IONA®

Artix™

Deploying & Managing Artix Solutions

Version 1.3, December 2003

Making Software Work Together™

Updated: 10-Dec-2003

M 3 1 8 1

Contents

List of Tables

List of Figures

Preface

Audience

This guide is intended for Artix system administrators. It assumes that the reader has a working knowledge of the middleware transports that are being used with Artix.

Organization of this guide

This guide is divided as follows:

- Chapter 1 provides an overview of the concepts behind using Artix to solve integration projects and how Artix fits into a software environment.
- Chapter 2 describes how to configure Artix services to provide optimal performance.
- Chapter 3 provides a detailed discussion of using the advanced logging features of Artix.
- Chapter 4 describes how to deploy the Artix standalone service.
- Chapter 5 describes how to use the Artix Locator Service.
- Chapter 6 describes how to use the Artix Session Manager.

Online help

Artix Designer includes comprehensive online help, providing:

- Detailed step-by-step instructions on how to perform important tasks.
- A description of each screen.
- A comprehensive index and glossary.
- A full search feature.
- Context-sensitive help.

The **Help** menu in Artix Designer provides access to this online help.

Related documentation

The library for Artix includes the following:

- *Getting Started with Artix*
- *Artix Tutorial*
- *Deploying and Managing Artix Solutions*
- *Designing Artix Solutions*
- *Developing Artix Appliations in C++*
- *Developing Artix Appliations in Java*
- *Artix Security Guide*
- *Artix Thread Library Reference*

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

Reading path

If you are new to Artix, you should read the documentation in the following order:

1. *Getting Started with Artix*

   The getting started book describes the basic concepts behind Artix. It also provides details on installing the system and a detailed walk through for developing a C++ client for a Web Service.

2. *Artix Tutorial*

   The tutorial guides you through programming Artix applications against all of the supported transports.

3. *Artix Administration Guide*

   The administration guide provides details about the services and capabilities of Artix and how to integrate them into your software environment.

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts, about Artix and other products. You can access the knowledge base at the following location:

The IONA update center contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical conventions

This guide 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 guide may use 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.

CHAPTER 1

Introduction to Artix

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

Overview

Artix provides a middleware connectivity solution that minimizes invasiveness and lets an organization avoid being locked into any one middleware transport. For example, Artix can be used to connect a BEA Tuxedo™-based 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. In fact, with Artix handling the communication, the client could be changed to an IBM WebSphere MQ™ client without modifying the server.

Along with this functionality Artix provides a great deal of configurability by being 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 with which they are communicating, 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 once. Figure 1 shows a high level view of how a message passes through Artix.

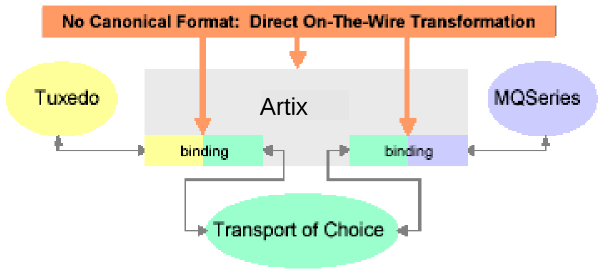

Figure 1: *Artix Message Transporting*

The approach taken by Artix provides a high level of throughput by avoiding the overhead of making two transformations for each message.

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
- CORBA (GIOP) – CORBA format
- FRL – fixed record length
- VRL – variable record length
- SOAP
- TibrvMsg - TIBCO Rendezvous format

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 is made up 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 is made up of the standard WSDL tags `<binding>`, `<port>`, and `<operation>`. It is also the portion of the contract that may contain IONA WSDL extensions.

Deployment models

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

Embedded mode is the most invasive use of Artix and provides the highest performance. 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 runs as a separate process invoked as a service. In standalone 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.

Advanced Features

Artix also supports the following advanced functionality:

- Message routing based on the operation or the port, including routing based on characteristics of the port.
- 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 Application Server Platform 6.0 and Tuxedo 7.1 or higher.
- Session Management
- Location Services
- Load Balancing

CHAPTER 2

Configuration

Artix's runtime configuration provides a great deal of control over how Artix systems perform.

Overview

There are several tasks involved in creating an environment in which Artix applications can run:

- Establishing the host computer environment
- Establishing the common and application-specific Artix runtime environments
- Configuring the plug-ins to provide optimal performance.

In this chapter

This chapter discusses the following topics:

Establishing the Host Computer Environment

Overview

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

Environmental variables

Artix requires that the following environment variables be set on your system:

- JAVA_HOME
- IT_PRODUCT_DIR
- IT_CONFIG_FILE
- IT_IDL_CONFIG_FILE
- IT_CONFIG_DIR
- IT_CONFIG_DOMAINS_DIR
- IT_DOMAIN_NAME
- PATH

JAVA_HOME

The path to your system's JDK is specified with the system environment variable `JAVA_HOME`. This must be set if you wish to use the Artix Designer.

IT_PRODUCT_DIR

`IT_PRODUCT_DIR` points to the top level of your IONA product installation. For example, if you install Artix into the `C:\Program Files\IONA` directory of your Windows system, you would set `IT_PRODUCT_DIR` to point to that directory.

Note: If you have other IONA products installed and you choose not to install them into the same directory tree, you will need to 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_CONFIG_FILE

IT_CONFIG_FILE specifies the location of the configuration file Artix services use by default. You can overide this setting by using the -ORBdomain_name and -ORBconfig_domains_dir command line options.

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. The configuration file for the Artix IDL compiler is set as follows.

UNIX

Defaults to *INSTALL_DIR*/artix/1.2/etc/idl.cfg.

Windows

Defaults to *INSTALL_DIR*\artix\1.2\etc\idl.cfg.

Note: Do not modify the default IDL configuration file.

IT_CONFIG_DIR

IT_CONFIG_DIR specifies the root configuration directory. The default root configuration directory is /etc/opt/iona on UNIX, and *product-dir*\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 *ORBconfig_dir*/domains on UNIX, and *ORBconfig_dir*\domains on Windows. You can override this variable using the -ORBconfig_domains_dir command line parameter.

IT_DOMAIN_NAME

IT_DOMAIN_NAME specifies the name of the configuration domain used by Artix to locate its configuration information. This variable also specifies the name of the file in which the configuration information is stored. For example the configuration information for domain artix would be stored in *ORBconfig_dir*\domains\atrix.cfg on Windows and *ORBconfig_dir*/domains/artix.cfg on Unix. You can override this variable with the -ORBdomain_name command line parameter.

PATH

The Artix bin directories should be placed first on the PATH to ensures 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 the Application Server Platform and/or Tuxedo (both of which 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/1.2/bin`

Windows

`%IT_PRODUCT_DIR%\artix\1.2\bin`

Running the artix_env Script

The installation process creates a script, `artix_env`, that captures the default information for setting the host computer's Artix environment. Running this script will properly configure your system to use Artix. It is located in the Artix bin directory.

```
IT_PRODUCT_DIR\artix\1.2\bin\artix_env
```

Configuring Artix Runtime Behavior

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.

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 `%IT_PRODUCT_DIR%\artix\1.2\etc\domains\artix.cfg` on Windows and `$IT_PRODUCT_DIR/artix/1.2/etc/domains/artix.cfg` on Unix.

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. However, this is only recommended if you are familiar with configuring IONA's ART platform.

Configuration Scopes

An Artix configuration domain is divided into 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, `( nameTag {…}; )`.

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"];
 };
 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 that the `orb_plugins` list is redefined within each configuration scope.

Mapping to a configuration scope

To make an Artix process run under a configuration scope, you name that scope using the `-ORBname` parameter. Configuration scope names are specified using the for *`scope`*`.`*`subscope`*. For example, the scope for the telco server demo shown in Example 1 on page 10 would be 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 methods for 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` configuration 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 configuration scope is not located, the process starts under the highest level configuration scope that matches the specified cope name. If there are no configuration scopes that correspond to the `ORBname` parameter, the Artix process runs under the default global scope. For example, if the nested configuration scope `tibrv` does not exist, the Artix process uses the configuration specified in the `demo` configuration scope; if the scope `demo` does not exist, the process runs under the default global scope.

Namespaces

Most configuration variables are organized within namespaces, which serve to 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 you would set the following variable:

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

To set the location of the routing plug-in's contract you would set the following variable:

```
plugins:routing:wsdl_url
```

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.

