOAP application receives responses in the correct codeset.

For example, a Web service client in a Chinese locale encodes its requests in eucTW and invokes on a service that is hosted on a mainframe that is behind an Artix router, as shown in Figure 9.

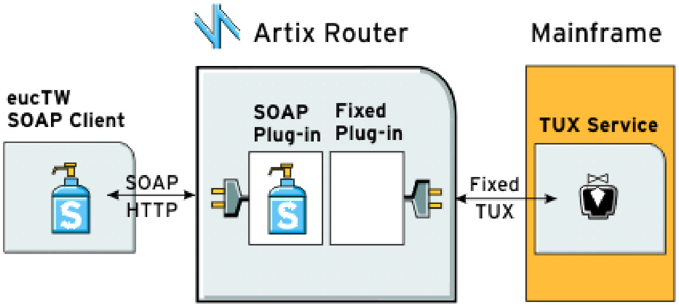

Figure 9: *Routing Internationalized Requests*

The Artix router would process the request as follows:

1. On receiving the SOAP request, the router inspects the XML prologue and decodes the message using the specified codeset (in this case, eucTW).
2. The fixed binding plug-in then writes out the message to the mainframe service.
3. When the mainframe sends its response back to the router, the fixed binding decodes the message and passes it back to the SOAP plug-in.
4. The SOAP plug-in inspects the message and determines the request to that corresponds it.
5. The SOAP plug-in then encodes the message using the codeset specified in the request (in this case, eucTW), and passes the response to the client.

Part III

Using Artix Services

In this part

This part contains the following chapters:

CHAPTER 7

Artix Generic Service

The Artix generic service is a comon artix service that can be used to house any Artix functionality that is implemented as a plug-in. It provide a consistent manner for deploying Artix services such as an Artix switch, the Artix locator service, or the Artix session manager. It also provides a clean and consisten way for shutting down deployed Artix services.

In this chapter

This chapter discusses the following topics:

The Artix Generic Service

Overview

The Artix generic service allows you to deploy and manage Artix services that are implemented as plug-ins using a consistent mechanism. It is a combination of an Artix plug-in container, one or more plug-ins that provide functionality, and an administrative interface that allows you to cleanly shutdown a service deployed using the generic service.

How it works

The Artix generic service is an Artix process that has no specific functionality beyond loading Artix plug-ins. It can be used to load a set of plug-ins that provide Artix functionality such as the Artix locator service or an Artix switch without writing any code. You simply need to provide the proper configuration for the service and start it.

In addition, the generic service exposes an IIOP port that enables it to be gracefully shutdown using the generic service administrative interface. The administrative interface is another generic process that simply requires that you configure it to contact the correct port.

Deployment patterns

The Artix generic service can be used to deployed a number of Artix services. By adding the proper plug-ins it can host the following Artix services:

- Artix locator
- Artix session manager
- Artix router
- Artix transformer
- Artix chain service

In addition, you can deploy any other Artix enabled service that you have implemented as a plug-in. Both Java and C++ plug-ins are supported.

What functionality the generic service exposes is controlled by the plug-ins that are listed in its `orb_plugins` list. For example, to deploy a router using the generic service you would include the router plug-in in the `orb_plugins` list. However, if you were deploying the locator service, you would replace the router plug-in with the locator plug-in.

Configuring the Generic Service

Overview

Each instance of the Artix generic service running on a host machine needs its own configuration scope to specify the unique port that its administrative interface listens on. The default scope in `artix.cfg` is `iona_services.artix_service`.

If you also want to use the Artix generic service administrative interface, you need to add a corresponding configuration scope for the administratice interface that will shutdown the service. The default scope included in `artix.cfg` is `iona_services.artix_service_admin`.

Having separate configuration scopes for each instance of the service also enables greater control over the resources the service uses. You can control the services threading and time-out behaviors.

For more information on Artix configuration, see "Configuring Artix" on page 23.

orb_plugins list

In addition to the plug-ins providing the service's functionality, the Artix generic service must load the following plug-ins:

- `iiop_profile`
- `iiop`
- `giop`

These plug-ins must be included in the services's `orb_plugins` list.

Service plug-in settings

The configuration variable that controls the behavior of the Artix generic service are in the `plugins:artix_service` namespace. Table 11 lists the variables and their settings.

Table 11: *Artix Generic Service Configuration Variables*

Variable	Effect
`shlib_name`	Specifies the name of the Artix service's shared library. This value should always be set to `it_artix_service_svr`.

Table 11: *Artix Generic Service Configuration Variables*

Variable	Effect
`iiop:port`	Specifies the port number that the service listens on for calls from its administrative interface. See "Service admin interface".
`iiop:host`	Specifies the name of the host computer that the service is running on. See "Service admin interface".
`direct_persistence`	Specifies if the service's object reference is persistent across multiple invocations.

Service admin interface

Each instance of the Artix generic service can have a corresponding administrative interface configuration scope. This scope must contain an entry for `initial_references:IT_ArtixServiceAdmin:reference`. This variable specifies the port number of this administrative interface's corresponding Artix service. The port number is specified using the `corbaloc` syntax:

```
corbaloc:iiop:1.2@hostname:port/IT_ArtixServiceAdmin
```

`hostname` is the name of the computer that the Artix service is running on. *`port`* is the port number that the Artix service is listening on.

Making a Generic Service a Background Process

Overview

Running an instance of the Artix generic service as a background process requires special configuration. Running the generic service in the background necessitates the configuration of an administrative interface that can shutdown the service. In addition, the service itself requires an additional sub-scope.

Along with the additional configuration, running the standalone service in the background requires that you follow certain naming conventions when defining your configuration scopes.

Scope naming conventions

In order to run as a background process the Artix generic service must run in a sub-scope named `artix_service` and its corresponding administrative interface's configuration scope must be named `artix_service_admin`. However, the super-scope containing the generic service's scopes can have any name. Therefore to configure multiple instances of the Artix generic service you will need to create multiple super-scopes that each contain a sub-scope for the service and its corresponding administrative interface.

For example, to configure a domain that contains three instances of the Artix generic service and have them all run in the background your configuration would need to contain one super-scope for each service. The configuration fragment in Example 12 shows an example of this. Each application group is given its own configuration super-scope and within each application scope an instance of the Artix generic service is configured separately.

Example 12: *Configuration for Multiple Instances of a Generic Service Running in the Background*

```
internal_banking
{
  artix_service
  {
   ...
  };
  artix_service_admin
  {
   ...
  };
};
atms
{
  artix_service
  {
   ...
  };
  artix_service_admin
  {
   ...
  };
};
partner_banking
{
  artix_service
  {
   ...
  };
  artix_service_admin
  {
   ...
  };
};
```

Service initialization scope

To run as a background process, the Artix generic service requires that an additional sub-scope be added to the `artix_service` scope. This scope, `artix_service_init`, is identical for every instance of the generic service. It is shown in Example 13.

Example 13: *Artix Service Initialization Scope*

```
artix_service
{
  ...
  artix_service_init
  {
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
                   "iiop"];
    plugins:artix_service:shlib_name = "it_artix_service_svr";
  };
};
```

Controlling the Generic Service

Overview

This section explains how to start and how to stop the Artix generic service, and lists the available startup options.

Starting the service

To start the Artix generic service, use the following script:

```
install-dir\artix\version\bin\start_artix_service
```

This script starts an instance of the Artix generic service using the default configuration scope of `iona_services.artix_service`.

itartix_service command

Alternatively, you can start the service directly using the following command:

```
itartix_service -ORBname orb_name -ORBdomain_name domain_name
    -ORBconfig_domains_dir domain_dir run [-background]
```

Table 12 describes the parameters taken by `itartix_service`.

Table 12: *itartix_service Parameters*

Parameter	Description
`-ORBname` *`orb_name`*	Specifies the scope under which the service finds its configuration.
`-ORBdomain_name` *`domain_name`*	Specifies the service's configuration file name. The configuration file name is *`domain_name`*`.cfg`. For example, given domain name `acmewidgets`, the service reads its configuration from `acmewidgets.cfg`.
`-ORBconfig_domains_dir` *`domain_dir`*	Specifies the location of the service’s configuration file.
`run`	Specifies that the service is to begin monitoring.

Table 12: *itartix_service Parameters*

Parameter	Description
`-background`	Specifies that the service is to run in the background. If this parameter is not specified, the service runs in the foreground of the active command window. **Note:** When the service is run in the background, the parent process uses a separate `artix_service.artix_service_init` sub-scope to start the service.

Stopping the service

To stop an Artix generic service, use the following script:

```
install-dir\artix\version\bin\stop_artix_service
```

This script stops an instance of the Artix generic service started using the start script, `start_artix_service`.

Alternatively, you can manually call the service's administrative interface to stop the service. Use the following command:

```
itartix_service_admin -ORBname orb_name
```

The value passed with the `-ORBname` flag specifies the configuration scope under which the administrative interface finds its configuration information. The vital entry in the administrative interfaces configuration is the entry for `initial_references:IT_ArtixServiceAdmin:reference`. This entry must contain the `corbaloc` address of the Artix service instance that you wish to shutdown.

Further information

For more information about configuring Artix see "Configuring Artix" on page 23.

Installing the Generic Service as a Windows Service

Overview

On Windows, you can install instances of the Artix generic service as a Windows service. This means the service starts at system boot and that limited management functionality is provided through the Windows service controls.

Installing the service

To install the Artix generic service as a Windows service, use the following script:

```
install-dir\artix\version\bin\install_artix_service
```

This script installs the Artix generic service using the default configuration scope of `iona_services.artix_service`.

Alternatively, you can install an instance of the service directly using the following command:

```
itartix_service -ORBname orb_name -ORBdomain_name domain_name
    -ORBconfig_domains_dir domain_dir install
```

Table 13 describes the parameters taken by `itartix_service`.

Table 13: *itartix_service Install Parameters*

Parameter	Description
`-ORBname` *orb_name*	Specifies the scope under which the service finds its configuration details.
`-ORBdomain_name` *domain_name*	Specifies the service's configuration file name. The configuration file name is *domain_name*`.cfg`. For example, given domain name `acmewidgets`, the service reads its configuration from `acmewidgets.cfg`.
`-ORBconfig_domains_dir` *domain_dir*	Specifies the location of the service's configuration file.
`install`	Specifies that the service is to installed as a Windows service.

Uninstalling the service

To uninstall the Artix standalone service as a Windows service use the following script:

```
install-dir\artix\version\bin\uninstall_artix_service
```

This script uninstalls the Artix standalone service using the default configuration scope of `iona_services.artix_service`.

Alternatively, you can uninstall instances of the service directly using the following command:

```
itartix_service -ORBname orb_name -ORBdomain_name domain_name
   -ORBconfig_domains_dir domain_dir uninstall
```

Table 13 describes the parameters taken by `itartix_service`.

Table 14: *itartix_service Uninstall Parameters*

Parameter	Description
`-ORBname` *`orb_name`*	Specifies the scope under which the service finds its configuration details.
`-ORBdomain_name` *`domain_name`*	Specifies the service's configuration file name. The configuration file name is *`domain_name`*`.cfg`. For example, given domain name `acmewidgets`, the service will read its configuration from `acmewidgets.cfg`.
`-ORBconfig_domains_dir` *`domain_dir`*	Specifies the location of the service’s configuration file.
`uninstall`	Specifies that the service is to remove itself from the Windows registry.

CHAPTER 8

Using Artix Switches

An Artix switch acts as a bridge between to non-Artix enabled applications. The Artix standalone service performs transport switching, message routing, and middleware bridging.

In this chapter

This chapter discusses the following topics:

The Artix Switch

Overview

Artix switches are a minimally invasive means of connecting applications that use different communication transports and message formats. It does not require that any Artix-specific code be compiled or linked into existing applications. A switch is created by loading the Artix routing plug-in into the Artix generic service. For more information on the Artix generic service, see "Artix Generic Service" on page 107.

How it works

An Artix switch is a routing daemon that listens for traffic on access points specified in an Artix contract. It re-directs messages based on the routing rules that you provide, and performs any transport switching and message formatting needed for the receiving application. Neither application is aware that its messages are being intercepted by Artix and no application development is required.

> **Note:** Artix requires that services being integrated use equivalent message layouts. For example, a service expecting a `long` cannot be sent a `float`.

The switch's behavior is controlled by a combination of an Artix contract and the Artix configuration file.

For more information on Artix contracts see the *Designing Artix Solutions*. For more information on configuring the Artix runtime see "Configuring Artix" on page 23.

Deployment patterns

Artix switched can be deployed in a number of ways. Two common deployment patterns are:

- Deploying multiple switches—each bridging between two applications.
- Deploying one switch to bridge between all applications in a domain.

