PROGRESS
SOFTWARE

Artix® ESB

Management Guide, C++ Runtime

Version 5.5, December 2008

Updated: February 20, 2009

Contents

List of Figures

Preface

What is covered in this book

This guide describes the enterprise management features for Artix applications that use the C++ runtime. It explains how to integrate and manage Artix applications with the following:

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

This guide applies to Artix applications written using in C++ or JAX-RPC (Java XML-Based APIs for Remote Procedure Calls) only.

For information on Artix applications written in JAX-WS (Java XML-Based APIs for Web Services) or JavaScript, see the Artix Management Guide, Java Runtime.

Who should read this book

This guide is aimed at system administrators managing distributed enterprise environments, and developers writing distributed enterprise applications. Administrators do not require detailed knowledge of the technology that is used to create distributed enterprise applications.ns.

This book assumes that you already have a good working knowledge of at least one of the management technologies mentioned in "What is covered in this book".

Organization of this book

This book contains the following parts:

Part I

- Chapter 1 introduces the Artix C++ management architecture and features.

Part II

- Chapter 2 introduces the JMX features supported by the Artix C++ runtime, and describes the Artix components that can be managed using JMX.
- Chapter 3 explains how to configure an Artix C++ runtime for JMX.
- Chapter 4 explains how to manage and monitor Artix services using JMX consoles.
- Chapter 5 shows how to manage Web Services Reliable Messaging persistence in Artix using a JMX console

Part III

- Chapter 6 describes the architecture of the Artix C++ runtime integration with Actional.
- Chapter 7 explains how to configure integration between Artix and Actional SOA management products.
- Chapter 8 shows examples of monitoring Artix services using Actional.

Part IV

- Chapter 9 describes the architecture of the Artix C++ runtime integration with AmberPoint.
- Chapter 10 explains how to configure integration with the Artix AmberPoint Agent, and shows examples from the Artix AmberPoint integration demo.

Part V

- Chapter 11 introduces Enterprise Management Systems, and the Artix integration with BMC Patrol.
- Chapter 12 describes how to configure your Artix environment for integration with BMC Patrol.
- Chapter 13 describes how to configure your BMC Patrol environment for integration with Artix.
- Chapter 14 describes how to extend an Artix BMC Patrol integration from a test environment to a production environment

The Artix Documentation Library

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

Part I

Introduction

In this part

This part contains the following chapters:

CHAPTER 1

Artix C++ Runtime Management

Artix provides support for integration with a range of management systems. This chapter introduces the management architecture for the Artix C++ runtime and the supported integrations.

In this chapter

This chapter includes the following section:

Introduction to Artix C++ Management

Overview

This section introduces the Artix ESB C++ runtime management architecture and explains its various components. This applies to Artix applications written in C++ and JAX-RPC.

Management architecture

The Artix ESB C++ management architecture provides:

- Integration with third-party enterprise management and SOA management systems
- Instrumentation used to monitor system status and potential problems
- Flexible runtime configuration
- Tools for developers without access to management systems.

Figure 1 shows a basic overview of the Artix C++ management architecture. The Artix C++ runtime uses Artix plug-ins and interceptors to send management instrumentation data to third-party management systems.

In addition, the Artix instrumentation data can also be monitored using JMX-compliant consoles.