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"];
```

Runtime Configuration Variables

In this section

The following topics are discussed in this section:

ORB Plug-ins List

Overview

The orb_plugins variable specifies the plug-ins that Artix processes load during initialization. A plug-in is a class or code library that can be loaded into an Artix application at runtime. These plug-ins provide the user the ability to load network transports, payload format mappers, error logging streams, and other features “on the fly.”

The default entry for the orb_plugins variable includes all of the logging and transport plug-ins:

```
orb_plugins = ["xmlfile_log_stream",
               "iiop_profile",
               "giop",
               "iiop",
               "soap",
               "http",
               "tunnel",
               "mq",
               "ws_orb"];
```

Artix plug-ins

Each network transport and payload format that Artix is capable of interoperating with uses its own plug-in. Many of the Artix features also use plug-ins. The Artix transport plug-ins are listed in Table 1.

Table 1: *Artix Transport Plug-ins*

Plug-in	Transport
`http`	Provides support for using HTTP and HTTPS.
`ws_orb`	Provides support for CORBA interoperability.
`tunnel`	Provides support for the IIOP transport using non-CORBA payloads.
`tuxedo`	Provides support for Tuxedo interoperability.
`mq`	Provides support for WebSphere MQ interoperability.
`tibrv`	Provides support for TIBCO Rendezvous interoperability.

The Artix payload format plug-ins are listed in Table 2.

Table 2: *Artix Payload Format Plug-ins*

Plug-in	Payload Format
`soap`	Decodes and encodes messages using the SOAP format.
`G2`	Decodes and encodes messages packaged using the G2++ format.
`fml`	Decodes and encodes messages packaged in FML format.
`tagged`	Decodes and encodes messages packed in variable record length messages or another self-describing message format.
`fixed`	Decode and encodes fixed record length messages.

The Artix feature plug-ins are listed in Table 3.

Table 3: *Artix Service Plug-ins*

Plug-in	Artix Feature
`routing`	Enables Artix routing.
`locator_endpoint`	Enables endpoints to use the Artix locator service.
`service_locator`	Enables the Artix locator. An Artix server acting as the locator service must load this plug-in.
`artix_wsdl_publish`	Enables Artix endpoints to publish and use Artix object references.
`session_manager_service`	Enables the Artix Session Manager. An Artix server acting as the session manager must load this plug-in.
`session_endpoint_manager`	Enables the Artix Session Manager. Endpoints wishing to be managed by the session manager must load this plug-in.

Table 3: *Artix Service Plug-ins*

Plug-in	Artix Feature
`sm_simple_policy`	Enables the policy mechanism for the Artix Session Manager. Endpoints wishing to be managed by the session manager must load this plug-in.

Policies

Overview

The `policies` namespace contains the following two variable for controlling the publishing of server hostnames:

- http:server_address_mode_policy:publish_hostname
- soap:server_address_mode_policy:publish_hostname

If the policy corresponding to the transport used by the server, the dynamically generated contract will be published with the original contents of the address element.

http:server_address_mode_policy:publish_hostname

`http:server_address_mode_policy:publish_hostname` specifies how the server's address is published in dynamically generated Artix contracts. When set this policy is set to `false`, the dynamically generated contract will publish the IP address of the running server in the `<http:address>` element describing the server's location. When this policy is set to true, the hostname of the machine hosting the running server is published in the `<http:address>` element describing the server's location.

soap:server_address_mode_policy:publish_hostname

`soap:server_address_mode_policy:publish_hostname` specifies how the server's address is published in dynamically generated Artix contracts. When set this policy is set to `false`, the dynamically generated contract will publish the IP address of the running server in the `<soap:address>` element describing the server's location. When this policy is set to true, the hostname of the machine hosting the running server is published in the `<soap:address>` element describing the server's location.

Binding Lists

Overview

When using Artix's CORBA functionality you need to configure how Artix binds itself to message interceptors. The Artix `binding` namespace contains variables that specify interceptor settings. An interceptor acts on a message as it flows from sender to receiver. Computing concepts that fit the interceptor abstraction include transports, marshaling streams, transaction identifiers, encryption, session managers, message loggers, containers, and data transformers. Interceptors are a form of the "Chain of Responsibility" design pattern. Artix creates and manages chains of interceptors between senders and receivers, and the interceptor metaphor is a means of creating a "virtual connection" between a sender and a receiver.

The Artix `binding` namespace includes the following variables:

- client_binding_list
- server_binding_list

client_binding_list

Artix provides client request-level interceptors for OTS, GIOP, and POA collocation (where server and client are collocated in the same process), and message-level interceptors used in client-side bindings for IIOP, SHMIOP and GIOP.

The `client_binding_list` specifies a list of potential client-side bindings. Each item is a string that describes one potential interceptor binding. For example:

```
binding:client_binding_list = ["OTS+POA_Coloc","POA_Coloc","OTS+GIOP+IIOP","GIOP+IIOP"];
```

Interceptor names are separated by a plus (+) character. Interceptors to the right are "closer to the wire" than those on the left. The syntax is as follows:

- Request-level interceptors, such as `GIOP`, must precede message-level interceptors, such as `IIOP`.
- `GIOP` or `POA_coloc` must be included as the last request-level interceptor.

- Message-level interceptors must follow the GIOP interceptor, which requires at least one message-level interceptor.
- The last message-level interceptor must be a message-level transport interceptor, such as IIOP or SHMIOP.

When a client-side binding is needed, the potential binding strings in the list are tried in order, until one successfully establishes a binding. Any binding string specifying an interceptor that is not loaded, or not initialized through the orb_plugins variable, is rejected.

For example, if the ots plug-in is not configured, bindings that contain the OTS request-level interceptor are rejected, leaving ["POA_Coloc", "GIOP+IIOP", "GIOP+SHMIOP"]. This specifies that POA collocations should be tried first; if that fails, (the server and client are not collocated), the GIOP request-level interceptor and the IIOP message-level interceptor should be used. If the ots plug-in is configured, bindings that contain the OTS request interceptor are preferred to those without it.

server_binding_list

server_binding_list specifies interceptors included in request-level binding on the server side. The POA request-level interceptor is implicitly included in the binding.

The syntax is similar to client_binding_list. However, in contrast to the client_binding_list, the left-most interceptors in the server_binding_list are “closer to the wire”, and no message-level interceptors can be included (for example, IIOP). For example:

```
binding:server_binding_list = ["OTS",""];
```

An empty string ("") is a valid server-side binding string; this specifies that no request-level interceptors are needed. A binding string is rejected if any named interceptor is not loaded and initialized.

The default server_binding_list is ["OTS", ""]. If the ots plug-in is not configured, the first potential binding is rejected, and the second potential binding ("") is used, with no explicit interceptors added.

Thread Pool Control

Overview

Variables in the `thread_pool` namespace set policies related to thread control. They can be set globally for Artix instances in a configuration scope, or they can be set on a per-service basis. The settings set on a per-service basis override the global settings for the configuration scope.

To set the values globally, use the following syntax:

```
thread_pool:variable_name
```

To set the values on a per-service basis you can specify either the service's name or the service's fully qualified QName. The syntax is as follows:

```
thread_pool:service_name:variable_name
thread_pool:service_qname:variable_name
```

For example, if an Artix instance's contract has a service named `personalInfoService`, you would specify its thread control settings as follows:

```
thread_pool:personalInfoService:variable_name
```

The thread control settings specify the values for the thread pool on a per-port basis. For instance, if `personalInfoService` describes three ports, each port will have its own thread pool with values as specified by the settings in the `thread_pool:personalInfoService` namespace.

The following variables are in this namespace:

- `high_water_mark`
- `initial_threads`
- `low_water_mark`

high_water_mark

`high_water_mark` sets the maximum number of threads allowed in each port's thread pool. Defaults to `25`.

initial_threads

`initial_threads` sets the number of initial threads in each port's thread pool. Defaults to `2`.

low_water_mark

`low_water_mark` sets the minimum number of threads in each port's thread pool. Artix will terminate unused threads until only this number exists. Defaults to `5`.

Artix Plug-in Configuration

Overview

Each Artix transport, payload format, and service have properties which are configurable. The variables used to configure plug-in behavior are specified in the configuration scopes of each Artix runtime instance and follow the same order of precedence. A plug-in setting specified in the global configuration scope will be overridden in favor of a value set in a narrower scope.

For example, if you set `plugins:routing:use_type_factory` to `true` in the global configuration scope and set it to `false` in the scope `widget_form`, all Artix runtimes, except for those running under the scope `widget_form`, would use `true` for the value of `use_type_factory`. Any Artix instance using the scope `widget_form` would use `false` for the value of `use_type_factory`.

In this section

This section discusses the following topics:

Routing Plug-in

Overview

The routing plug-in uses the following variables:

- plugins:routing:routing_wsdl
- plugins:routing:use_type_factory
- plugins:routing:use_pass_through

plugins:routing:routing_wsdl

`plugins:routing:routing_wsdl` specifies the URL to search for Artix contracts containing the routing rules for your application. This value can be either a single URL or a list of URLs. If your application is using the routing plug-in you must specify a value for this variable.

plugins:routing:use_type_factory

`plugins:routing:use_type_factory` specifies if the routing plug-in loads user compiled type factories. The default setting is `false`.

Note: The use of type factories in routing is deprecated.

plugins:routing:use_pass_through

`plugins:routing:use_pass_through` specifies if the routing plug-in uses the pass-through routing optimization. This optimization allows the router to copy the message buffer directly from the source endpoint to the destination endpoint if both use the same binding. The default value is `true`.

Note: A few attributes are carried in the message body, as opposed to by the transport. Such attributes are always propagated when the pass-through optimization is in effect, regardless of attribute propagation rules.

CORBA Plug-in

Overview

In general, the Artix CORBA plug-in does not have any configuration variables directly associated with it. However, the CORBA plug-in is implemented using the same framework as IONA's Application Server Platform and it is affected by the same configuration settings as IONA's Application Server Platform.

For example, if you set the configuration variable:

```
policies:giop:interop_policy:send_principal = "true";
```

This will impact the CORBA messages that Artix sends.

Or, if you remove the plug-in `POA_Coloc` from the client binding list, then collocation will not work.

Tuxedo Plug-in

Overview

The Tuxedo plug-in has only one configuration variable:

- plugins:tuxedo:server

plugins:tuxedo:server

`plugins:tuxedo:server` is a boolean that specifies if the Artix process is a Tuxedo server and must be started using `tmboot`. The default is `false`.

Locator Service Plug-in

Overview

The locator service plug-in, `service_locator`, has the following configuration variables:

- plugins:locator:service_url
- plugins:locator:peer_timeout

plugins:locator:service_url

`plugins:locator:service_url` specifies the location of the Artix contract defining the location service and configuring its address. A copy of this contract, `locator.wsdl`, is located in the wsdl folder of your Artix installation.

plugins:locator:peer_timeout

`plugins:locator:peer_timeout` specifies the amount of time, in milliseconds, the locator plug-in waits between keep-alive pings of the services registered with it. The default is 4000000 (4 sec.).

Locator Service Endpoint Plug-in

Overview

The locator service endpoint plug-in, `locator_endpoint`, has the following configuration variables:

- plugins:locator:wsdl_url
- plugins:session_endpoint_manager:peer_timout

plugins:locator:wsdl_url

`plugins:locator:wsdl_url` specifies the location of the Artix contract defining the location service and specifying the address locator endpoints use to communicate with the locator service. A copy of this contract, `locator.wsdl`, is located in the wsdl folder of your Artix installation.

plugins:session_endpoint_manager:peer_timout

`plugins:session_endpoint_manager:peer_timout` specifies the amount of time, in milliseconds, the server waits between keep-alive pings of the locator service. The default is 4000000 (4 sec.).

Session Manager Plug-in

Overview

The session manager plug-in, `session_manager_service`, has the following configuration variables:

- plugins:session_manager_service:service_url
- plugins:session_manager_service:peer_timeout

plugins:session_manager_service:service_url

`plugins:session_manager_service:service_url` specifies the location of the Artix contract defining the session manager. A copy of this contract, `session-manager.wsdl`, is located in the wsdl folder of your Artix installation.

plugins:session_manager_service:peer_timeout

`plugins:session_manager_service:peer_timeout` specifies the amount of time, in milliseconds, the session manager plug-in waits between keep-alive pings of the services registered with it. The default is 4000000 (4 sec.).

Session Manager Simple Policy Plug-in

Overview

The session manager's simple policy plug-in, `sm_simple_policy`, has the following configuration variables:

- plugins:sm_simple_policy:max_concurrent_sessions
- plugins:sm_simple_policy:min_session_timeout
- plugins:sm_simple_policy:max_session_timeout

plugins:sm_simple_policy:max_concurrent_sessions

`plugins:sm_simple_policy:max_concurrent_sessions` specifies the maximum number of concurrent sessions the session manager will allocate. Default value is `1`.

plugins:sm_simple_policy:min_session_timeout

`plugins:sm_simple_policy:min_session_timeout` specifies the minimum amount of time, in seconds, allowed for a session's timeout setting. Zero means the unlimited. Default is 5.

plugins:sm_simple_policy:max_session_timeout

`plugins:sm_simple_policy:max_session_timeout` specifies the maximum amount of time, in seconds, allowed for a session's timesout setting. Zero means the unlimited. Default is 600.

Session Manager Endpoint Plug-in

Overview

The session manager endpoint plug-in, `session_endpoint_manager`, has the following configuration variables:

- plugins:session_endpoint_manager:wsdl_url
- plugins:session_endpoint_manager:endpoint_manager_url
- plugins:session_endpoint_manager:default_group
- plugins:session_endpoint_manager:header_validation

plugins:session_endpoint_manager:wsdl_url

`plugins:session_endpoint_manager:wsdl_url` specifies the location of the contract defining the session management service the endpoint manager is to contact.

plugins:session_endpoint_manager:endpoint_manager_url

`plugins:session_endpoint_manager:endpoint_manager_url` specifies the location of the contract defining the endpoint manager. The contract contains the contact information for the endpoint manager.

plugins:session_endpoint_manager:default_group

`plugins:session_endpoint_manager:default_group` specifies the default group name for all endpoints that are instantiated using the configuration scope.

plugins:session_endpoint_manager:header_validation

`plugins:session_endpoint_manager:header_validation` specifies whether or not a server validates the session headers passed to it by clients. Default value is `true`.

WSDL Publishing Plug-in

Overview

The WSDL publishing plug-in, `artix_wsdl_publishing`, has the following configuration variables:

- plugins:wsdl_publish:publish_port

plugins:wsdl_publish:publish_port

`plugins:wsdl_publish:publish_port` specifies the port on which the WSDL publishing port can be contacted.

CHAPTER 3

Artix Logging and SNMP Support

This chapter describes various Artix logging approaches, including Artix support for SNMP (Simple Network Management Protocol) and integration with third-party SNMP management tools.

In this chapter

This chapter includes the following sections:

Artix Logging

Overview

Artix provides the following `IT_Logging::logstream` plug-ins: the `xmlfile_logstream` and `snmp_logstream`. In addition, IONA Application Server Platform logging features such as `local_logstream`. are provided.

For information on configuring these plug-ins see "Configuration" on page 5.

Using Trace Macros

Artix Trace Macros

In using Trace macros, the most important concept is the trace level. Trace level is an enum, defined in `it_bus/logging_support`, that lets you filter events:

```
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");
```

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 set 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);
```

If you must create your own output using `iostreams` or another expensive process that isn't supported by the macro, you use the trace guard block, so that the trace level test will prevent your trace creation code from running when it will 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, then the entry is written to disk. The default log file name is `it_bus.log`.

Application Server Platform Trace Macros

Overview

<orbix\logging_support.h> defines ASP-style logging macros.

IT_LOG_MESSAGE Macros

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

Here 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.

Using the SNMP Logging Plug-in

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 SNMPv1/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.

the 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 2: *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 2: *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 2: *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 2: *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 2: *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 logstream plug-in called `snmp_log_stream`. The shlib name of the SNMP plug-in found in the `artix.cfg` file is:

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

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>"
```

The last plugin described, `oid`, is the Enterprise Object Identifier. This identifier 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. Thus 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 hence, acts like the `local_log_stream` plug-in.

Using the XML Logging Plug-in

Using the XML Logging Plug-in

You can modify your event log filters to enable or disable Artix tracing.

The out-of-the-box setting for `event_log:filters` is `["*=FATAL+ERROR"]`.

So, for example, to cause transport buffer events to be shown, update the event_log:filters to include `INFO_MED`:

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

The following causes typical trace statement output:

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

In addition, you can:

- add `xmlfile_log_stream` to the `orb_plugins` list
- update the filename variable (default is it_bus.log):

  ```
  plugins:xmlfile_log_stream:filename = "artix_logfile.xml";
  ```

- modify the size element (default is 2MB):

  ```
  plugins:xmlfile_log_stream:max_file_size = "100000";
  ```

- add optional element (default is false):

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

The Artix logging output from the TRACE macros now goes to the event log, so `local_log_stream`, `xmlfil_log_stream` or SNMP_log_stream can be used.

logging_support.h

The following example shows the contents of logging_support.h:

Example 3: *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, these are just 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 3: *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 3: *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 3: *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 wouldn't 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 3: *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 3: *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, if you want 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 Overview

The `IT_Logging` module is the centralized point for controlling all logging methods. 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 4: *IT_Logging Common Data Types, Methods, and Macros*

Common Data Types	Methods and Macros
`ApplicationId`	`format_message()`
`EventId`	
`EventParameters`	`IT_LOG_MESSAGE()`
`EventPriority`	`IT_LOG_MESSAGE_1()`
`SubsystemId`	`IT_LOG_MESSAGE_2()`
`Timestamp`	`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

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()
```

Example

Controlling Application Logging

This example shows application logging enable by including the `xmlfile_log_stream` plugin in the `orb_plugins` list (this plugin is included in the default `orb_plugins` list, though it is not included in the `orb_plugins` lists within many of the demo program configuration scopes). If you want to enable logging to an XML file for the applications you develop, include this plugin in your orb_plugins list.