Deploying multiple switches—each bridging between two applications. This approach simplifies designing integration solutions and provides faster processing of each message (shown in Figure 10). Using this approach, the Artix contract describing the interaction of the applications is simpler because it contains only the logical interfaces shared by the two applications, the bindings for each payload format, and the routing rules.

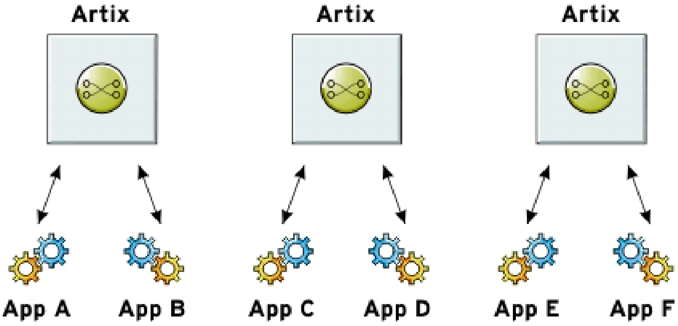

Figure 10: *Using Multiple Artix Switches*

Because most applications use only one network transport, the number of ports is minimal and the routing rules are simple. Keeping the contract simple also enhances the performance of each switch because it has less processing to do. In this approach, each switch's resource usage can also be limited by tailoring its configuration to optimize the switch for the integration task for which it is responsible.

Deploying one switch to bridge between all applications in a domain. This approach limits the number of external services required in your deployment environment (shown in Figure 11). This can simplify monitoring and installation of deployments. It also reduces the number of moving parts in an integration solution.

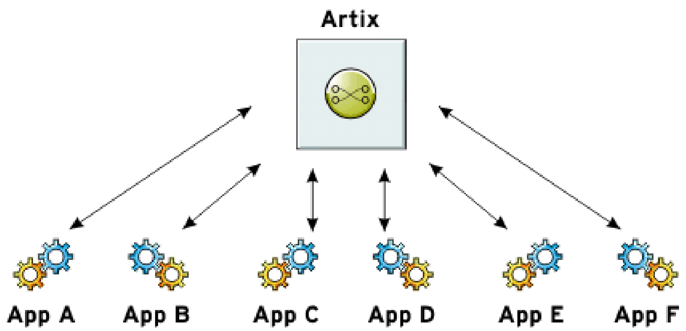

Figure 11: *Using a Single Artix Switch*

Configuring a Switch

Overview

Because Artix's routing functionality is implemented as a plug-in, you can make any Artix applciation a switch by adding routing rules to its contract and adding the routing plug-in to its orb_plugins list. To deploy a switch that does not require any modifications to existing applications, you can deploy a switch using the Artix generic service.

For more information on configuring the Artix generic service, see "Artix Generic Service" on page 107.

orb_plugins list

An Artix switch must include the following plug-ins in its orb_plugins list:

- iiop_profile
- iiop
- giop
- routing

WARNING: The routing plug-in must always be the last plug-in listed in orb_plugins.

Switch plug-in settings

You need to add configuration information to point the switch to the contract, or contracts, that contain the routing information it is to use. This is done with the plugins:routing:wsdl variable. This variable specifies the contracts the switch will parse for routing rules. The contract names are relative to the location from which the Artix switch is started.

For example, if a switch's configuration contained the following entry:

```
plugins:routing:wsdl=["route1.wsdl", "../route2.wsdl",
                      "/artix/routes/route3"];
```

The switch would expect that route1.wsdl was located in the directory in which it was started and route2.wsdl was located one directory level higher.

CHAPTER 9

Using the Artix Locator

The Artix locator enables Artix servers to publish their references for dynamic discovery by Artix clients.

In this Chapter

This chapter discusses the following topics:

Overview of the Artix Locator

Overview

A system with many servers cannot afford the overhead of manually propagating each server's contact information to the clients that need to contact them. Given the large number of clients and the distributed nature of enterprise level deployments, the time required to accomplish this, and the room for error, are too great. Also, over time, hardware upgrades, machine failures, or site reconfiguration will require you to move servers and repeat the exercise of propagating the server's information to all clients.

The Artix locator service isolates clients from changes in a server's contact information. The Artix contract defining how the client contacts the server contains the address for the Artix locator and it is the locator that provides the client with a reference to the server. Servers are automatically registered with the locator when they start-up.

Locator service components

The Artix locator's functionality is built into two plug-ins:

Locator Service Plug-in (`service_locator`)
This is the central service plug-in. It accepts service registrations, performs service look-ups, hands out references to clients who request them, and controls the load balancing of service groups.

Locator Endpoint Manager Plug-in (`locator_endpoint`)
This is the portion of the locator that resides in a registered service. It registers its location with the service plug-in and monitors the health of the service plug-in to ensure fault tolerance.

How do the plug-ins interact?

Figure 12 shows an overview of how the locator plug-ins are deployed in an Artix system. While in this example, the locator service plug-in is deployed in a standalone service, it can be deployed in any Artix process.

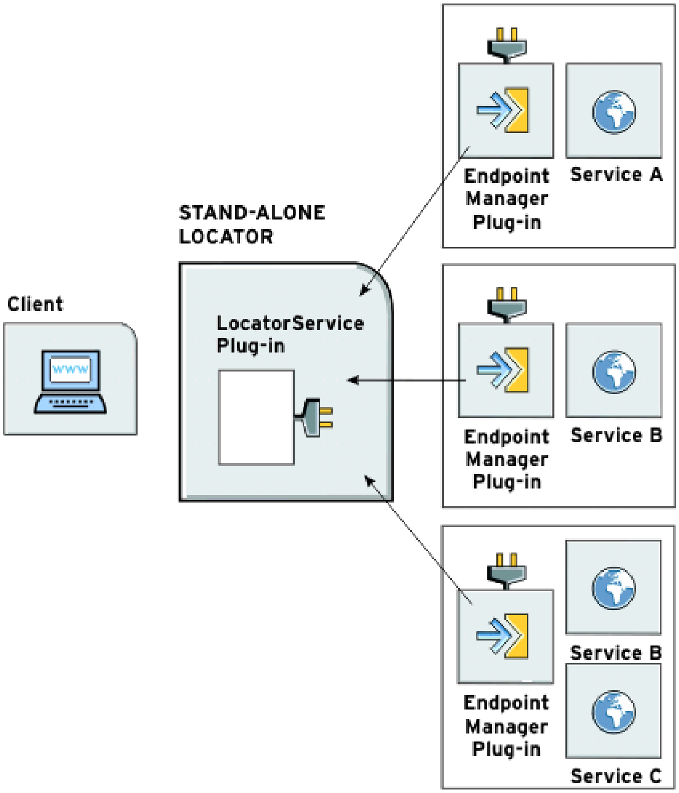

Figure 12: *The Locator Plug-ins*

The endpoint manager plug-ins are deployed in the server processes that contain services registered with the locator. A process can host two services, (for example, *Service C* and *Service D* in Figure 12), but the process can have only one endpoint manager. The endpoint manager plug-ins are in constant communication with the locator service plug-in to report on endpoint health, and to check on the health of the locator service.

Load balancing

The locator also provides load balancing functionality. When a group of services register with the locator using the same service name, the locator considers the services as a single service and uses a round-robin load balance algorithm to hand out references to the separate instances.

As shown in Figure 13, when each client makes a request for `widget_service`, the locator cycles through the pool of registered `widget_service` instances. For example, when `client4` makes a request, the locator starts handing out references from the top of the pool (`widget_service`$_a$).

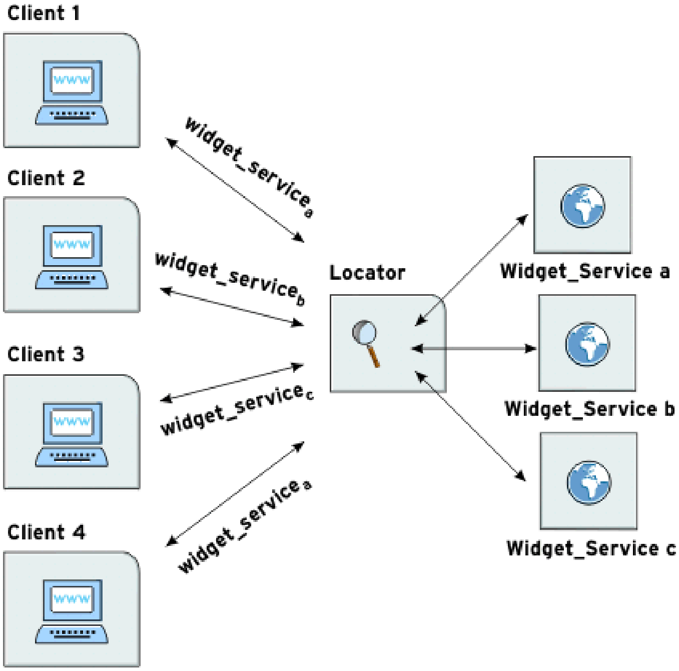

Figure 13: *Locator Load Balancing*

Services can also implement their own load balancing internally using calls to the Artix locator service that temporarily remove them from the pool of active references.

Deploying the Locator

Overview

The Artix locator is implemented as a group of Artix plug-ins. This means that any Artix application can host the locator service by loading the `service_locator` plug-in. However, it is recommended that you deploy it using the Artix generic service.

In either case, the locator service requires modifications to the Artix configuration domain that the locator runs in. You must also generate a copy of `locator.wsdl`, the contract that describes the locator service and contains its contact information.

Configuring the locator service

To confugre the Artix generic service to be a locator service you need to modify an instance of the generic service as follows:

1. Ensure that it loads the locator service plug-in, `service_locator`. In addition, the locator must load the `soap` and `http` plug-ins as all of its communication is done using SOAP over HTTP.
2. In the locator's configuration scope, specify that the service plug-in reads the correct Artix contract for the locator by setting `plugins:locator:service_url` to point to the copy of `locator.wsdl` that contains the address for this instance of the locator.

Example 14 shows the configuration scope used to start the locator.

Example 14: *Locator Configuration Scope*

```
locator_service
{
  artix_service
  {
    plugins:locator:service_url="locator.wsdl"
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
                   "iiop", "soap", "http", "service_locator"];
    ...
  };
  ...
};
```

For more information on configuring the Artix generic service see “Artix Generic Service” on page 107.

You must also configure the port that the locator runs on. To do this, you must modify the following file:

```
install-dir\artix\version\wsdl\locator.wsdl
```

Deploying on a fixed port

To deploy the locator on a well-known, fixed port, open `locator.wsdl` in any text editor and edit the `<soap:address>` entry at the bottom of the contract to specify the correct address. Example 15 shows a modified locator service contract entry. The highlighted part has been modified to point to the desired address.

Example 15: *Locator Service Address*

```
<service name="LocatorService">
  <port name="LocatorServicePort" binding="ls:LocatorServiceBinding">
    <soap:address
   location="http://localhost:8080/services/locator/LocatorService"/>
  </port>
</service>
```

Deploying on a dynamic port

To deploy the locator on a dynamically allocated port, you must generate a custom service mainline for the locator service. This custom service mainline must include code that will publish a new copy of its contract with valid contact information to allow services to find it. Example 16 shows how to publish the locator's contract.

Example 16: *Dynamic Locator Service*

```
\\ C++
IT_Bus::Bus_var bus = IT_Bus::init(argc, argv,
                                   "locator_service");

// Now we write out the updated WSDL for the Locator Services

// Get the WSDL Defintions object.
IT_Bus::QName service_name("",
                           "LocatorService",
                           "http://ws.iona.com/locator");
IT_Bus::Service * service = bus->get_service(service_name);
const IT_WSDL::WSDLDefinitions & definitions =
                                service->get_wsdl_definitions();

// Serialize the WSDL model to another wsdl file.
IT_Bus::FileOutputStream stream("active-locator.wsdl");
IT_Bus::XMLOutputStream xml_stream(stream, true);
definitions.write(xml_stream);
stream.close();

IT_Bus::run();
```

For information on creating Artix service mainlines see *Developing Artix Applications in C++*.

The configuration for the new service will be similar to the configuration used to deploy the locator service using the Artix generic service. However, when deploying the locator with a dynamic port, you configure it to use the copy of `locator.wsdl` shipped with Artix instead of a modified version.

Starting the locator

When the locator has been generated and properly configured, it can be started just like any other application.

Registering a Server with the Locator

Overview

A server does not need to have its implementation changed to work with the Artix locator. All that is required is that the server be configured to load the correct plug-ins and to reference the correct locator contract.

Configuring the server

Any server that wishes to register itself with the locator must load the following plug-ins in addition to the transport and payload plug-ins that it requires:

- `soap`
- `http`
- `locator_endpoint`

`locator_endpoint` enables the server to register with the running locator.

The server's configuration also must have `plugins:locator:wsdl_url` set to the appropriate locator contract.

Example 17 shows the configuration scope of a server that registers with the locator service.