Integration with third-party management systems

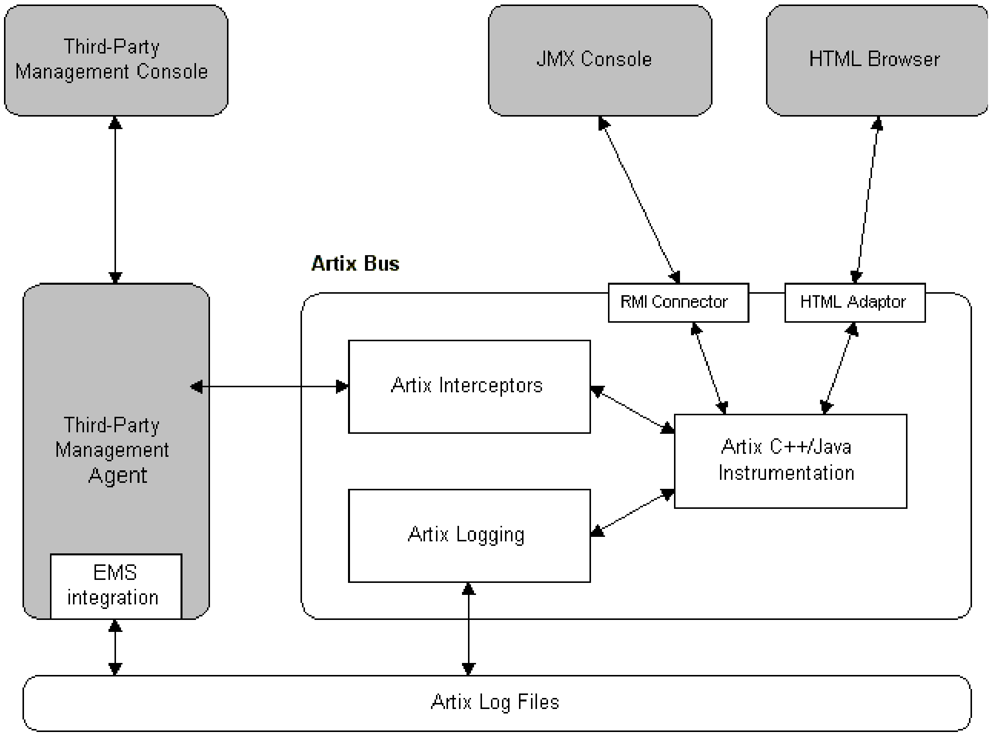
Integrations with third-party enterprise management and SOA management systems are critical to large corporations. Artix provides integration with the Actional and AmberPoint SOA management systems, and the BMC Patrol Enterprise Management System (EMS).

These management systems give a top-to-bottom view of enterprise infrastructure. For example, this means that instead of getting 100 different messages when services are not responding, you get a single message saying your services on these hosts are not working because the following network segment is dead.

If you integrate with an enterprise management or SOA management system, your product can also be hooked into higher-level monitoring tools such as Business Activity Monitoring (BAM), Service Level Agreement monitoring, and impact analysis tools. For example, when something goes wrong, the relevant administrators are automatically notified, trouble tickets are created, and service level impact is analyzed.

For more details on integration with third-party management systems, see "Artix C++ Management Integrations" on page 21.

Figure 1: *Artix C++ Runtime Management Architecture*



Instrumentation

Management *instrumentation* refers to application code used to monitor specific components in a system (for example, code that outputs logging or performance data to a management console). Instrumentation is used to reflect the state of a system and view potential problems with the normal operation of the system, while imposing minimal overhead. If you are using instrumentation to view problems, it is important that the act of observing the system causes minimal disturbance.

The main types of instrumentation supported by Artix include:

- Object-based instrumentation (for example, JMX)
- Logging

Object-based instrumentation

Artix supports object-based instrumentation using Java Management Extensions (JMX). The main purpose of this object-based instrumentation is to enable monitoring and management of Artix applications by JMX-aware third-party management consoles such as JConsole (see Figure 1).

Artix has been instrumented to allow Java runtime components to be exposed as JMX Managed Beans (MBeans). This enables an Artix Java runtime to be monitored and managed either in process or remotely using the JMX Remote API. Managed components are exposed using an `Object` interface with attributes and methods.

Artix Java runtime components can be exposed as JMX MBeans out-of-the-box (for example, Artix C++ service endpoints and Artix C++ bus). In addition, the Artix C++ runtime supports the registration of custom MBeans. Java developers can create their own MBeans and register them either with their JMX MBean server of choice, or with a default MBean server created by Artix

For more details on JMX object-based instrumentation, see Part II “Java Management Extensions”.