To enable usage of the xmlfile_log_stream plugin, several other configuration variables must also be set. These variable are all set within the default/global scope in the `artix.cfg` file:

```
plugins:xmlfile_log_stream:shlib_name =
  “it_xmlfile”;

plugins:xmlfile_log_stream:filename =
  “artix_logfile.xml”;
# default: it_bus.log

plugins:xmlfile_log_stream:max_file_size =
  “2000000”;
# default: 2 mb

plugins:xmlfile_log_stream:use_pid =
  “false”;
# default: false

# standard logging setting; logs errors and warnings
event_log:filters =
    [“*=FATAL+ERROR+WARNING”];

# very detailed logging
#event_log:filters = [“*=*”];

# transport buffer logging
#event_log:filters =
    [“*=FATAL+ERROR+WARNING+INFO_MED”];

# high level informational logging
#event_log:filters =
    [“*=FATAL+ERROR+WARNING+INFO_HI”];
```

Using the Logging Functionality

The default configuration settings enable logging of only serious errors and warnings. If you want more exhaustive information, you should either select a different filter list at the default scope, or include a more expansive `event_log:filters` configuration variable within your configuration scope.

If you have trouble running any of the demos, you should enable a high level of logging, which requires adding the xmlfile_log_stream plugin to the orb_plugins list and selecting the desired reporting level.

Performance Logging

Overview

The performance logging plug-ins allow applications based on IONA products to integrate effectively with Enterprise Management Systems (EMSs). Currently artix support integration with IBM Tivoli™.

This section covers general Artix details. For information on integration with a Tivoli installation, refer to the *Tivoli Integration Guide*.

Performance logging

Performance logging lets you see how each server is responding to load. These plug-ins log this data to file or syslog. Your EMS can read the performance data from the logs and initiate appropriate actions. For example, issuing a restart to a server that has become unresponsive, or starting a new replica for an overloaded cluster.

Configuration

The performance logging component consist of three plug-ins:

- The response time logger plugin
- The request counter plugin
- The collector plugin

The response time logger plugin monitors response times of requests as they pass through ART binding chains. It can be used to collect response times for CORBA, RMI-IIOP or HTTP calls in IONA's CORBA and J2EE products. The request counter plugin performs the same function for Artix.

The collector plugin periodically harvests data from the response time logger and request counter plug-ins and logs the results. To monitor the performance of CORBA or J2EE requests (made in the context of IONA's Application Server Platform), you must perform the following steps to reconfigure the Application Server Platform:

Add `it_response_time_logger` to the orb_plugins list for the server you wish to instrument. Add `it_reponse_time_logger` to the server and servlet binding lists for that server. For example:

```
binding:servlet_binding_list= [
"it_response_time_logger + it_servlet_context + it_character_encoding
+ it_locale + it_naming_context + it_exception_mapping + it_http_sessions
+ it_web_security + it_servlet_filters + it_web_redirector + it_web_app_activator "
];
binding:server_binding_list=[
"it_response_time_logger+it_naming_context+CSI+j2eecsi+OTS+it_security_role_mapping",
"it_response_time_logger+it_naming_context+OTS+it_security_role_mapping",
"it_response_time_logger+it_naming_context + CSI+j2eecsi+it_security_role_mapping",
"it_response_time_logger+it_naming_context+it_security_role_mapping",
"it_response_time_logger+it_naming_context",
"it_response_time_logger"
];

orb_plugins=[
"it_servlet_binding_manager", "it_servlet_context",
"it_http_sessions", "it_servlet_filters", "http",
"it_servlet_dispatch", "it_exception_mapping", "it_naming_context",
"it_web_security", "it_web_app_activator",
"it_default_servlet_binding", "it_security_service", "it_character_encoding",
"it_locale", "it_classloader_servlet","it_classloader_mapping",
"it_web_redirector", "it_deployer",
"it_response_time_logger"
];
```

Configuring the collector plugin

You can configure the collector plugin to log data either to a file or to syslog. The following example results in performance data being logged to `/var/log/my_app/perf_logs/treasury_app.log` every 90 seconds (if you do not specify the period, it defaults to 60 seconds):

```
plugins:it_response_time_collector:period = "90";

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

You can also configure the collector to log to a syslog daemon or Windows Event Log:

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

syslog_appid lets you specify the application name, which is prepended to all syslog messages. If you do not specify a syslog_appid, it defaults to "iona".

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

Alternatively, you can use syslogd as a mechanism for monitoring a cluster, by choosing one syslogd to act as the central logging server for the cluster. For example, to use the host teddy as the central log server, edit the /etc/syslog.conf file for each host that runs a server replica, and add:

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

Some syslog daemons do 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. Consult the man pages on your system to determine whether this is necessary and what flags to use.

Logging Formats

Performance data is logged in a well-defined format. For CORBA and J2EE applications based on IONA's Application Server Platform, this format is:

```
YYYY-MM-DDTHH:MM:SS [operation=name] count=n avg=n max=n min=n
```

- operation is the name of the operation for CORBA invocations or the URI for requests on servlets.
- count is the number of times this operation or URI was logged during the last interval.
- avg is average response time (in milliseconds) for this operation or URI during the last interval.
- max is the longest response time (in milliseconds) for this operation or URI during the last interval.
- min is the shortest response time (in milliseconds) for this operation or URI during the last interval.

The format for Artix log messages is:

```
YYYY-MM-DDTHH:MM:SS [namespace=nnn service=sss port=ppp operation=name] count=n avg=n max=n min=n
```

- `namespace` is an Artix namespace.
- `service` is an Artix service.
- `port` is an Artix port.
- `operation` is the name of the operation for CORBA invocations or the URI for requests on servlets.
- `count` is the number of times this operation or URI was logged during the last interval.
- `avg` is average response time (in milliseconds) for this operation or URI during the last interval.
- `max` is the longest response time (in milliseconds) for this operation or URI during the last interval.
- `min` is the shortest response time (in milliseconds) for this operation or URI during the last interval.

The combination of namespace, service and port denote a unique Artix Service Access Point.

CHAPTER 4

Artix Standalone Service

Artix lets you deploy middleware translation functions as a standalone service external to both client and server applications. The Artix standalone service can perform transport switching, message routing, and middleware bridging between non-Artix enabled applications.

In this chapter

This chapter discusses the following topics:

The Artix Standalone Service

Overview

The Artix standalone service is 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.

How it works

The Artix standalone service is a daemon that listens for traffic on access points specified in the Artix contract. It re-directs messages based on the routing rules 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 standalone service's behavior is controlled by a combination of an Artix contract and the Artix configuration file. For more information on Artix contracts see the *Artix Developer's Guide*. For more information on configuring the Artix runtime see "Configuration" on page 5.

Deployment patterns

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

Deploying several daemons, each of which bridges between two distinct applications.

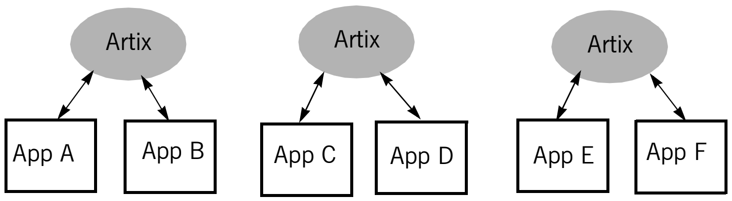

Figure 2: *Using Multiple Artix Daemons*

This approach simplifies designing integration solutions and provides faster processing of each message. 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.

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

Deploying one daemon to bridge between all of the applications in a particular domain.

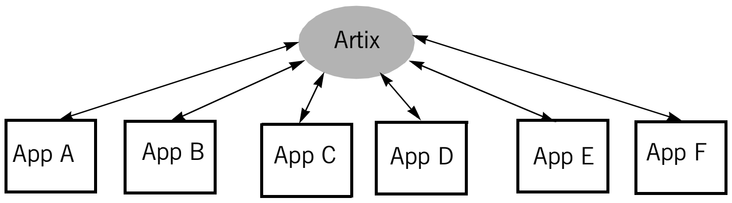

Figure 3: *Using a Single Artix Daemon*

This approach limits the number of external services required in your deployment environment. This can simplify monitoring and installation of deployments. It also reduces the number of “moving parts” in an integration solution.

Configuring the Service

Overview

Each instance of the Artix standalone service running on a host machine needs its own configuration scope to specify the unique port on which its administrative interface listens. Each instance also needs a corresponding administrative interface configuration scope.

Having separate configuration scopes for each instance of the service also allows greater control over the resources the service uses. You can specify that it only load the transport and payload format plug-ins it requires. You can also control the services threading and time-out behaviors.

For more information on configuring Artix, see "Configuration" on page 5.

Orb plugins list

In addition to the Artix plugins that provide support for the transports and payload formats it will be working with, the Artix standalone service needs to load the following plugins:

- iiop_profile
- iiop
- giop

These need to be entered in its `orb_plugins` list.

Service plug-in settings

The configuration variable that controls the behavior of the Artix standalone service are in the `plugins:artix_service` namespace. Table 5 lists the variables and their settings.

Table 5: *Artix Standalone 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 5: *Artix Standalone Service Configuration Variables*

Variable	Effect
`iiop:port`	Specifies the port number on which the service listens for calls from its administrative interface. See "Service admin interface".
`iiop:host`	Specifies the name of the host computer on which the service is running. 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 standalone service must have a corresponding administrative interface configuration scope. This scope must contain an entry for `initial_references:IT_ArtixServiceAdmin:reference`. `initial_references:IT_ArtixServiceAdmin:reference` specifies the port number of this admin 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 hostname of the computer on which the corresponding Artix service is running. *`port`* is the port number on which the corresponding Artix service is listening.

Starting and Stopping the Service

Starting the service

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