Example 17: *Server Configuration Scope*

```
my_server
{
  plugins:locator:wsdl_url="locator.wsdl";
  orb_plugins = ["xmlfile_log_stream", "soap", "http", "tunnel",
    "locator_endpoint"];
 };
```

`my_server` provides its services using SOAP over IIOP so in addition to the locator plug-ins it also loads the `tunnel` plug-in.

For more information on Artix configuration see "Configuring Artix" on page 23.

Server registration

When a properly configured server starts up, it automatically registers with the locator specified by the contract pointed to by `plugins:locator:wsdl_url`.

You can register multiple instances of the same server with a locator. The locator generates a pool of references for the server type. When clients make a request for a server, the locator supplies references from this pool using a round-robin algorithm. For more information on load balancing see "Load Balancing" on page 137.

Obtaining References from the Locator

Overview

Unlike servers, clients must be specifically written to work with the Artix locator. There are three steps a client must take to obtain a server reference from the Artix locator:

1. Instantiate a proxy for the locator service.
2. Look up the desired server's endpoint using the locator service proxy.
3. Create a proxy for the desired server using the returned endpoint.

Instantiating a locator service proxy

Before a client can invoke any of the look up methods on the locator service, it must create a proxy to forward requests to the running locator. To do this the client creates an instance of `LocatorServiceClient` using the following information:

- The locator service contract name, `locator.wsdl`.
- The locator service QName.
- The port name used in the locator service contract, `LocatorServicePort`.

Note: For more information on Artix proxy constructors, see the *Artix C++ Programmer's Guide*.

Example 18 shows how to instantiate a locator service proxy. The parameters used to create the locator service's QName, `LocatorService` and `http://ws.iona.com/locator`, should never be modified.

Example 18: *Instantiating a Locator Service Proxy*

```
// C++
QName locator_service_name("", "LocatorService",
                           "http://ws.iona.com/locator");
locator_proxy = new LocatorServiceClient("locator.wsdl",
                                         locator_service_name,
                                         "LocatorServicePort");
```

Looking up a server's endpoint

After instantiating a locator service proxy, a client can then look up servers using the proxy's `lookup_endpoint()` method. This method has the following signature:

```
void lookup_endpoint(lookupEndpoint input,
                     lookupEndpointResponse output);
```

`input` Contains the QName of the server the client is looking up. The QName is set using the `setservice_qname()` method. The QName includes the service name specified in the Artix contract's `<service>` tag and the target namespace of the Artix contract.

`output` Contains a reference to the server. If the locator cannot find a registered instance of the requested server, `lookup_endpoint()` returns an `endpointNotExistFault` exception.

Example 19 shows the client code to look up an instance of the widget ordering service, `orderWidgetService`.

Example 19: *Looking up a Server Using the Locator Service*

```
// C++
// Create the QName for the server
QName service_name("", "orderWidgetsService",
                     "http://widgetVendor.com/widgetOrderForm");

// Create lookup input parameter
lookupEndpoint input;
input.setservice_qname(service_name);

// The output parameter is set by lookup_endpoint
lookupEndpointResponse output;

// call lookup_endpoint on the locator proxy
try
{
  locator_proxy->lookup_endpoint(input, output);
}
catch (IT_BusServices::endpointNotExistFault& e)
{
  // handle fault
}
```

Creating a server proxy

The client uses the reference returned in the output parameter of `lookup_endpoint()` to instantiate a server proxy for making requests on the requested server. To instantiate the proxy, use the correct proxy class for the server you have requested and pass the return value of the returned `lookupEndpointResponse`'s `getservice_endpoint()` method to the proxy class' constructor.

> **Note:** Because the Artix locator's look up is only one level deep, it is possible that the original look up can return a reference to a second Artix locator. Clients running in an environment where multiple locator redirects are possible must be explicitly designed to handle this situation.

Example 20 shows the client code for creating a proxy widget server from the results of the look up performed in Example 19 on page 135.

Example 20: *Instantiate a Proxy Server*

```
// C++
orderWidgetsClient widget_proxy(output.getservice_endpoint());
```

For more information on writing Artix client code, see the *Artix C++ Programmer's Guide*.

Load Balancing

Overview

The Artix locator provides a lightweight mechanism for balancing workloads among a group of servers. When a number of servers with the same service name register with the Artix locator, it automatically creates a list of the references and hands out the references to clients using a round robin algorithm. This process is invisible to both the clients and the servers.

Starting to load balance

When the locator is deployed and your servers are properly configured, you need to bring up a number of instances of the same service. This can be accomplished by one of two methods depending on your system topology:

1. Create an Artix contract with a number of ports for the same service and have each server instance startup on a different port.
2. Create a number of copies of the Artix contract defining the service, change the port information so each copy has a separate port address, and then bring up each server instance using a different copy of the Artix contract.

Note: The locator uses the service name specified in the `<service>` tag of the server's Artix contract to determine if it is part of a group. If you are using the Artix locator to load balance, your services should be associated with the same binding and logical interface.

As each server starts up it automatically registers with the locator. The locator recognizes that the servers all have the same service name specified in their Artix contracts and creates a list of references for these server instances.

As clients make requests for the service, the locator cycles through the list of server instances to hand out references.

Controlling Server Workloads

Overview

Services can request that they temporarily be taken off of the locator’s list of active references. This is particularly useful for managing the workloads placed on services. When they reach a certain capacity, a service can disappear from any new clients wishing to access it. When the service’s workload is reduced it can then reappear and once again become available to new clients.

Procedure

To control the registered state of service, perform the following steps:

1. Obtain a handle for the service with which you intend to work.
2. Use the obtained handle to temporarily deregister the service from the locator.
3. Use the obtained handle to reregister the service with the locator.

Get a service instance

To get an instance of a service, use `IT_Bus::get_service()` on a bus instance. The `get_service()` method takes the QName of the desired service and returns a generic service handle, `IT_Bus::Service*`.

Note: A bus instance can only return service handles for services that are activated on that particular bus.

Example 21 shows how to obtain a handle for a service from the active bus.

Example 21: *Obtaining a Service Handle*

```
//C++
// Build service QName
IT_Bus::QName service_name("", "MMService", "http://MM.com");

// Get the service handle from the active bus
IT_Bus::Service* = bus->get_service(service_name);
```

For more information on using `get_service()` see the *Artix C++ Programmer’s Guide*.

Deregistering a service

To temporarily deregister a service, use the `reached_capacity()` method of the service handle returned by the active bus. This method informs the service's endpoint manager that the service is busy and does not want to receive requests from any new clients. The endpoint manager then contacts the locator and asks to be removed from the list of available services.

Note: Clients that already have a valid reference for the service will still be able to make request on the service when it has been deregistered.

Example 22 shows how to call `reached_capacity()`.

Example 22: *Calling reached_capacity()*

```
\\ C++
\\ Service otained previously
service->reached_capacity();
```

Reregistering a service

When the service is ready to be reregistered, use the `below_capacity()` method of the service handle used when deregistering the service. `below_capacity()` informs the endpoint manager that the service is capable of accepting requests from new clients. The endpoint manager then contacts the locator and asks to be placed on the list of available services.

Example 23 shows how to call `reached_capacity()`.

Example 23: *Calling below_capacity()*

```
\\ C++
\\ Service otained previously
service->below_capacity();
```

Fault Tolerance

Overview

Enterprise level deployments demand that applications can cleanly recover from occasional failures. The Artix locator is designed to recover from the two most common failures faced by a look-up service:

- Failure of a registered endpoint.
- Failure of the look-up service.

Endpoint failure

When an endpoint gracefully shuts down, it notifies the locator that it will no longer be available. The locator removes the endpoint from its list so it cannot give a client a reference to a dead endpoint. However, when an endpoint fails unexpectedly, it cannot notify the locator and the locator can unknowingly give a client an invalid reference causing the failure to cascade.

To mitigate the risk of passing invalid references to clients, the locator service occasionally pings all of its registered endpoints to see if they are still running. If an endpoint does not respond to a ping, the locator removes that endpoint's reference.

You can adjust the interval between locator service pings by setting the `plugins:locator:peer_timeout` configuration variable. The default setting is `4` seconds. For more information see "Configuring Artix" on page 23.

Service failure

When the locator service fails, all the references to the registered endpoints are lost and the active endpoints are no longer registered with the locator. To ensure that the active endpoints reregister with the locator when it restarts, the endpoints, after the locator has missed its ping interval, periodically attempts to reregister with the locator until they are successful.

You can adjust the interval at which the endpoint pings the locator by setting the `plugins:session_endpoint_manager:peer_timout` configuration variable. The default setting is `4` seconds. For more information see "Configuring Artix" on page 23.

CHAPTER 10

Using the Artix Session Manager

The Artix session manager enables you to manage service resources.

In this chapter

This chapter discusses the following topics:

Introduction to Session Management in Artix

Overview

The Artix session manager is a group of plug-ins that work together to manage the number of concurrent clients accessing a group of services. This enables you to control how long each client can use the services in the group before having to check back with the session manager.

Note: The Artix session manager is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports the Artix session manager.

The two main session manager plug-ins are:

Session manager service plug-in (`session_manager_service`)
This is the central service plug-in. It accepts and tracks service registration, hands out sessions to clients, and accepts or denies session renewal.

Session manager endpoint plug-in (`session_endpoint_manager`)
This is the portion of the session manager that resides in a registered service. It registers its location with the service plug-in, and accepts or rejects client requests based on the validity of their session headers.

The Artix session manager also has a pluggable policy callback mechanism that enables you to implement your own session management policies. The session manager includes a simple policy callback plug-in, `sm_simple_policy`. This provides control over the allowable duration for a session and the maximum number of concurrent sessions allowed for each group.

How do the plug-ins interact?

Figure 14 shows how the session manager plug-ins are deployed in an Artix system. The session manager service plug-in and the policy callback plug-in are both deployed into the same process.

While, in this example, these plug-ins are deployed into a standalone service, they can be deployed in any Artix process. The session manager service plug-in and the policy plug-in interact to ensure that the session manager does not hand out sessions that violate the policies established by the policy plug-in.

The endpoint manager plug-ins are deployed into the server processes that contain session managed services. A process can host two services (for example, *Service C* and *Service D* in Figure 14), but the process can have only one endpoint manager. The endpoint manager plug-ins are in constant communication with the session manager service plug-in to report on endpoint health. They also receive information on new sessions that have been granted to the managed services, and check on the health of the session manager service.

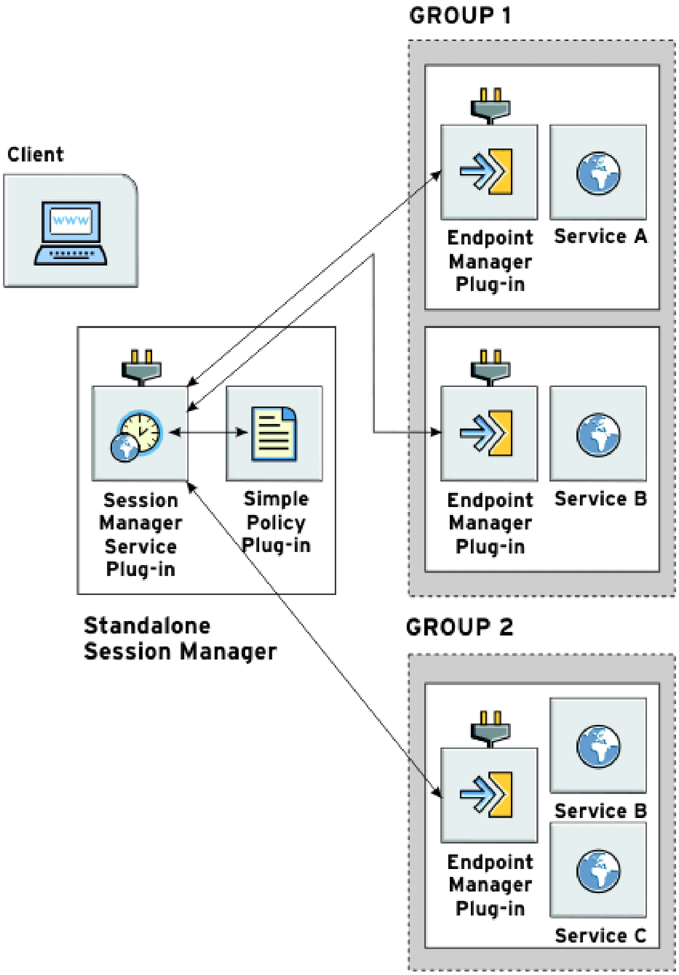

Figure 14: *The Session Manager Plug-ins*

What are sessions?

The session manager controls access to services by handing out *sessions* to clients who request access to the services. A session is a pass that provides access to the services in a specific group for a specific time.