```
start_artix_service
```

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

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 6 describes the parameters taken by `itartix_service`.

Table 6: *itartix_service 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 has the name *`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.
`run`	Specifies that the service is to begin monitoring.
`-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.

For more information about configuring Artix see "Configuration" on page 5.

Stopping the service

To stop the Artix standalone service use the following script:

```
stop_artix_service
```

This script will stop an instance of the Artix standalone service started using the start script, `start_artix_service`.

Alternatively, you can manually call the service's administrative interface to stop the service. To do so 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 you wish to shutdown.

Installing the Service as a Windows Service

Overview

On Windows systems, you can install instances of the Artix standalone 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 standalone service as a Windows service, use the following script:

```
install_artix_service
```

This script installs the Artix standalone 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 7 describes the parameters taken by `itartix_service`.

Table 7: *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 has the name *`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.
`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:

```
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 7 describes the parameters taken by `itartix_service`.

Table 8: *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 has the name *`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.

Contracts for the Standalone Service

Routing

Contracts for instances of the Artix standalone service must have routing rules to direct the flow of messages between the services defined within the contract.

You must also ensure that the routing plug-in is loaded by the Artix standalone service by placing the following entry in the `orb_plugins` list of the instance's configuration scope:

```
orb_plugins = [... "routing"];
```

Locating the contracts

The Artix standalone service loads the contract specified by the `plugins:routing:wsdl_url` configuration variable. For example if an instance of the Artix standalone service was designed to use a contract called `personalInfo.wsdl` and the contract was located in `/etc/contracts`, you would place the following in the instance's configuration scope:

```
plugins:routing:wsdl_url="/etc/contracts/personalInfo.wsdl";
```

For more information

For more information on Artix runtime configuration, see "Configuring Artix Runtime Behavior" on page 9.

CHAPTER 5

Using the Artix Locator Service

The Artix Locator allows 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 Service

Overview

A system with many servers cannot afford the overhead of manually propagating each servers contact information to all off 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 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.

Service components

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

Locator Service Plug-in (`service_locator`) 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`) is the portion of the session manager 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 4 shows a diagram of how the locator plug-ins are deployed in an Artix System. While in this example, the locator service plug-in is deployed into a standalone service, it can be deployed in any Artix process.

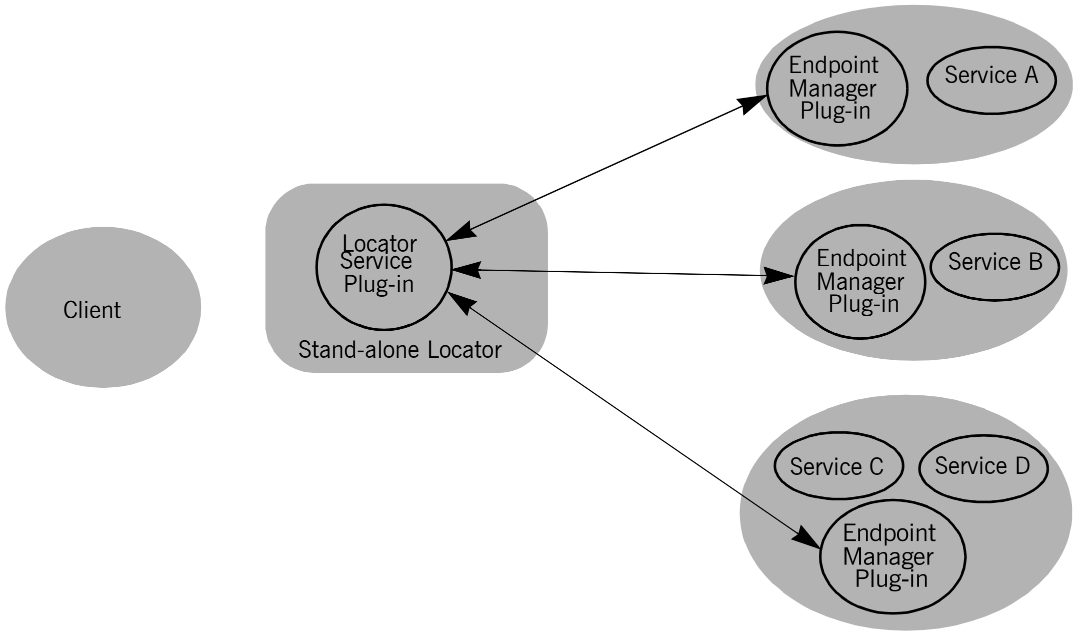

Figure 4: *The Locator Plug-ins*

The endpoint manager plug-ins are deployed into the server processes which contain services that are registered with the locator. A process can host two services, like *Service C* and *Service D* in Figure 4, but the process will 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 will consider the services as a single service and use a round-robin load balance algorithm to hand out references to the separate instances. As shown in Figure 5, as each client makes a request for `widget_service`, the locator cycles through the pool of registered `widget_service` instances. When the fourth client makes a request, the locator will start handing out references from and the top of the pool, `widget_service`$_a$.

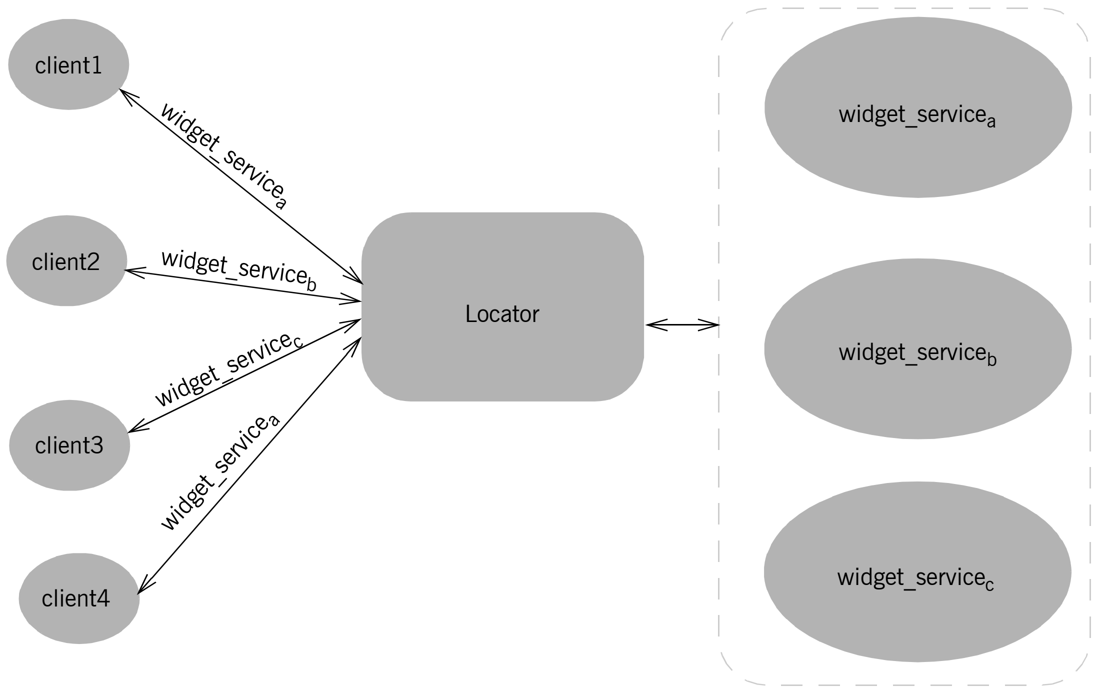

Figure 5: *Locator Load balancing*

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

Deploying the Locator

Overview

The Artix locator is implemented as a group of ART 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 users generate an Artix server that only hosts the locator service and deploy that service into their Artix environment.

In either case, the locator service requires modifications to the Artix configuration domain in which the locator is run. You also need to generate a copy of `locator.wsdl`, the contract that describes the locator service, containing the locator service's contact information.

Building a standalone locator service

To generate a standalone locator service you write a simple Artix server mainline and link it with the Artix libraries. Example 4 shows an example of the locator's mainline.

Example 4: *Artix Locator Mainline*

```
include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_bus/fault_exception.h>

using namespace IT_Bus;

int main(int argc, char* argv[])
{
  try
    {
      IT_Bus::init(argc, argv, "locator_service");
      IT_Bus::run();
      IT_Bus::shutdown();
    }
  catch (IT_Bus::Exception& e)
    {
      printf("Exception occurred: %s", e.Message());
      return 1;
    }

  return 0;
}
```

The locator's `main()` only needs to initialize the Artix bus with the name of the locator's configuration scope and call `IT_Bus::run()`. The configuration scope's name is the third parameter to `IT_Bus::init()`, `locator.service`. The Artix bus will load the plug-ins for the locator service.

Example 5 shows a sample makefile for building the locator service.

Example 5: *Locator Makefile*

```
IT_PRODUCT_VER = 1.2

ART_BIN_DIR=$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\bin
ART_CXX_INCLUDE_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\i
   nclude"
ART_LIB_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\lib"

CXX=cl
CXXFLAGS=-I$(ART_CXX_INCLUDE_DIR) -Zi -nologo -GR -GX -W3 -Zm250
   -MD $(EXTRA_CXXFLAGS) $(CXXLOCAL_DEFINES)

LINK=link
LDFLAGS=/DEBUG /NOLOGO
LDLIBS=/LIBPATH:$(ART_LIB_DIR) $(EXTRA_LIB_PATH) $(LINK_WITH)
   kernel32.lib ws2_32.lib advapi32.lib user32.lib

SHLIB_CXX_COMPILER_ID=          vc60
SHLIBLDFLAGS=-dll -debug -incremental:no

OBJS=$(SOURCES:.cxx=.obj)

LINK_WITH=it_bus.lib it_afc.lib it_art.lib  it_ifc.lib

SOURCES = locator.cxx
all: locator.exe

locator.exe:$(SOURCES) $(OBJS)
  if exist $@ del $@
    $(LINK) /out:$@ $(LDFLAGS) $(OBJS) $(LDLIBS)
```

The locator must be linked with the following Artix libraries:

- `it_bus.lib`
- `it_afc.lib`
- `it_art.lib`
- `it_ifc.lib`

Configuring the locator

To run the locator you need to 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.

In the locator's configuration scope specify that the service plug-in will read the correct Artix contract for the locator by setting `plugins:locator:service_url` to point to the copy of `locator.wsdl` containing the address for this instance of the locator.

Example 6 shows the configuration scope used to start the locator.

Example 6: *Locator configuration scope*

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

For more information on Artix configuration see "Configuration" on page 5.

Generating the locator's contact information

You also need to configure the port on which the locator will run. To do this you modify `locator.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the locator service will listen. This can be either done manually for deploying the locator on a well-known fixed port, or automatically for deploying the locator on a dynamically allocated port.

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 proper address. Example 7 shows a modified locator service contract entry. The highlighted part has been modified to point to the desired address.

Example 7: *Locator Service Address*

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

Dynamic Port

To deploy the locator on a dynamically allocated port, configure the locator to use the copy of `locator.wsdl` shipped with Artix. Once the locator initializes the Artix bus, it will need to publish a new copy of its contract with the actual contact information. Example 8 shows how to publish the locator's contract.

Example 8: *Dynamically Located 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();
```

Starting the locator

Once 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 proper 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 it requires:

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

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

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

Example 9 shows the configuration scope of a server that registers with the locator service.

Example 9: *Server Configuration Scope*

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

`rune_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 "Configuration" on page 5.

Registration

Once 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 will generate a pool of references for the server type. When clients make a request for a server, the locator will supply references from this pool using a round-robin algorithm. For more information on load balancing see "Load Balancing" on page 93.

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. They are:

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 locator service's contract name, `locator.wsdl`, the locator service's QName, and the port name used in the locator service's contract, `LocatorServicePort`.

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

Example 10 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 10: *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. `lookup_endpoint()` 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 of the service is comprised of 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 11 shows the client code to look up an instance of the widget ordering service, `orderWidgetService`.

Example 11: *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 12 shows the client code for creating a proxy widget server from the results of the look up performed in Example 11 on page 91.

Example 12: *Instantiate a Proxy Server*

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

For more information on writing Artix client code read 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

Once 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. It is recommended that 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 will automatically register with the locator. The locator will recognize that the servers all have the same service name specified in their Artix contracts and will create a list of references for these server instances.

As clients make requests for the service, the locator will cycle 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 in effect 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 you need to do the following three things:

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 you need to use `IT_Bus::get_service()` on a bus instance. `get_service()` 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 13 shows how to obtain a handle for a service from the active bus.

Example 13: *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, you 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 will then contact the locator and ask 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 once it has been deregistered.

Example 14 shows how to call `reached_capacity()`.

Example 14: *Calling reached_capacity()*

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

Reregistering a service

When the service is ready to be reregistered, you 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 15 shows how to call `reached_capacity()`.

Example 15: *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 itself.

Endpoint failure

When an endpoint gracefully shuts down, it notifies the locator that it will no longer be available and 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 configuration variable `plugins:locator:peer_timeout`. The default setting is 4 seconds. For more information see "Configuration" on page 5.

Service failure

When the locator service fails all of 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, will periodically attempt to reregister with the locator until they are successful.

You can adjust the interval at which the endpoint pings the locator by setting the configuration variable `plugins:session_endpoint_manager:peer_timout`. The default setting is 4 seconds. For more information see "Configuration" on page 5.

CHAPTER 6

Using the Artix Session Manager

The Artix Session Manager helps you 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 ART plug-ins that work together to provide you control over the number of concurrent clients accessing a group of services and how long each client can use the services in the group before having to check back with the session manager. The two main session manager plug-ins are:

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

Session Manager Endpoint Plug-in (`session_endpoint_manager`) 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 session manager also has a pluggable policy callback mechanism that allows you to implement your own session management policies. Artix session manager includes a simple policy callback plug-in, `sm_simple_policy`, that 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 6 shows a diagram of how the session manager plug-ins are deployed in an Artix System. As you can see the session manager service plug-in and the policy callback plug-in are both deployed into the same process. While in this example, they 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.

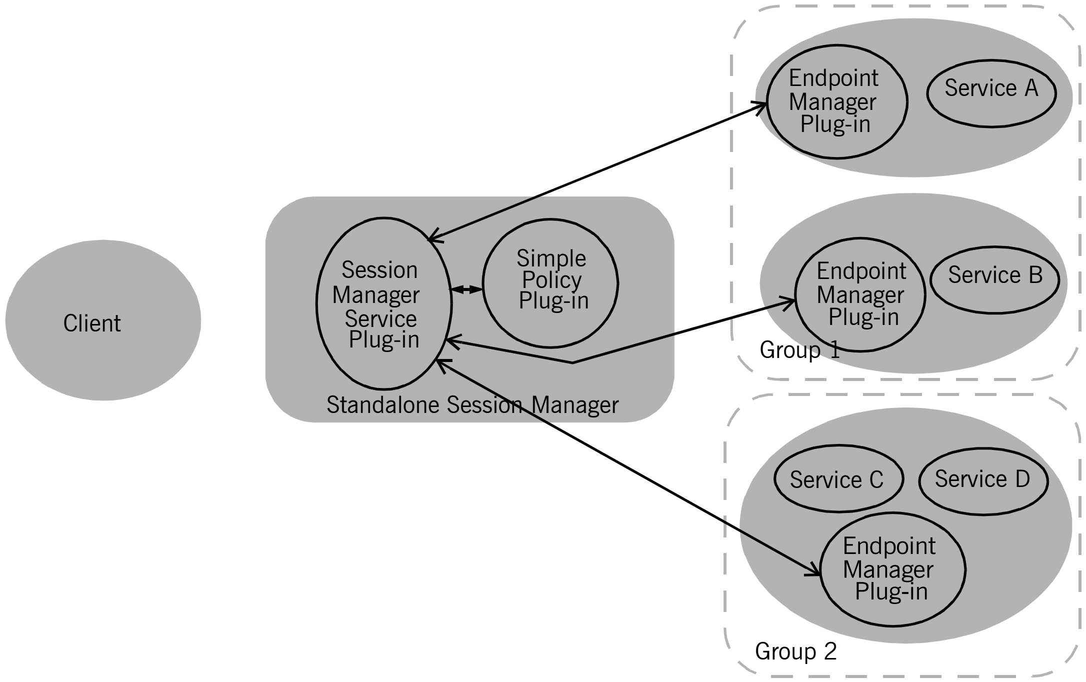

Figure 6: *The Session Manager Plug-ins*

The endpoint manager plug-ins are deployed into the server processes which contain session managed services. A process can host two services, like *Service C* and *Service D* in Figure 6, but the process will 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, to receive information on new sessions that have been granted to the managed services, and to check on the health of the session manager service.

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 water-slide group, it would ask the session manager for a session with the water-slide group. The session manager would then check and see if the water-slide group had an available session, and if so it would return a session id and the list of water-slide service references to the client. The session manager would then notify the endpoint managers in the water-slide group that a new session had been issued, the new session's id, and the duration for which the session is valid. When the client then makes requests on the services in the water-slide group, it must include the session information as part of the request. The endpoint manager for the services then check 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 water-slide services beyond the duration of its lease, the client will have to ask the session manager to renew its session before the session expires. Once a client's session has expired, it will have to 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, you edit the value for `plugins:session_endpoint_manager:default_group` in the process' configuration scope. For more information on Artix configuration see "Configuration" on page 5.

Deploying the Session Manager Service

Overview

Because the Artix session manager is implemented as a group of ART 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 users generate an Artix server that only hosts the session manager and deploy that server into the Artix environment.

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

Building a standalone session manager

To generate a standalone instance of the session manager you need to write a simple Artix server mainline and link it with the Artix libraries. Example 16 shows an example of the session manager's mainline.

Example 16: *Artix Session Manager Mainline*