For example, if a client application wants to use the services in the `sales` group, it asks the session manager for a session with the `sales` group. The session manager then checks and see if the `sales` group has an available session, and if so, it returns a session ID and the list of `sales` service references to the client. The session manager then notifies the endpoint managers in the `sales` group that a new session has been issued. It also supplies the new session ID, and the duration for which the session is valid.

When the client then makes requests on the services in the `sales` group, it must include the session information as part of the request. The endpoint manager for the services then checks the session information to ensure it is valid. If it is, the request is accepted. If it is not, the request is rejected.

If the client wants to continue using the `sales` services beyond the duration of its lease, the client will have to ask the session manager to renew its session before the session expires. When a client's session has expired, it must request a new one.

What are groups?

The Artix session manager does not pass out sessions for each individual service that is registered with it. Instead, services are registered as part of a *group*, and sessions are handed out for the group. A group is a collection of services that are managed as one unit by the session manager. While the session manager does not specify that the services in a group be related, it is recommended that the endpoints have some relationship.

A service's group affiliation is controlled by the configuration scope under which it is run. To change a service's group, edit the value for `plugins:session_endpoint_manager:default_group` in the process configuration scope. For more information on Artix configuration, see "Configuring Artix" on page 23.

Deploying the Session Manager Service

Overview

Because the Artix session manager is implemented as a group of plug-ins, any Artix application can host the session manager's core functionality by loading the `session_manager_service` and `sm_simple_policy` plug-ins. However, it is recommended that you deploy it using the Artix generic service.

In either case, the session manager requires modifications to the Artix configuration domain that the session manager is run in. You also need to generate a copy of `session-manager.wsdl`. This is the contract that describes the session manager and contains the session manager's contact information.

Configuring the session manager

To deploy the session manager, you must include the session manager service plug-in, `session_manager_service` and the session manager policy plug-in, `sm_simple_policy` in the `orb_plugins` list. In addition, the session manager must load the `soap` and `http` plug-ins because all of its communication uses SOAP over HTTP.

In the session manager's configuration scope, you must specify the location for the session manager's contract. You do this by setting `plugins:session_manager_service:service_url` to point to the copy of `session-manager.wsdl` containing the contact information for this session manager.

Example 24 shows the configuration scope used to start the session manager.

Example 24: *Session Manager Configuration Scope*

```
managed_sessions
{
  artix_service
  {
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop", "iiop",
                   "soap", "http", "session_manager_service",
                   "sm_simple_policy"];
    plugins:session_manager_service:service_url="session-namager.wsdl"
    ...
  };
  ...
};
```

For more information on configuring a service to deploy using the Artix generic service, see “Artix Generic Service” on page 107.

Deploying on a fixed port

To deploy the session manager on a fixed port you must modify `session-manager.wsdl`, in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the session manager listens. To do this, open `session-manager.wsdl` in any text editor, and edit the `<soap:address>` entry for the `SessionManagerService` to specify the correct address. Example 25 shows a modified session manager contract entry. The highlighted part specifies the desired address.

Example 25: *Session Manager Address*

```
<service name="SessionManagerService">
  <port name="SessionManagerPort" binding="sm:SessionManagerBinding">
    <soap:address
   location="http://localhost:8080/services/sessionManagement/sessionManagerService"/>
  </port>
</service>
```

Deploying on a dynamic port

To deploy the session manager on a dynamically allocated port, you need to generate a custom service mainline for the session manager that includes code to publish the session manager's contract after it has been initialized. Example 26 shows how to publish the session manager's contract.

Example 26: *Dynamically Located Session Manager*

```
IT_Bus::Bus_var bus = IT_Bus::init(argc, argv,
                                   "managed-sessions");

// Now we write out the updated WSDL for the session manager

// Get the WSDL Defintions object.
IT_Bus::QName service_name("",
                           "SessionManagerService",
                          "http://ws.iona.com/session-manager");
IT_Bus::Service * service = bus->get_service(service_name);
const IT_WSDL::WSDLDefinitions & definitions =
                                 service->get_wsdl_definitions();

// Serialize the WSDL model to another wsdl file.
IT_Bus::FileOutputStream stream("active-smservice.wsdl");
IT_Bus::XMLOutputStream xml_stream(stream, true);
definitions.write(xml_stream);
stream.close();

IT_Bus::run();
```

For information on creating Artix service mainlines see *Developing Artix Applications in C++*.

The configuration for the new service will be similar to the configuration used to deploy the session manager using the Artix generic service. However, when deploying the session manager on a dynamic port, you configure the session manager to use the copy of `session-manager.wsdl` shipped with Artix.

You can limit the range of ports that the session manger is deployed on by specifying a range of ports for the session managers SOAP or HTTP address. Example 27 shows a modified session manager contract entry. The highlighted part specifies the desired range of ports.

Example 27: *Session Manager Port Range*

```
<service name="SessionManagerService">
  <port name="SessionManagerPort" binding="sm:SessionManagerBinding">
    <soap:address
   location="http://localhost:11000-11100/services/sessionManagement/sessionManagerService"/>
  </port>
</service>
```

When the session manager has been generated and correctly configured, it can be started just like any other application. The only difference is that the session manager must be started before any servers that need to register with it.

Registering a Server with the Session Manager

Overview

Services that wish to be managed by the session manager must register with a running session manager. To do this, the servers instantiating these services must load the session manager endpoint plug-in and correctly configure themselves. They do not require any special application code.

When registered with a session manager, the services only accept requests containing a valid session header. All clients wishing to access the services must be written to support session managed services.

Configuring the server

Any server hosting services that are to be managed by the session manager must load the following plug-ins in addition to the transport and payload plug-ins it requires:

- `soap`
- `http`
- `session_endpoint_manager`

`session_endpoint_manager` enables the server to register with a running session manager.

The server's configuration also needs to set the following configuration variables:

plugins:session_endpoint_manager:wsdl_url points to the contract describing the contact information for the session manager that manages the services.

plugins:session_endpoint_manager:endpoint_manager_url points to the contract describing the contact information for the endpoint manager for this server. This enables the session manager to contact the service to with updated state information.

plugins:session_endpoint_manager:default_group specifies the default group name for the services instantiated by the server.

Example 28 shows the configuration scope of a server that hosts services managed by the session manager.

Example 28: *Server Configuration Scope*

```
acme_server
{
  orb_plugins = ["xmlfile_log_stream", "soap", "http", "fixed", "session_endpoint_manager"];
  plugins:session_endpoint_manager:wsdl_url="session-manager-service.wsdl";
  plugins:session_endpoint_manager:endpoint_manager_url="session-manager-endpoint.wsdl";
  plugins:session_endpoint_manager:deafult_group="acme_group";
 };
```

A server loaded into the `acme_server` configuration scope is managed by the session manager at the location specified in `session-manager-service.wsdl`. Its endpoint manager comes up at the address specified in `session-manager-endpoint.wsdl`, and by default all services instantiated by the server belongs to the `acme_group` session manager group.

For more information on Artix configuration see "Configuring Artix" on page 23.

You also need to configure the port on which the endpoint manager runs. To do this, modify `session-manager.wsdl`, in the `wsdl` folder of your Artix installation, to specify the HTTP address that the endpoint manager available at. Using any text editor, open `session-manager.wsdl` and edit the `<soap:address>` entry for the `SessionEndpointManagerService` to specify the proper address. Example 29 shows a modified session manager contract entry. The highlighted part specifies the desired address.

Example 29: *Endpoint Manager Address*

```
<service name="SessionEndpointManagerService">
  <port name="SessionEndpointManagerPort" binding="sm:SessionEndpointManagerBinding">
    <soap:address
   location="http://localhost:8080/services/sessionManagement/sessionEndpointManager"/>
  </port>
</service>
```

In the server's configuration scope, specify the endpoint manager plug-in to read the correct Artix contract for the endpoint manager. You can do this by setting `plugins:session_endpoint_manager:endpoint_manager_url` to point to the copy of `session-manager.wsdl` containing the address for this instance of the endpoint manager.

Server registration

When a properly configured server starts up, it automatically registers with the session manager specified by the contract pointed to by `plugins:session_endpoint_manager:wsdl_url`.

Working with Sessions

Overview

Clients wishing to make requests from session managed services must be designed explicitly to interact with the Artix session manager and pass session headers to the session managed services.

The client takes the following steps when making requests on a session managed service:

1. Instantiate a proxy for the session management service.
2. Start a session for the desired service's group using the session manager proxy.
3. Obtain the list of endpoints available in the group.
4. Create a service proxy from one of the endpoints in the group.
5. Build a session header to pass to the service.
6. Invoke requests on the endpoint using the proxy.
7. Renew the session as needed.
8. End the session using the session manager proxy when finished with the services.

Instantiate a session manager proxy

Before a client can request a session from the session manager, it must create a proxy to forward requests to the running session manager. To do this, the client creates an instance of `SessionManagerClient` using the session manager's contract name, `session-manager.wsdl`.

Example 30 shows how to instantiate a session manager proxy.

Example 30: *Instantiating a Session Manager Proxy*

```
// C++
SessionManagerClient session_manager_proxy = new
    SessionManagerClient("session_manager.wsdl");
```

For more information on instantiating Artix proxies, see the *Artix C++ Programmer's Guide*.

Start a session

After instantiating a session manager proxy, a client can then start a session for the desired service's group using the session manager's begin_session() method:

```
void begin_session(IT_Bus_Services::BeginSession input,
                   IT_Bus_Services::BeginSessionResponse output);
```

input Contains the name of the desired group and the desired duration of the session. The group name is set using the setendpoint_group() method. The group name can be any valid string and corresponds to the default group name set in the service's configuration scope as described in "Configuring the server" on page 150.

Specifying the session duration The session duration is set using the setprefered_renew_timeout() method. The duration is specified in seconds. If the specified duration is less than the value specified by the session manager's min_session_timeout configuration setting, it is set to the configured minimum value. If the specified duration is higher than the value specified by the session manager's max_session_timeout configuration setting, it is set to the configured max value. For more information see "Configuring Artix" on page 23.

output Contains the information needed to use the session.

When a session is returned in output, you must extract the session ID to work with the session. This is done using getsession_id(), which returns the session ID as an IT_Bus_Services::SessionID.

Example 31 shows the client code to begin a session for `acme_group`.

Example 31: *Beginning a Session*

```
// C++
IT_Bus_Services::BeginSession begin_session_request;
IT_Bus_Services::BeginSessionResponse begin_session_response;

// set the group to request
begin_session_request.setendpoint_group("acme_group");
// set session renewal interval to 10 mins
begin_session_request.setpreferred_renew_timeout(600);

session_mgr.begin_session(begin_session_request,
                          begin_session_response);

IT_Bus_Services::SessionId session;
session =
   begin_session_response.getsession_info().getsession_id();
```

Get a list of endpoints in the group

The session manager hands out sessions for a group of services. Therefore, to get an individual service to make requests on, a client needs to get a list of the services in the session's group. The session manager proxy's `get_all_endpoints()` method returns a list of all endpoints registered to the specified group:

```
void get_all_endpoints(IT_Bus_Services::GetAllEndpoints request,
        IT_Bus_Services::GetAllEndpointsResponse response)
```

`request` Contains the session ID that you are requesting services for. Set the session ID using the `setsession_id()` method on `request` with the session ID returned from the session manager.

`response` Contains the list of services returned from `get_all_endpoints()`. If the group has no services, `response` will be empty.

Example 32 shows how to get the list of services for a group.

Example 32: *Retrieving the List of Services in a Group*

```
//C++
IT_Bus_Services::GetAllEndpoints request;
IT_Bus_Services::GetAllEndpointsResponse response;

// group session initialized above.
get_all_endpoints_request.setsession_id(session);

session_mgr.get_all_endpoints(request, response);
```

Create a proxy for the requested service

The client can use any of the services returned by `get_all_endpoints()` to instantiate a service proxy. To instantiate the proxy, you must first narrow down the list returned services to the desired one. `GetAllEnpointsResponse` contains an array of references to active services that can be retrieved using its `getendpoints()` method. You can use simple indexing to get one of the references. For example, to use the first service in the list you would use the following:

```
response.getendpoints()[0]
```

Because the session manager simply returns the services in the order that the services registered with the session manager, the clients must be responsible for circulating through the list or else they all make requests on only one service in the group. Also, because the session manager does not force all members of a group to implement the same interface, you may want to have your clients check each service to see if it implements the correct interface by checking the reference's service name as shown in Example 33.

Example 33: *Checking the Service Reference for its Interface*

```
//C++
IT_Bus::Reference endpoint = response.getendpoints()[0];
if (endpoint.get_service_name() ==
    QName("", "AcmeService", "http://acme.com"))
  {
  // instantiate an AcmeService using endpoint
  }
else
  {
  // do something else
  }
```

Example 34 shows the client code for creating a proxy `acme` server from a group service.