```
include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_bus/fault_exception.h>

using namespace IT_Bus;
```

Example 16: *Artix Session Manager Mainline*

```
#int main(int argc, char* argv[])
{
  try
    {
      IT_Bus::Bus_var bus = IT_Bus::init(argc, argv,
                                           "managed_sessions");
      bus->run();
      bus->shutdown();
    }
  catch (IT_Bus::Exception& e)
    {
      printf("Exception occurred: %s", e.Message());
      return 1;
    }

  return 0;
}
```

The session manager's `main()` only needs to initialize the Artix bus with the name of the session manager's configuration scope and call `IT_Bus::run()`. The configuration scope name is third parameter to `IT_Bus::init()`, `managed_sessions`. The Artix bus will load the plug-ins for the session manager.

Example 17 shows a sample makefile for building the session manager.

Example 17: *Session Manager Makefile*

```
IT_PRODUCT_VER = 1.2

ART_BIN_DIR=$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\bin
ART_CXX_INCLUDE_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\i
   nclude"
ART_LIB_DIR="$(IT_PRODUCT_DIR)\artix\$(IT_PRODUCT_VER)\lib"

CXX=cl
CXXFLAGS=-I$(ART_CXX_INCLUDE_DIR) -Zi -nologo -GR -GX -W3 -Zm250
   -MD $(EXTRA_CXXFLAGS) $(CXXLOCAL_DEFINES)
```

Example 17: *Session Manager Makefile*

```
LINK=link
LDFLAGS=/DEBUG /NOLOGO
LDLIBS=/LIBPATH:$(ART_LIB_DIR) $(EXTRA_LIB_PATH) $(LINK_WITH)
   kernel32.lib ws2_32.lib advapi32.lib user32.lib

SHLIB_CXX_COMPILER_ID=          vc60
SHLIBLDFLAGS=-dll -debug -incremental:no

OBJS=$(SOURCES:.cxx=.obj)

LINK_WITH=it_bus.lib it_afc.lib it_art.lib  it_ifc.lib

SOURCES = session_manager.cxx
all: session_manager.exe

session_manager.exe:$(SOURCES) $(OBJS)
  if exist $@ del $@
    $(LINK) /out:$@ $(LDFLAGS) $(OBJS) $(LDLIBS)
```

The session manager must be linked with the following Artix libraries:

- `it_bus.lib`
- `it_afc.lib`
- `it_art.lib`
- `it_ifc.lib`

Configuring the session manager

To run the session manager you need to ensure that it loads the session manager service plug-in, `session_manager_service` and the session manager policy plug-in, `sm_simple_policy`. In addition, the session manager must load the `soap` and `http` plug-ins as all of its communication is done using SOAP over HTTP.

In the session manager's configuration scope you will need to specify the location for the session manager's contract 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 18 shows the configuration scope used to start the session manager.

Example 18: *Session Manager Configuration Scope*

```
managed_sessions
{
  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 Artix configuration see "Configuration" on page 5.

Generating the session manager's contact information

You also need to configure the port on which the session manager will run. To do this you modify `session-manager.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the session manager will listen. This can be either done manually for deploying the session manager on a well-known fixed port, or automatically for deploying the session manager on a dynamically allocated port.

Fixed Port

To deploy the session manager on a well-known fixed port, open `session-manager.wsdl` in any text editor and edit the `<soap:address>` entry for the `SessionManagerService` to specify the proper address. Example 19 shows a modified session manager contract entry. The highlighted part has been modified to point to the desired address.

Example 19: *Session Manager Address*

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

Dynamic Port

To deploy the session manager on a dynamically allocated port, configure the session manager to use the copy of `session-manager.wsdl` shipped with Artix.

You can limit the rang of ports on which the session manger will be deployed by specifying a rang of ports for the session managers SOAP or HTTP address. Example 20 shows a modified session manager contract entry. The highlighted part has been modified to specify to the desired range of ports.

Example 20: *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>
```

Once the session manager initializes the Artix bus, it will need to publish a new copy of its contract with the actual contact information. Example 21 shows how to publish the session manager's contract.

Example 21: *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();
```

Starting the session manager

Once the session manager has been generated and properly configured it can be started just like any other application. The only caveat 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 properly configure themselves. They do not require any special application code.

Once registered with a session manager, the services will 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` allows 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 will be managing 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 22 shows the configuration scope of a server that hosts services managed by the session manager.

Example 22: *Server Configuration Scope*

```
qajaq_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="qajaq_group";
 };
```

A server loaded into the `qajaq_server` configuration scope will be managed by the session manager at the location specified in `session-manager-service.wsdl`, its endpoint manager will come up at the address specified in `session-manager-endpoint.wsdl`, and by default all services instantiated by the server will belong to the session manager group `qajaq_group`.

For more information on Artix configuration see "Configuration" on page 5.

You also need to configure the port on which the endpoint manager will run. To do this you modify `session-manager.wsdl`, provided in the `wsdl` folder of your Artix installation, to specify the HTTP address at which the endpoint manager will be available. Using any text editor, open `session-manager.wsdl` and edit the `<soap:address>` entry for the `SessionEndpointManagerService` to specify the proper address. Example 23 shows a modified session manager contract entry. The highlighted part has been modified to point to the desired address.

Example 23: *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 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.

Registration

Once 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.

There are eight steps a client takes when making requests on a session managed service. They are:

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.

Instantiating 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 24 shows how to instantiate a session manager proxy.

Example 24: *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. `begin_session()` has the following signature:

```
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 107.

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 will be 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 will be set the configured max value. For more information see "Configuration" on page 5.

`output` contains the information needed to use the session.

Once a session is returned in `output`, you will need to extract the session ID to work with the session. This is done using `getsession_id()`. `getsession_id()` returns the session ID as an `IT_Bus_Services::SessionID`.

Example 25 shows the client code to begin a session for `qajaq_group`.

Example 25: *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("qajaq_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, so in order to get an individual service upon which to make requests 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. `get_all_endpoints()` has the following signature:

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

`request` contains the session ID for which you are requesting services. 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 26 shows how to get the list of services for a group.

Example 26: *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 first need to narrow down the list returned services to the desired one. `GetAllEnpointsResponse` contains an array of references to active services that can be retrieved using `GetAllEndpointsResponse`'s `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 the services registered with the session manager, the clients must be responsible for circulating through the list or else they will 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 27.

Example 27: *Checking the Service Reference for its Interface*

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

Example 28 shows the client code for creating a proxy qajaq server from a group service.

Example 28: *Instantiate a Proxy Server*

```
// C++
QajaqClient qajaq_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()`, that 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 29 shows how to set the client proxy port's `use_input_message_attributes` property to `true`.