Example 34: *Instantiate a Proxy Server*

```
// C++
AcmeClient acme_proxy(response.getendpoints()[0]);
```

Create a session header

Services that are being managed by the session manager will only accept requests that include a valid session header. The session header information is passed to the server as part of the proxy's input message attributes. Creating the session header and putting into the input message attributes takes three steps:

1. Set the proxy to use input message attributes.
2. Get a handle to the proxy's input message attributes.
3. Set the session information into the input message attributes.

Setting the proxy to use input message attributes

Artix client proxies all support a helper method, `get_port()`, which provides access to the port information used by the client to connect the service. One of an Artix proxy's port properties is `use_input_message_attributes`.

Setting this property to `true` tells the bus to ensure the input message attributes are propagated through to the server. Example 35 shows how to set the client proxy port's `use_input_message_attributes` property to `true`.

Example 35: *Use Input Message Attributes*

```
//C++
// Get the proxy's port
IT_Bus::Port proxy_port = acme_proxy.get_port();

// set the port property
proxy_port.use_input_attributes(true);
```

Getting a handle to the input message attributes

A pointer to the proxy port's input message attributes is returned by the port's `get_input_message_attributes()` method. Example 36 shows how to get a handle to the input message attributes.

Example 36: *Getting the Input Message Attributes*

```
MessageAttributes& input_attributes =
    proxy_port().get_input_message_attributes();
```

Setting the session information into the input message attributes

There are two attributes that must be set to include the correct session information in the input message:

`SessionName` Specifies the name the session manager has given this session. The session manager endpoints in the group will also be given this name to validate session header's against. The session name is returned by invoking `getname()` of the session ID of the active session.

`SessionGroup` Specifies the name the session manager has given this session. The session manager endpoints in the group will also be given this name to validate session header's against. The session name is returned by invoking `getname()` of the session ID of the active session.

The input message attributes are set using the message attribute handle's `set_string()` method. `set_string()` takes two attributes. The first is a string specifying the name of the attribute being set. The second is the value to be set for the attribute. Example 37 shows how to set the session information in to the input message attributes.

Example 37: *Setting the Input Message Attributes*

```
// C++
input_attributes.set_string("SessionName", session.getname());
input_attributes.set_string("SessionGroup",
                            session.getendpoint_group());
```

Make requests on service proxy

When the session information is added to the proxy's port information, the client can invoke operations on the client as it would a non-managed service. If the endpoint rejects the request because the client's session is not valid, an exception is raised.

Renew the session

If a client is going to use a session for a longer than the duration the session was granted, the client must renew its session or the session times out. A session is renewed using the session manager proxy's `renew_session()` method:

```
void renew_session(IT_Bus_Services::RenewSession params,
                   IT_Bus_Services::RenewSessionResponse renewed);
```

`params` Contains the session ID of the session being renewed and the duration, in seconds, of the renewal. The session ID is set using the `params`'s `setsession_id()` method. The renewal duration is set using the `params`'s `setrenew_timeout()` method.

`renewed` If the renewal is successful, `renewed` returns containing the duration of the renewal. The returned duration may be different if the requested renewal duration was outside of the configured range for session timeouts.

If the renewal is unsuccessful, an `IT_Bus_Services::renewSessionFaultException` is raised.

Example 38 shows how to end a session.

Example 38: *Ending a Session*

```
//C++
IT_Bus_Services::RenewSession params;
IT_Bus_Services::RenewSessionResponse renewed;
params.setsession_id(session);
parames.setrenewal_timeout(600);
try
{
  session_mgr.renew_session(params, renewed);
}
catch (IT_Bus_Services::renewSessionFaultException)
{
  // handle the exception
}
```

End the session

When a client is finished with a session managed service, it should explicitly end its session. This ensures that the session is freed up immediately. A session is ended using the session manager proxy's `end_session()` method:

```
void end_session(IT_Bus_Services::EndSession params);
```

`params` contains the session ID of the session being ended. The session ID is set using `params`'s `setsession_id()` method.

Example 39 shows how to end a session.

Example 39: *Ending a Session*

```
//C++
IT_Bus_Services::EndSession params;
params.setsession_id(session);
session_mgr.end_session(params);
```

For more information on writing Artix client code, see the *Artix C++ Programmer's Guide*.

Fault Tolerance

Overview

Enterprise level deployments demand that applications can cleanly recover from occasional failures. The Artix session manager is designed to recover from the two most common failures:

- Failure of a registered endpoint.
- Failure of the session manager itself.

Endpoint failure

When an endpoint gracefully shuts down, it notifies the session manager that it is no longer available. The session manager removes the endpoint from its list so it cannot give a client a reference to a dead endpoint. However, when an endpoint fails unexpectedly, it cannot notify the session manager and the session manager can unknowingly give a client an invalid reference causing the failure to cascade.

To mitigate the risk of passing invalid references to clients, the session manager occasionally pings all of its registered endpoint managers to see if they are still running. If an endpoint manager does not respond to a ping, the session manager removes that endpoint manager's references.

You can adjust the interval between session manager pings by setting the configuration variable `plugins:session_manager:peer_timeout`. The default setting is `4` seconds. For more information see "Configuring Artix" on page 23.

Service failure

When the session manager fails all of the references to the registered services are lost and the active services are no longer be registered. To ensure that the active services reregister with the session manager when it restarts, the endpoint managers, after the session manager has missed its ping interval, will periodically attempt to reregister with the session manager until they are successful.

You can adjust the interval between the endpoint manager's pings of the session manager by setting the configuration variable `plugins:session_endpoint_manager:peer_timout`. The default setting is 4 seconds. For more information see "Configuring Artix" on page 23.

CHAPTER 11

Deploying a Service Chain

Artix provides a chain builder that enables you to create a series of services to invoke as part of a larger process.

In this chapter

This chapter includes the following sections:

The Artix Chain Builder

Overview

The Artix Chain Builder enables you to link together a series of services into a multi-part process. This is useful if you have processes that require a set order of steps to complete, or if you wish to link together a number of smaller service modules into a complex service.

Chaining services together

For example, you may have a four services that you wish to combine to service requests from a single client. You can deploy a service chain like the one shown in Figure 15.

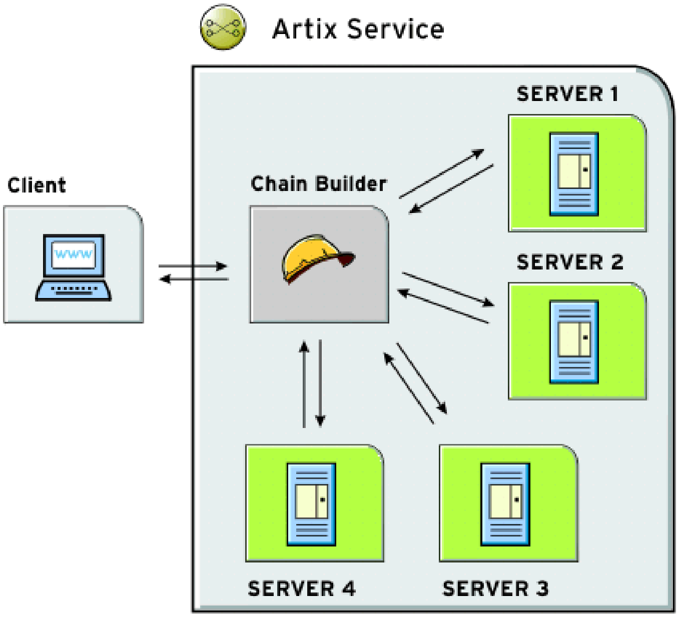

Figure 15: *Chaining Four Servers to Form a Single Service*

In this scenario, the client makes a single request and the chain builder dispatches the request along the chain starting at `Server1`. The chain builder takes the response from `Server1` and passes that to the next endpoint in the chain, `Server2`. This continues until the end of the chain is reached at `Server4`. The chain builder then returns the finished response to the client.

The chain builder is implemented as an Artix plug-in so it can be deployed into any Artix process. The decision about which process that you deploy it in depends on the complexity of your system, and also how you choose to allocate resources for your system.

Assumptions

To make the discussion of deploying the chain builder as straightforward as possible, this chapter assumes that you are deploying it into an instance of the Artix standalone service. However, the configuration steps for configuring and deploying a chain builder are the same no matter which process you choose to deploy it in.

Configuring the Artix Chain Builder

Overview

To configure the Artix Chain Builder, complete the following steps:

1. Add the chain builder's plug-in to the process' `orb_plugins` list.
2. Configure all of the endpoints that are a part of the chain as generic Artix endpoints.
3. Configure the chain so that it knows what servants to instantiate and the service chain for each operation implemented by the servant.

Adding the chain builder in the orb_plugins list

Configuring the application to load the chain builder's plug-in requires adding it to the application's `orb_plugins` list. The plug-in name for the chain builder is `ws_chain`. Example 40 shows an `orb_plugins` list for a process hosting the chain builder.

Example 40: *Plug-in List for Using a Web Service Chain*

```
orb_plugins={"ws_chain", "xml_log_stream"};
```

Configuring the endpoints in the chain

Each service that is a part of the chain and the client that makes requests through the chain service must be configured as generic Artix endpoints in the chain builder's configuration scope. This provides the chain builder with the necessary information to instantiate a servant that the client can make requests against. It also supplies the information needed to make calls to the services that make up the chain.

To configure a group of Artix endpoints, use the configuration variables in the `artix:endpoint` namespace. These variables are described in Table 15.

Table 15: *Artix Endpoint Configuration*

Variable	Function
`artix:endpoint:endpoint_list`	Specifies a list of the endpoints and their names for the current configuration scope.
`artix:endpoint:`*`endpoint_name`*`:wsdl_location`	Specifies the location of the contract describing this endpoint.

Table 15: *Artix Endpoint Configuration*

Variable	Function
`artix:endpoint:`*`endpoint_name`*`:service_namespace`	Specifies the XML namespace of the service that this endpoint implements.
`artix:endpoint:`*`endpoint_name`*`:service_name`	Specifies the name, from the WSDL `<service>` element, of the service this endpoint implements.
`artix:endpoint:`*`endpoint_name`*`:port_name`	Specifies the name, from the WSDL `<port>` element, of the port that this endpoint can be contacted on.

Configuring the service chains

The chain builder requires you to provide the following details

- A list of endpoints that are clients to the chain builder.
- A list of operations that each client can invoke.
- Service chains for each operation that the clients can invoke.

Specifying the servant list

The first configuration setting tells the chain builder how many servants to instantiate, the interfaces that the servants must support, and the physical details of how the servants are contacted. You specify this using the variable `plugins:chain:servant_list`. `plugins:chain:servant_list`. This takes a list of endpoint names from the list of Artix endpoints that you defined earlier in the configuration scope.

Specifying the operation list

The second part of the chain builder's configuration is a list of the operations that each client to the chain builder can invoke. You specify this using `plugins:chain:`*`endpoint`*`:operation_list` where *`endpoint`* refers to one of the endpoints in the chain's service list.
`plugins:chain:`*`endpoint`*`:operation_list` takes a list of the operations that are defined in `<operation>` tags in the endpoint's contract. You must list all of the operations for the endpoint or an exception will be thrown at runtime. You must also be sure to enter a list of operations for each endpoint specified in the chain's service list.

Specifying the service chain

The third piece of the chain builder's configuration is to specify a service chain for every operation defined in the endpoints listed in `plugins:chain:servant_list`. This is specified using the `plugins:chain:`*`endpoint`*`:`*`operation`*`:service_chain` configuration variable. The syntax for entering the service chains is shown in Example 41.

Example 41: *Entering a Service Chain*

```
plugins:chain:endpoint:operation:service_chain=["op1@endpt1", "op2@endpt2", ..., "opN@endptN"];
```

For each entry, the syntax is as follows:

`endpoint` Specifies the name of an endpoint from the chain builder's servant list

`operation` Specifies one of the operations defined by an `<operation>` entry in the endpoints contract. The entries in the list refer to operations implemented by other endpoints defined in the configuration.

`opN` Specifies one of the operations defined by an `<operation>` entry in the contract defining the service specified by *`endptN`*. The operations in the service chain are invoked in the order specified. The final result is returned back to the chain builder which then responds to the client.

Configuration example

Example 42 shows the contents of a configuration scope for a process that hosts the chain builder.