Example 29: *Use Input Message Attributes*

```
//C++
// Get the proxy's port
IT_Bus::Port proxy_port = qajaq_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 30 shows how to get a handle to the input message attributes.

Example 30: *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 need to be set to include the proper 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 group name for which the session is valid. The session endpoints also use to ensure that the session is for the correct group. The session group is returned by invoking `getendpoint_group()` on 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 31 shows how to set the session information in to the input message attributes.

Example 31: *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

Once 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.

Renewing a session

If a client is going to use a session for a longer than the duration the session was granted, the client will need to renew its session or the session will timeout. A session is renewed using the session manager proxy's `renew_session()` method. `renew_session()` has the following signature:

```
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 `params`' `setsession_id()` method. The renewal duration is set using `params`' `setrenew_timeout()` method.

If the renewal is successful, `renewed` will return 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 32 shows how to end a session.

Example 32: *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 will ensure that the session will be freed up immediately. A session is ended using the session manager proxy's `end_session()` method. `end_session()` has the following signature:

```
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`' `setsession_id()` method.

Example 33 shows how to end a session.

Example 33: *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 read 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 will no longer be available and 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 "Configuration" on page 5.

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 "Configuration" on page 5.

CHAPTER 7

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

Artix, because it is built on IONA's flexible ART platform, can be embedded within any CORBA application implemented using IONA's Application Server Platform 6.0 or later without modifying any of the CORBA application's code. Embedding Artix is done by altering the application's configuration to load the required Artix plug-ins.

Embedding Artix into your CORBA application has several advantages:

- You do not need a separate process to route messages to the non-CORBA pieces of your application.
- You improve messaging performance over using the Artix standalone service.
- You can still code using a familiar paradigm and realize the benefits of using Artix.
- You can leverage all of the CORBA infrastructure to provide enterprise level qualities of service and management.

CORBA client applications

To embed Artix into a CORBA client application you need to 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 the configuration scope for your CORBA client 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 will be interacting with a sever using SOAP over WebSphere MQ your ORB plug-in list would be similar to the one in Example 34 on page 121. Note that the required Artix plug-ins for the SOAP binding, the WebSphere MQ transport, CORBA, and routing are highlighted.
3. Make an entry for `plugins:routing:wsdl_url` that specifies where the Artix applications contract resides.

In Example 34, the Artix contract describing the application is stored in `/artix/wsdlRepos/scoreBox.wsdl`.

Example 34: *Embedded Artix orb_plugin 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 proper ORB name to load the Artix plug-ins.

   For a client that uses the configuration shown in Example 34, you would start the client with the following command:

   ```
   client -ORBname corba_client.artix
   ```

CORBA server applications

To embed Artix into a CORBA server that uses the routing plug-in there are two caveats:

- Your CORBA server must generate persistent object references.
- Your CORBA server must run one time to export the persistent references and then be restarted for the Artix routing plug-in to work.

The routing plug-in requires valid object references to properly load itself and when embedded into 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.

Complete the following steps to configure a CORBA server to embed Artix:

1. Create an Artix contract that fully describes the interfaces, bindings, transports, and routing rules used in your Artix application.
2. Edit the configuration scope for your CORBA server 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 will be interacting with a client using SOAP over WebSphere MQ your ORB plug-in list would be similar to the one in Example 35 on page 122. Note that the required

Artix plug-ins for the SOAP binding, the WebSphere MQ transport, CORBA, and routing are highlighted.

3. Make an entry for `plugins:routing:wsdl_url` that specifies where the Artix applications contract resides.

   In Example 35, 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 35 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 35: *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 proper ORB name to load the Artix plug-ins.

   For a server that uses the configuration shown in Example 35, you would start the client with the following command:

   ```
   server -ORBname corba_server.artix
   ```

Using the CORBA Naming Service

Overview

In order to fully integrate with deployed CORBA systems, Artix can use a CORBA naming service that supports the `CosNaming` interface. Doing so requires editing the port information in the service's contract and modifying the Artix configuration.

Servers

To specify that an Artix instance (acting as proxy for a server) is to use the CORBA naming service, you edit the `<corba:address>` element of the CORBA port. In place of the file name used 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`.

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 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 "Servers" looks up the IOR for an object named `personalInfoService`.

Configuration

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, `IT_PRODUCT_DIR\artix\1.2\etc\artix.cfg`, in a text editor.
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 "Configuration" on page 5.

Load Balancing with CORBA

Overview

If an Artix SAP is mapped to a CORBA service, and that CORBA service is accessible via IONA's Application Server Platform 6.0 Service Pack 1 (or later), the implementation of that service can be load balanced using the Application Server Platform's locator service. In order to accomplish this, the Artix configuration file must duplicate some of the information from the Application Server Platform configuration domain, as described in the following steps.

For information on the load balancing feature of the Application Server Platform's load balancing features read the *Application Server Platform Administrator's Guide*.

Configuration Steps

The following steps work with an Application Server Platform 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 Application Server Platform domain exists, and that the locator service is run from this domain:

1. From the `domain.cfg` file, obtain the following configuration information and add it to `artix.cfg` file.

```
initial_references:IT_NodeDaemon:reference =
   "IOR:000000000000002149444c3a49545f4e6f64654461656d6f6e2f4e6f64654461656d6f6e3a312e3000000000
   0000000100000000000000760001020000000008686f726174696f00782800000000001d3a3e0233310c6e6f64655
   f6461656d6f6e000a4e6f64654461656d6f6e00000000000003000000010000001800000000000100010000000000
   0101040000000100010109000001a000000040100000000000000600000006000000000011";
```

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 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` will either result in an IOR that cannot be used for load balancing, or a process that outlives the POA.

To run such a ClusterDemo, you 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 Application Server Platform services

If your Application Server Platform services are replicated, and if Artix is deployed on each of the machines on which those services are replicated, then the Artix SAPs themselves can be replicated and load-balanced. For example,

1. On the "master" machine (e.g., 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`, `or 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

It is possible to 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 Application Server Platform 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 Application Server Platform domain configuration and put it in the `artix.cfg` file.
5. On each of the slave machines 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 Application Server Platform 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 Application Server Platform configuration file can also affect load balancing by specifying:

1. Persistent or Transient POA policy
2. Object ID

These load-balancing-related configuration values can be specified in an Artix WSDL contract using WSDL extensions for CORBA ports:

The POA name can be specified as follows:

```
<corba:policy poaname="my_poa_name"/>
```

The default POA name is `WSORB`.

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`.

The Service ID can be set as follows:

```
<corba:policy serviceid="ncname"/>
```

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`.

The following WSDL examples illustrate these points.

The contract fragment in Example 36 results in the following POA policy settings:

- PERSISTENT
- USER_ID
- POAName="master1"

- `ObjectID`="master1"

Example 36: *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>
```

The contract fragment in Example 37 results in the following POA policy settings:

- `TRANSIENT` (Default)
- `SYSTEM_ID` (Default)
- `POAName`="master1"

Example 37: *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>
```

The contract fragment in Example 38 results in a POA with the following policy settings:

- `TRANSIENT` (Default)
- `USER_ID`
- `POAName`="WSORB" (Default)
- `ObjectID`="master1"

Example 38: *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>
```

The contract fragment in Example 39 results in a POA with all default policies.

Example 39: *Default POA policies*

```
<service name="BaseService">
  <port binding="tns:BasePortCorbaBinding" name="BasePortCorba">
    <corba:address location="file://master.ref"/>
  </port>
</service>
```

Embedding Artix in a Tuxedo Container

Artix can be run and managed by Tuxedo like a native Tuxedo application.

Overview

In order to have Artix interact properly with native Tuxedo applications, you need to embed Artix into the Tuxedo container. At a minimum this involves adding information about Artix to your Tuxedo configuration file and registering your Artix processes with the Tuxedo bulletin board. You can also have Tuxedo bring up your Artix process as a Tuxedo server when running `tmboot`.

Procedure

To embed an Artix process into a Tuxedo container complete the following steps:

1. Ensure that your environment is properly configured for Tuxedo.
2. Add the Tuxedo plug-in, `tuxedo`, to your Artix process's `orb_plugins` list. See "ORB Plug-ins List" on page 14.

   ```
   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 into 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 `boingo`, you make the following entry in the `SERVERS` section:

   ```
   boingo SVRGRP=OINGO SVRID=1
   ```

   This associates `boingo` with the Tuxedo group called `OINGO` in your configuration and assigns boingo 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 process' Artix contract.

   For example, given the port definition shown in Example 40, the `SERVICES` entry would be `personalInfoService`.

Example 40: *Sample Service Entry*

```
<service name="personalInfoService">
  <port binding="tns:personalInfoBinding" name="tuxInfoPort">
    <tuxedo:server serviceName="personalInfoService" />
  </port>
</service>
```

7. If you made the Tuxedo configuration changes in the ASCII version of the configuration, `UBBCONFIG`, reload the `TUXCONFIG` with `tmload`.

Once you have properly configured Tuxedo, it will manage your Artix process as if it were a regular Tuxedo server.

Index