Example 42: *Configuration for Hosting the Artix Chain Builder*

```
colaboration
{
  orb_plugins = ["ws_chain"];

  artix:endpoint:endpoint_list = ["customer", "pm", "designer",
   "builder"];

```

Example 42: *Configuration for Hosting the Artix Chain Builder*

```
   artix:endpoint:customer:wsdl_location = "order.wsdl";
   artix:endpoint:customer:service_namespace =
    "http://needs.com";
   artix:endpoint:customer:service_name = "POC";
   artix:endpoint:customer:port_name = "order_port";

   artix:endpoint:pm:wsdl_location = "manager.wsdl";
   artix:endpoint:pm:service_namespace = "http://ORBSrUs.com";
   artix:endpoint:pm:service_name = "prioritize";
   artix:endpoint:pm:port_name = "pm_port";

   artix:endpoint:designer:wsdl_location = "designer.wsdl";
   artix:endpoint:designer:service_namespace =
    "http://ORBSrUs.com";
   artix:endpoint:transformer:service_name = "design";
   artix:endpoint:transformer:port_name = "desinger_port";

   artix:endpoint:builder:wsdl_location = "engineer.wsdl";
   artix:endpoint:builder:service_namespace =
    "http://ORBSrUs.com";
   artix:endpoint:builder:service_name = "produce";
   artix:endpoint:builder:port_name = "builder_port";

   plugins:chain:servant_list = ["customer"];
   plugins:chain:oldClient:operation_list = ["requestSolution"];
   plugins:chain:customer:requestSolution:service_chain =
     ["estimatePriority@pm", "makeSpecification@designer",
      "buildORB@builder"];
};
```

CHAPTER 12

Deploying the Artix Transformer

Artix provides an XSLT transformer service that can be configured to run as a servant process that replaces an Artix server.

In this chapter

This chapter discusses the following topics:

The Artix Transformer

Overview

The Artix transformer provides a means of processing messages without writing application code. The transformer processes messages based on XSLT scripts and returns the result to the requesting application. XSLT stands for *Extensible Stylesheet Language Transformations*.

These XLST scripts can perform message transformations, such as concatenating two string fields, reordering the fields of a complex type, and truncating values to a given number of decimal places. XSLT scripts can also be used to validate data before passing it onto a Web service for processing, and a number of other applications.

Deployment Patterns

The Artix transformer is implemented as an Artix plug-in. Therefore, it can be loaded into any Artix process. This makes it extremely flexible in how it can be deployed in your environment. If the speed of calls or security is an issue, the transformer can be loaded directly into an application. If you need to spread resources across a number of machines, the transformer plug-in can be loaded in a separate process.

There are two main patterns for deploying the Artix Transformer:

- Standalone deployment
- Deployment as part of a chain

Standalone deployment

The first pattern is to deploy the transformer by itself. This is useful if your application is doing basic data manipulation that can be described in an XSLT script. The transformer replaces the server process and saves you the cost of developing server application code. This style of deployment can also be useful for performing data validation before passing requests to a server for processing.

The most straightforward way to deploy the transformer is to deploy it as a separate servant process hosted by the Artix standalone service. When deployed in this way the transformer receives requests from a client, processes the message based on supplied XSLT scripts, and replies with the results of the script. In this configuration, shown Figure 16, the transformer becomes the server process in the Artix solution.

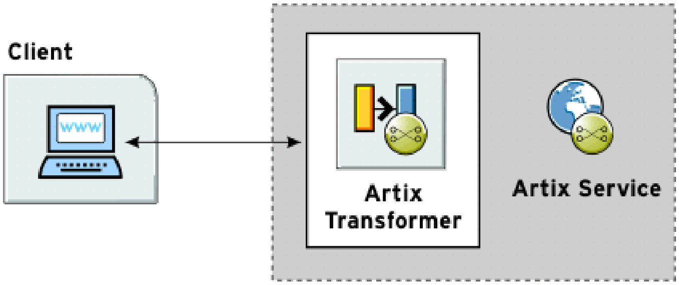

Figure 16: *Artix Transformer Deployed as a Servant*

You can modify the deployment pattern shown in Figure 16 by eliminating the Artix standalone service and having your client directly load the transformer's plug-in as shown in Figure 17. This saves the overhead of making calls outside of the client process to reach the transformer However, it can reduce the overall efficiency of your system if the transformer requires a large amount of resources to perform its work.

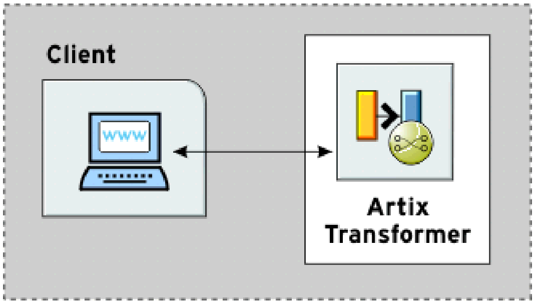

Figure 17: *Artix Transformer Loaded by Client*

Deployment as part of a chain

The second pattern is to deploy the Artix transformer as part of a Web service chain controlled by the Web Service Chain Builder. This deployment is useful if you need to connect legacy clients to updated servers whose interfaces may have changed or are connecting applications that have different interfaces. It can also be useful for a range of applications where data transformation is needed as part of a larger set of business logic.

Figure 18 shows an example of this type of deployment where the transformer and the chain builder are both hosted by the Artix standalone service. The chain builder directs the requests to the transformer which transforms messages. When the transformer returns the processed data, the chain builder then passes it onto the server.

In this example, the server returns the results to the client without further processing, but the results can also be passed back through the transformer. Neither the client nor the server need to be aware of the processing.

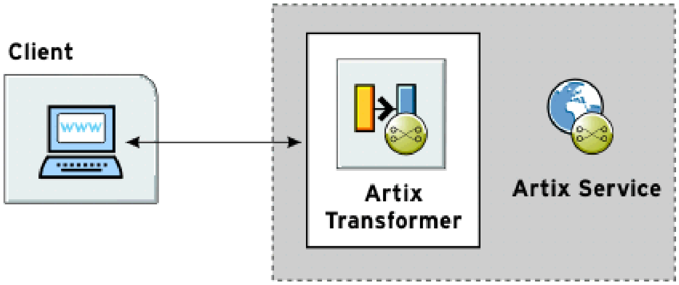

Figure 18: *Artix Transformer Deployed with the Chain Builder*

You could modify this deployment pattern in a number of ways depending on how you wish to allocate your resources. For example, you can configure the client process to load the chain builder and the transformer. You can also have the chain builder and the transformer loaded into separate processes.

Standalone Deployment

Overview

To deploy an instance of the Artix transformer you must first decide what process is hosting the transformer's plug-in. You must then add the following to the process configuration scope:

- The transformer plug-in, `xslt`.
- An Artix endpoint configuration to represent the transformer.
- The transformer's configuration information.

Updating the orb_plugins list

Configuring the application to load the transformer requires adding it to the application's `orb_plugins` list. The plug-in name for the transformer is `xslt`. Example 43 shows an `orb_plugins` list for a process hosting the transformer.

Example 43: *Plug-in List for Using XSLT*

```
orb_plugins={"xslt", "xml_log_stream"};
```

Adding an Artix endpoint definition

The transformer is defined as a generic Artix endpoint. To instantiate it as a servant, Artix must know the following details:

- The location of the Artix contract that defines the transformer's endpoint.
- The interface that the endpoint implements.
- The physical details of its instantiation.

This information is configured using the configuration variables in the `artix:endpoint` namespace. These variables are described in Table 16.

Table 16: *Artix Endpoint Configuration*

Variable	Function
`artix:endpoint:endpoint_list`	Specifies a list of the endpoints and their names for the current configuration scope.
`artix:endpoint:`*`endpoint_name`*`:wsdl_location`	Specifies the location of the contract describing this endpoint.

Table 16: *Artix Endpoint Configuration*

Variable	Function
`artix:endpoint:`*`endpoint_name`*`:service_namespace`	Specifies the XML namespace of the service that this endpoint implements.
`artix:endpoint:`*`endpoint_name`*`:service_name`	Specifies the name, from the WSDL `<service>` element, of the service this endpoint implements.
`artix:endpoint:`*`endpoint_name`*`:port_name`	Specifies the name, from the WSDL `<port>` element, of the port on which this endpoint can be contacted.

Configuring the transformer

Configuring the transformer involves two steps that enable it to instantiate itself as a servant process and perform its work.

- Configuring the list of servants.
- Configuring the list of scripts.

Configuring the list of servants

The name of the endpoints that will be brought up as transformer servants is specified in `plugins:xslt:servant_list`. The endpoint identifier is one of the endpoints defined in `artix:endpoint:endpoint_list` entry. The transformer uses the endpoint's configuration information to instantiate the appropriate servants

Note: `artix:endpoint:endpoint_list` must be specified in the same configuration scope.

Configuring the list of scripts

The list of the XSLT scripts that each servant uses to process requests is specified in `plugins:xslt:`*`endpoint_name`*`:operation_map`. Each endpoint specified in the servant list has a corresponding operation map entry in the configuration. The operation map is specified as an ordered list using the syntax shown in Example 44.

Example 44: *Operation Map Syntax*

```
plugins:xslt:endpoint_name:operantion_map = ["wsdlOp1@filename1"
    , "wsdlOp2@filename2", ..., "wsdlOpN@filenameN"];
```

Each entry in the map specifies a logical operation that is defined in the service’s contract by an <operation> element and the XSLT script to run when a request is made on the operation. You must specify an XSLT script for every operation defined for the endpoint. If you do not, the transformer raises an exception when the unmapped operation is invoked.

Configuration example

Example 45 shows the configuration scope of an Artix application, transformer, that loads the Artix Transformer to process messages. The transformer is configured as an Artix endpoint named hannibal and the transformer uses the endpoint information to instantiate a servant to handle requests.

Example 45: *Configuration for Using the Artix Transformer*

```
transformer
{
orb_plugins = ["local_log_stream","xslt"];

artix:endpoint:endpoint_list = ["hannibal"];

artix:endpoint:hannibal:wsdl_location = "transformer.wsdl";
artix:endpoint:hannibal:service_namespace = "http://transformer.com/xslt";
artix:endpoint:hannibal:service_name = "WhiteHat";
artix:endpoint:hannibal:port_name = "WhitePort";

plugins:xslt:servant_list=["hannibal"]
plugins:xslt:hannibal:operation_map = ["op1@../script/op1.xsl", "op2@../script/op2.xsl",
   "op3@../script/op3.xsl"]
}
```

Deployment as Part of a Chain

Overview

Deploying the Artix Transformer as part of Web service chain allows you to use it as part of an integration solution without needing to necessarily modify your applications. The Artix Web Service Chain Builder facilitates the placement of the transformer into a series of Web service calls managed by Artix.

The plug-in architecture of the transformer and the chain builder allow for you to deploy this type of solution in a variety of ways depending on what is the best fit for your particular solution. The most straightforward way to deploy this type of solution is to deploy both the transformer and the chain builder into the same process. This is the deployment that will be used to outline the steps for configuring the transformer to be deployed as part of a Web service chain. In general, you will need to complete all of the same steps regardless of how you choose to deploy your solution.

Procedure

To deploy the transformer as part of a Web service chain you need to complete the following steps:

1. Modify your process's configuration scope to load the transformer and the chain builder.
2. Configure Artix endpoints for each of the applications that will be part of the chain.
3. Configure an Artix endpoint to represent the transformer.
4. Configure the transformer.
5. Configure the service chain to include the transformer at the appropriate place in the chain.

Updating the orb_plugins list

Configuring the application to load the transformer plug-in and the chain builder plug-in requires adding them to the process's `orb_plugins` list. The plug-in name for the transformer is `xslt` and the plug-in name for the chain builder is `ws_chain`. Example 46 shows an `orb_plugins` list for a process hosting the transformer and the chain builder.

Example 46: *Loading the Artix Transformer as Part of a Chain*

```
orb_plugins={"xslt", "ws_chain", "xml_log_stream"};
```

Configuring the endpoints in the chain

The Artix Web Service Chain Builder uses generic Artix endpoints to represent all of the applications in a chain, including the transformer. Table 16 on page 175 shows the configuration variables used to configure a generic Artix endpoint.

Configuring the transformer

The transformer requires the same configuration information regardless of how it is deployed. You must provide it with the name of the endpoints it will instantiate from the list of endpoints and provide each instantiation with an operation map. For more information about providing this information see "Configuring the transformer" on page 176.

Placing the transformer in the chain

The chain builder instantiates a servant for each endpoint specified in its servant list. Each servant can have a multiple operations. For each operation that will be involved in a Web service chain, you need to specify an ordered list of endpoints and their operations that make up the chain. This list is specified using `plugins:chain:`*`endpoint_name`*`:`*`operation_name`*`:service_chain`.

To include the transformer in one of the chains, you add the appropriate operation and endpoint names for the transformer at the appropriate place in the service chain.

For more information on configuring the chain builder see "Deploying a Service Chain" on page 163.

Configuration example

Example 47 shows a configuration scope that contains configuration information for deploying the transformer as part of a Web service chain.

Example 47: *Configuration for Deploying the Artix Transformer in a Web Service Chain*

```
transformer
{
  orb_plugins = ["ws_chain", "xslt"];

  artix:endpoint:endpoint_list = ["oldClient", "newServer",
   "transformer"];

  artix:endpoint:oldClient:wsdl_location = "bank.wsdl";
  artix:endpoint:oldClient:service_namespace =
   "http://bank.com";
  artix:endpoint:oldClient:service_name = "ATM";
  artix:endpoint:oldClient:port_name = "client_port";

  artix:endpoint:newServer:wsdl_location = "bank.wsdl";
  artix:endpoint:newServer:service_namespace =
   "http://bank.com";
  artix:endpoint:newServer:service_name = "newATM";
  artix:endpoint:newServer:port_name = "server_port";

  artix:endpoint:transformer:wsdl_location = "bank.wsdl";
  artix:endpoint:transformer:service_namespace =
   "http://bank.com";
  artix:endpoint:transformer:service_name = "transformer";
  artix:endpoint:transformer:port_name = "transformer_port";

  plugins:xslt:servant_list = ["transformer"];
  plugins:xslt:transformer:operation_map =
   ["transform@transformer.xsl"];

  plugins:chain:servant_list = ["oldClient"];
  plugins:chain:oldClient:operation_list = ["withdraw"];
  plugins:chain:oldClient:client_operation:service_chain =
   ["transform@transformer", "withdraw@newServer"];
};
```

Part IV

Integrating with Other Middleware Systems

In this part

This part contains the following chapters:

CHAPTER 13

Using Artix in a CORBA Environment

Artix can be run inside an existing CORBA environment and leverage a number of its services.

In this chapter

This chapter discusses the following topics:

Embedding Artix in a CORBA Application

Overview

Because Artix is built on IONA's flexible ART platform, it can be embedded in any CORBA application implemented using IONA's Orbix 6.0 or later without modifying any CORBA application's code. You can embed Artix by updating the application configuration to load the required Artix plug-ins.

Embedding Artix in your CORBA application has several advantages:

- No need for a separate process to route messages to the non-CORBA parts of your application.
- Improved messaging performance over using the Artix standalone service.
- Write your code using a familiar paradigm and realize the benefits of using Artix.
- Leverage all of the CORBA infrastructure to provide enterprise-level qualities of service and management.

Embedding Artix in a CORBA client

To embed Artix in a CORBA client application, do the following:

1. Create an Artix contract that fully describes the interfaces, bindings, transports, and routing rules used in your Artix application.
2. Edit your CORBA client's configuration scope so that the ORB plug-ins list contains the required Artix plug-ins to support the bindings and transports used by your Artix application.

   For example, if your CORBA client interacts with a sever using SOAP over WebSphere MQ, your ORB plug-in list would be similar to Example 48 on page 185. The required Artix plug-ins are highlighted.
3. Make an entry for `plugins:routing:wsdl_url` that specifies where the Artix application's contract resides.

In Example 48, the Artix contract describing the application is stored in /artix/wsdlRepos/scoreBox.wsdl.

Example 48: *Embedded Artix orb_plugins list*

```
corba_client.artix
{
  orb_plugins=["iiop_profile", "giop", "soap", "mq", "ws_orb",
   "routing"];
  plugins:routing:wsdl_url="/artix/wsdlRepos/scoreBox.wsdl";
}
```

4. When you start your CORBA client, ensure that you start it using the correct ORB name to load the Artix plug-ins.

   For a client that uses the configuration in Example 48, you would start the client with the following command:

   ```
   client -ORBname corba_client.artix
   ```

Embedding Artix in a CORBA server

To embed Artix in a CORBA server that uses the routing plug-in, you must first ensure that:

- Your CORBA server generates persistent object references.
- Your CORBA server runs one time to export the persistent references, and is then restarted for the Artix routing plug-in to work.

The routing plug-in requires valid object references to properly load itself. When embedded in the CORBA server, the routing plug-in is loaded by the ORB before any object references are generated. By using persistent object references and pregenerating them before fully deploying the server, as when using the naming service, you satisfy the routing plug-in.

To configure a CORBA server to embed Artix, complete the following steps:

1. Create an Artix contract that fully describes the interfaces, bindings, transports, and routing rules used in your Artix application.
2. Edit the CORBA server's configuration scope so that the ORB plug-ins list contains the required Artix plug-ins to support the bindings and transports used by your Artix application.

For example, if your CORBA server interacts with a client using SOAP over WebSphere MQ, your ORB plug-in list would be similar to Example 49. The required Artix plug-ins are highlighted.

3. Make an entry for `plugins:routing:wsdl_url` that specifies where the Artix applications contract resides.

   In Example 49, the Artix contract describing the application is stored in `/artix/wsdlRepos/scoreBox.wsdl`.

4. Edit the server's client binding list, `binding:client_binding_list`, so that none of the listed bindings use `POA_Coloc`.

   The configuration scope in Example 49 shows a client binding list that does not use `POA_Coloc`. The default client binding list includes entries for `"OTS+POA_Coloc"` and `"POA_Coloc"`.

Example 49: *Embedded Artix Server Configuration*

```
corba_server.artix
{
  orb_plugins=["iiop_profile", "giop", "soap", "mq", "ws_orb",
   "routing"];
  plugins:routing:wsdl_url="/artix/wsdlRepos/scoreBox.wsdl";
  binding:client_binding_list=[“OTS+GIOP+IIOP”, “GIOP+IIOP”];
  binding:server_binding_list=["OTS"];
}
```

5. When you start your CORBA server ensure that you start it using the correct ORB name to load the Artix plug-ins.

   For a server that uses the configuration in Example 49, you would start the client with the following command:

```
server -ORBname corba_server.artix
```

Using the CORBA Naming Service

Overview

To fully integrate with deployed CORBA systems, Artix can use a CORBA naming service that supports the `CosNaming` interface. This requires editing the port information in the service's contract and modifying the Artix configuration.

Specifying servers

To specify that an Artix instance (acting as proxy for a server) is to use the CORBA naming service, edit the `<corba:address>` element of the CORBA port. In place of the file name in the `location` attribute, specify a `corbaname`.

For example, to specify that the `converter` server publishes its IOR to the CORBA naming service, specify the `<corba:address>` as follows:

```
<corba:address location="corbaname:rir:/NameService#personalInfoService"/>
```

This registers the server in the name service under the name `personalInfoService`.

Specifying clients

An Artix instance (acting as a proxy for a client) can also use the `<corba:address>` element to specify what name to look up in the CORBA name service. The name that the client looks up in the name service is the string after the `#` in the specified location.

For example, a client using the `<corba:address>` shown above in "Specifying servers" looks up the IOR for an object named `personalInfoService`.

Configuring Artix

Artix applications that wish to use a CORBA name service must be configured to load a name resolver plug-in and have an initial reference for the running name service.

To modify the Artix configuration, do the following:

1. Open the Artix configuration file in a text editor: *install-dir*`\artix\2.0\etc\artix.cfg`,.
2. In the global scope, add the following lines:

```
initial_references:NameService:reference="corbaloc::localhost:portNumber/NameService";
url_resolvers:corbaname:plugin="naming_resolver";
plugins:naming_resolver:shlib_name="it_naming";
```

`portNumber` is the number of the port on which the name service is running.

For more information on configuring Artix, see “Configuring Artix” on page 23.

Load Balancing with CORBA

Overview

If an Artix Service Access Point (SAP) is mapped to a CORBA service, and that CORBA service is accessible using IONA's Orbix 6.0 SP 1 (or later), the implementation of that service can be load balanced using the Orbix locator service. To accomplish this, the Artix configuration file must duplicate some of the information from the Orbix configuration domain, as described in this section.

For information on the Orbix load balancing features, see the *Orbix Administrator's Guide*.

Configuration Steps

The following steps work with an Orbix installation that uses either file-based configuration or a configuration repository. However, because Artix supports only file-based configuration, the relevant configuration information must be inserted into the `artix.cfg` file. The following configuration example assumes that an Orbix domain exists, and that the locator service is run from this domain:

1. From the `*orbix-domain-name*.cfg` file, get the following configuration information, and add it to `artix.cfg` file.

```
initial_references:IT_NodeDaemon:reference =
   "IOR:000000000000002149444c3a49545f4e6f64654461656d6f6e2f4e6f6465446165
   6d6f6e3a312e30000000000000000100000000000000760001020000000008686f72617
   4696f00782800000000001d3a3e0233310c6e6f64655f6461656d6f6e000a4e6f646544
   61656d6f6e0000000000000300000001000000180000000000010001000000000001010
   400000001000101090000001a000000040100000000000006000000060000000000011";
```

2. Create an `ORBname` for each Artix SAP that participates in load balancing. For example:

```
itadmin orbname create demos.clustering.server_1
itadmin orbname create demos.clustering.server_2
itadmin orbname create demos.clustering.server_3
```

3. Create a Portable Object Adapter (POA) that declares these ORBnames as replicas, and specify either round-robin or random load balancing. For example:

```
itadmin poa create -replicas
   demos.clustering.server_1,demos.clustering.server_2,demos.clustering.server_3
   -load_balancer round_robin ClusterDemo
```

The POA name (ClusterDemo) is expressed in WSDL as:

```
<corba:policy persistent="true" serviceid="service_id" poaname="ClusterDemo"/>
```

You can choose any POA name; however, the POA name you register using itadmin must be the same name you declare in the WSDL file.

When corba:policy persistent=true is specified, you must also specify serviceid. Failure to specify serviceid either results in an IOR that cannot be used for load balancing, or a process that outlives the POA.

4. To run ClusterDemo, you would start the CORBA servers that underlie the Artix SAP as follows:

```
Server -ORBname demos.clustering.server_1
Server -ORBname demos.clustering.server_2
Server -ORBname demos.clustering.server_3
```

When you run a client to connect to the Artix SAP, the first request goes to the first server (because round_robin load balancing was declared). If a second client is started, its request goes to the second server, and a third client's request goes to the third server.

Replicated Orbix services

If your Orbix services are replicated, and if Artix is deployed on each of the machines that those services are replicated on, the Artix SAPs themselves can be replicated and load-balanced. For example:

1. On the *master* machine (the machine that hosts the configuration repository), create an ORBname for each Artix SAP that participates in load balancing. For example:

```
itadmin orbname create demos.clustering.server_1
itadmin orbname create demos.clustering.server_2
itadmin orbname create demos.clustering.server_3
```

2. Create a POA that declares these ORBnames as replicas, and specify either round-robin or random load balancing. For example:

```
itadmin poa create -replicas
   demos.clustering.server_1,demos.clustering.server_2,demos.clustering.server_3
   -load_balancer round_robin ClusterDemo
```

3. On each machine that replicates the service, obtain the node daemon's initial reference, and add it to the artix.cfg file on that machine.
4. Start a server on each machine, passing one of the three specified ORBnames to it:

```
clustering.server_1,
demos.clustering.server_2
demos.clustering.server_3
```

This service is now load balanced among the three replicated Artix SAPs. If one or two of these SAPs is killed, the client invocation is directed to the remaining machine(s).

Creating the load-balanced environment dynamically

You can create a load balance environment without creating the POA or manually registering ORB names. To accomplish this:

1. On the master machine, obtain the node daemon initial reference and put it in the artix.cfg file.
2. Start the CORBA service, passing the same ORB name as that specified in the Artix client's WSDL contract. This ORB name is received by the Node Daemon, which creates a POA with that name. If you do not specify an ORB name, the name WSORB is used.
3. On the master machine, issue the following command in the Orbix environment with the name you chose:

```
itadmin poa modify -allowdynreplicas yes POA_Name
```

4. On each of the slave machines where the service is replicated, obtain the node daemon initial reference from the Orbix domain configuration and put it in the artix.cfg file.
5. On each slave machine where the service is replicated, start the CORBA service, using a *different* ORBname each time.

6. On the master machine, issue the following command in the Orbix environment (inserting the type of load balancing and the `ORBnames` you have chosen):

```
itadmin poa modify -l <round_robin | random> POA_name
```

7. Start the Artix SAP.

Other load balancing features

In addition to POA name, the Orbix configuration file can also affect load balancing by specifying:

- Persistent or Transient POA policy
- Object ID

These load-balancing-related configuration values can be specified in an Artix WSDL contract using WSDL extensions for CORBA ports:

Specifying the POA name

The POA name can be specified as follows:

```
<corba:policy poaname="my_poa_name"/>
```

The default POA name is `WSORB`.

Specifying the POA Persistence policy

The POA persistence policy can be set as follows:

```
 <corba:policy persistent="true | false"/>
```

If this value is set to `true`, the POA policy is persistent. The default persistence value is `false`.

Specifying the Service ID

The Service ID can be set as follows:

```
<corba:policy serviceid="ncname"/>
```

Specifying the Object ID

Object ID is provided by the POA if the POA Policy `SYSTEM_ID` is set. Setting this to any string sets the POA policy `USER_ID` and uses the value provided as the `object_id`. If this is not set, the POA policy is `SYSTEM_ID`.

Examples

The following WSDL examples illustrate these additional load balancing features.

Setting the persistent POA policy

The contract fragment in Example 50 results in the following POA policy settings:

- PERSISTENT
- USER_ID
- POAName="master1"
- ObjectID="master1"

Example 50: *Setting the PERSISTENT POA policy*

```
<service name="BaseService">
  <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
    <corba:address location="file://master.ref"/>
    <corba:policy persistent="true" poaname="master1" serviceID="master1"/>
  </port>
</service>
```

Setting the POA name policy

The contract fragment in Example 51 results in the following POA policy settings:

- TRANSIENT (Default)
- SYSTEM_ID (Default)
- POAName="master1"

Example 51: *Setting the POAName POA policy*

```
<service name="BaseService">
  <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
    <corba:address location="file://master.ref"/>
    <corba:policy poaname="master1"/>
  </port>
</service>
```

Setting the User ID policy

The contract fragment in Example 52 results in a POA with the following policy settings:

- TRANSIENT (Default)
- USER_ID
- POAName="WSORB" (Default)
- ObjectID="master1"

Example 52: *Setting the USER_ID POA policy*

```
<service name="BaseService">
   <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
       <corba:address location="file://master.ref"/>
       <corba:policy poaname="master1" serviceID="master1"/>
   </port>
</service>
```

Setting the default policy

The contract fragment in Example 53 results in a POA with all default policies.

Example 53: *Default POA policies*

```
<service name="BaseService">
  <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
    <corba:address location="file://master.ref"/>
  </port>
</service>
```

CHAPTER 14

Embedding Artix in a BEA Tuxedo Container

Artix can be run and managed by BEA Tuxedo like a native Tuxedo application.

In this chapter

This chapter includes the following sections:

Introduction

Overview

To enable Artix to interact with native BEA Tuxedo applications, you must embed Artix in the Tuxedo container.

At a minimum, this involves adding information about Artix in your Tuxedo configuration file, and registering your Artix processes with the Tuxedo bulletin board.

In addition, you can also enable to Tuxedo bring up your Artix process as a Tuxedo server when running `tmboot`.

Note: BEA Tuxedo integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports Tuxedo integration.

This chapter explains these steps in detail.

Embedding an Artix Process in a Tuxedo Container

Procedure

To embed an Artix process in a Tuxedo container, complete the following steps:

1. Ensure that your environment is correctly configured for Tuxedo.
2. Add the Tuxedo plug-in, `tuxedo`, to your Artix process's `orb_plugins` list. See "ORB Plug-ins" on page 37.

   ```
   orb_plugins=["iiop_profile", "giop", "iiop", "tuxedo"];
   ```

3. Set `plugins:tuxedo:server` to `true` in your Artix configuration scope.
4. Ensure that the executable for your Artix process is placed in the directory specified in the `APPDIR` entry of your Tuxedo configuration.
5. Edit your Tuxedo configuration's `SERVERS` section to include an entry for your Artix process.

   For example, if the executable of your Artix process is `ringo`, add the following entry in the `SERVERS` section:

   ```
   ringo SVRGRP=BEATLES SVRID=1
   ```

   This associates `ringo` with the Tuxedo group called `BEATLES` in your configuration and assigns `ringo` a server ID of 1. You can modify the server's properties as needed.
6. Edit your Tuxedo configuration's `SERVICES` section to include an entry for your Artix process.

   While standard Tuxedo servers only require a `SERVICES` entry if you are setting optional runtime properties, Artix servers in the Tuxedo container require an entry, even if no optional runtime properties are being set. The name entered for the Artix process is the name specified in the `serviceName` attribute of the Tuxedo port defined in the Artix contract for the process.

For example, given the port definition shown in Example 54, the SERVICES entry would be personalInfoService.

Example 54: *Sample Service Entry*

```
<service name="personalInfoService">
   <port name="tuxInfoPort" binding="tns:personalInfoBinding">
       <tuxedo:server>
           <tuxedo:service name="personalInfoService"/>
       </tuxedo:server>
  </port>
</service>
```

7. If you made the Tuxedo configuration changes in the ASCII version of the configuration, UBBCONFIG, reload the TUXCONFIG with tmload.

When you have configured Tuxedo, it manages your Artix process as if it were a regular Tuxedo server.

CHAPTER 15

Integrating with Enterprise Java Beans

Artix can connect to Enterprise Java Beans and provide a means of exposing them throughout your environment.

In this chapter

This chapter includes the following sections

Artix EJB Integration

Overview

Many applications are developed using Java technology and J2EE Enterprise Java Beans (EJBs). While Java enables applications to run on a number of hardware platforms, it does not provide native interoperability with applications developed using a variety of other middleware platforms.

In addition, J2EE (prior to 1.4) does not provide a native means of exposing stateless EJBs as Web services. Artix provides tools that map stateless EJBs to WSDL and expose its functionality as a Web service or to applications using any of the transports supported by Artix.

Exposure by proxy

Artix exposes a stateless EJB by placing an Artix proxy in front it. This proxy communicates with the stateless EJB and handles the transformation of the incoming and outgoing messages. The proxy is implemented in a Java Artix server process that registers a special `EJBServant` object with the Artix bus. This special servant object does not require an implementation object because it makes requests on the stateless EJB's methods to fulfil client requests.

Instantiating an EJBServant

Example 55 shows the signature for the `EJBServant` constructor.

Example 55: *Instantiating an EJBServant*

```
public EJBServant(String wsdlLoc, Bus bus, String jndiName);
```

This method takes the following parameters:

`wsdlLoc` The location of the WSDL document describing the stateless EJB's interface and the ports over which it is being exposed.

`bus` A reference to an initialized Artix bus.

`jndiName` The JNDI name of the EJB's remote interface.

When the `EJBServant` is instantiated, it gets the stateless EJB's contact information using a JNDI look-up.

Example

Figure 19 shows how Artix exposes a stateless EJB to applications outside of the J2EE container. In this instance, Artix is connecting the stateless EJB to a Web service client communicating using SOAP over HTTP.

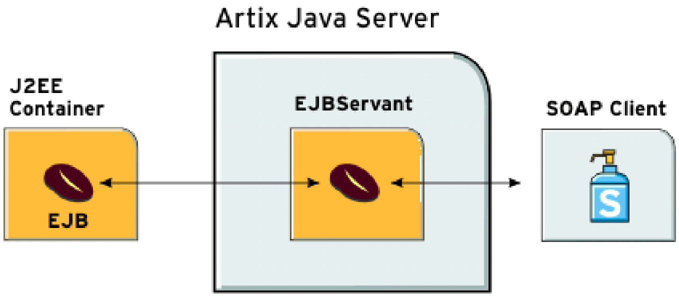

Figure 19: *Exposing an EJB*

Configuring an Artix EJB proxy to use JNDI

Overview

For your Artix EJB proxy to perform the JNDI look-up of a stateless EJB, it must have its initial context properties configured. You can do this by adding the entries shown Table 17 in to a file named `initial_context.properties`.

Table 17: *JNDI Initial Context Properties*

Property	Description
`java.naming.factory.initial`	Specifies the class name of initial context factory to use with your JNDI implementation.
`java.naming.factory.url.packages`	Specifies a colon-separated list of package prefixes to use when loading in URL context factories. `com.sun.jndi.url` is always added to end of list.
`java.naming.provider.url`	Specifies configuration information for provider to use.

Example

Example 56 shows the initial context properties for using JBoss.

Example 56: *JBoss Initial Context Properties*

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=jnp://localhost:1099
```

Exposing a Stateless EJB

Procedure

To expose a stateless EJB using Artix, complete the following steps:

1. Use the `javatowsdl` tool on your stateless EJB's remote interface to generate a logical WSDL document representing its interface. This process is described in *Designing Artix Solutions*.
2. Add the binding for the desired payload format to the generated WSDL document. For example, if you are exposing the stateless EJB as a Web service, you would add a SOAP binding.
3. Add the port information defining the transport that the stateless EJB can be contacted on. For example, to expose the stateless EJB as a Web service, you would add an HTTP port.
4. Generate Java code for an Artix server from your edited WSDL document using `wsdltojava`. This procedure is described in *Developing Artix Applications with Java*.
5. Edit the generated server's mainline to instantiate an `EJBServant` instead of a `SingleInstanceServant`.
6. Build and run the generated server.

Example

Example 57 shows the `main()` of an Artix Java server modified to expose a stateless EJB. The highlighted code instantiates an `EJBServant` for a stateless EJB with the JNDI name `HelloWorld`.

Example 57: *EJBServant main()*

```
// Java
import com.iona.jbus.*;
import javax.xml.namespace.QName;

public class Server
{
  public static void main(String args[])
  throws Exception
  {
```

Example 57: *EJBServant main()*

```
    // Initialize the Artix bus
    Bus bus = Bus.init(args);

    // Register the implementation object factory
    QName name = new QName("http://xmlbus.com/HelloWorld",
                           "HelloWorldService");
    Servant servant =
                   new EJBServant(“./HelloWorld.wsdl”,
                                  bus, “HelloWorld”);
    bus.registerServant(servant, name, "HelloWorldPort");

    // Start the Bus
    bus.run();
  }
}
```

Glossary

A

Artix Designer

A suite of GUI tools for creating, managing, and deploying Artix integration solutions.

B

Binding

A binding associates a specific transport/protocol and data format with the operations defined in a `<portType>`.

Bus

See Service Bus

Bridge

A usage mode in which Artix is used to integrate applications using different payload formats.

C

Collection

A group of related WSDL contracts that can be deployed as one or more physical entities such as Java, C++, or CORBA based applications. It can also be deployed as a switch process.

Connection

An established communication link between any two Artix endpoints.

Contract

An Artix contract is a WSDL file that defines the interface and all connection-related information for that interface. A contract contains two components: logical and physical. The logical contract defines things that are independent of the underlying transport and wire format, and is specified in the `<portType>`, `<operation>`, `<message>`, `<type>`, and `<schema>` WSDL tags.

The physical contract defines the payload format, middleware transport, and service groupings, and the mappings between these things and portType 'operations.' The physical contract is specified in the `<port>`, `<binding>` and `<service>` WSDL tags.

Contract Editor

A GUI tool used for editing Artix contracts. It provides several wizards for adding services, transports, and bindings to an Artix contract.

D

Deployment Mode

One of two ways in which an Artix application can be deployed: Embedded and Standalone. An embedded-mode Artix application is linked with Artix-generated stubs and skeletons to connect client and server to the service bus. A standalone application runs as a separate process in the form of a daemon.

E

Embedded Mode

Operational mode in which an application creates a Service Access Point, either by invoking Artix APIs directly, or by compiling and linking Artix-generated stubs and skeletons to connect client and server to the service bus.

Endpoint

The runtime deployment of one or more contracts, where one or more transports and its marshalling is defined, and at least one contract results in a generated stub or skeleton (thus an endpoint can be compiled into an application). Contrast with Service.

H

Host

The network node on which a particular service resides.

M

Marshalling Format

A marshalling format controls the layout of a message to be delivered over a transport. A marshalling format is bound to a transport in the WSDL definition of a Port and its binding. A binding can also be specified in a logical contract portType, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

P

Payload Format

The on-the-wire structure of a message over a given transport. A payload format is associated with a port (transport) in the WSDL using the binding definition.

Protocol

A protocol is a transport whose format is defined by an open standard.

R

Routing

The redirection of a message from one WSDL binding to another. Routing rules are specified in a contract and apply to both endpoints and standalone services. Artix supports port-based routing and operation-based routing defined in WSDL contracts. Content-based routing is supported at the application level.

Router

A usage mode in which Artix redirects messages based on rules defined in an Artix contract.

S

Service

An Artix service is an instance of an Artix runtime deployed with one or more contracts, but with no generated language bindings. The service has no compile-time dependencies. A service is dynamically configured by deploying one or more contracts on it.

Service Access Point

The mechanism, and the points at which individual service providers and consumers connect to the service bus.

Service Bus

The set of service providers and consumers that communicate via Artix. Also known as an Enterprise Service Bus.

Standalone Mode

An Artix instance running independently of either of the applications it is integrating. This provides a minimally invasive integration solution and is fully described by an Artix contract.

Switch

A usage mode in which Artix connects applications using two different transport mechanisms.

System

A collection of services and transports.

T

Transport

An on-the-wire format for messages.

Transport Plug-in

A plug-in module that provides wire-level interoperation with a specific type of middleware. When configured with a given transport plug-in, Artix will interoperate with the specified middleware at a remote location or in another process. The transport is specified in the `<port>` element of a contract.

W

Workspace

The Artix Workspace defines the structure of your Artix solution. It is the first thing you need to create when using the Designer, and all of the solution's components are included within it.

A workspace will typically have one or more collections, which in turn contain resources that define your solution's interface. A workspace also contains shared resources that are common across one or more collections.

Index