Logging

Logging in the Artix C++ runtime is controlled by the `event_log:filters` configuration variable, and by the log stream plug-ins. For example, the `local_log_stream` sends logging to a text file, and the `xmlfile_log_stream` directs logging to an XML file.

The `event_log:filters` configuration variable is used to specify logging severity levels—for information, warning, error, and fatal error messages. You can also use the `event_log:filters` variable to set fine-grained logging for specific Artix subsystems. For example, you can set logging for the Artix core, specific transports, bindings, or services. You can set logging for Artix services, such as the locator, and for services that you have implemented.

For more details on Artix C++ runtime logging, see Configuring and Deploying Artix Solutions, C++ Runtime.

Flexible configuration

The Artix C++ runtime is based on the highly flexible and scalable Adaptive Runtime (ART). This is a plug-in based architecture in which runtime behavior is configured using common and application-specific settings that are applied during application start up. This means that the same application code can be run, and can exhibit different capabilities, in different configuration environments.

You can change default behavior, enable specific functionality, or fine-tune behavior using a number of different configuration mechanisms. These include configuration file, command line, or programmatic configuration.

Artix configuration files are typically organized into a hierarchy of scopes, whose fully-qualified names map directly to Artix bus names. By organizing configuration variables into various scopes, you can provide different settings for individual services, or common settings for groups of services.

For more details on ART-based configuration, see Configuring and Deploying Artix Solutions, C++ Runtime.

Developer-based tools

Large corporations use third-party enterprise management and SOA management systems to monitor Artix applications in production environments. However, the following users need to use more lightweight management tools:

- Application developers who need to test the effects of their changes in a running test environment.
- Application developers who do not have access to an enterprise management or SOA management system.
- Support engineers who need to diagnose or correct problems raised by customers or management systems.

For facilitate such users, Artix provides out-of-the-box integration with JConsole. For more details, see "JMX" on page 21.

Artix C++ Management Integrations

Overview

Artix has been designed to integrate with a range of third-party management systems. These include enterprise management systems, SOA management systems, and developer-focused tools. This section introduces Artix integrations with the following systems:

- "JMX"
- "Progress Actional"
- "AmberPoint"
- "BMC Patrol"

JMX

The JMX instrumentation provided in Artix enables Artix service endpoints and the Artix bus to be monitored by any JMX-compliant management console (for example, JConsole or MC4J).

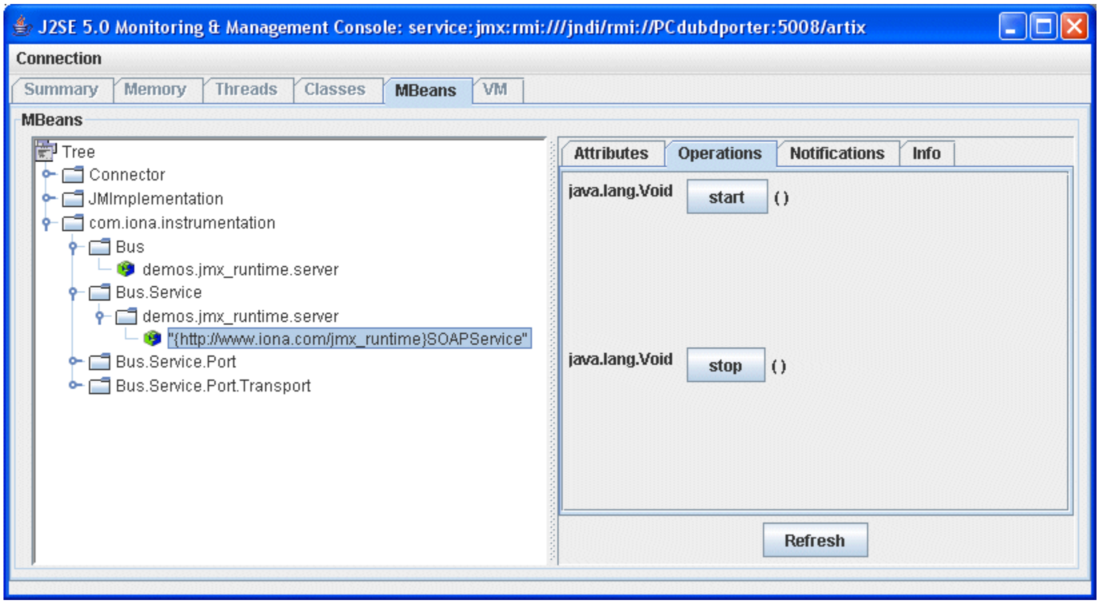
You can use JMX consoles to monitor and manage key Artix Java runtime components both locally and remotely. For example, using any JMX-compliant client, you can perform tasks such as:

- View service status
- View a service endpoint's address
- Stop or start a service
- Shutdown an Artix Java bus

Artix provides out-of-the-box integration with JConsole, which is the JMX-based management console provided with JDK 1.5.

Figure 2 shows an example Artix service endpoint monitored in JConsole. For more details on Artix integration with JMX, see Part II.

Figure 2: *Artix Service Endpoint in JConsole*



Progress Actional

Integration between Artix and Progress Actional enables Actional SOA management systems to monitor Artix services. For example, you can use Actional monitoring, auditing, and reporting on Artix services. You can also correlate and track messages through your network to perform dependency mapping and root cause analysis.

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

This integration uses the following components to monitor your services and report data back to the Actional SOA management tools:

Actional agents

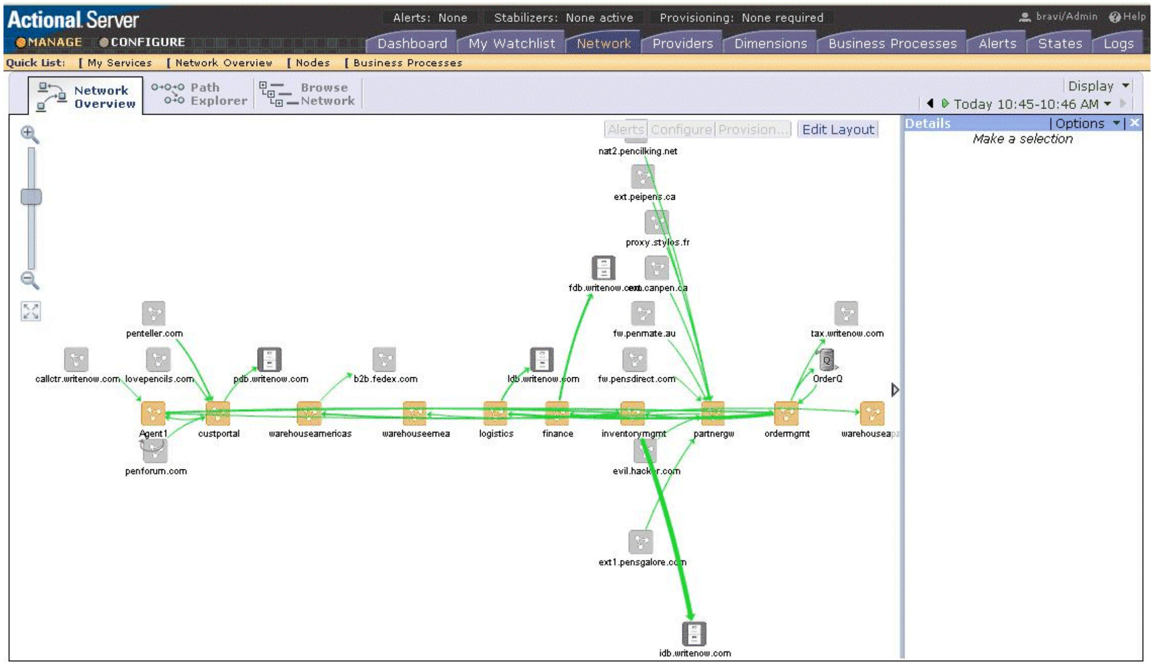
An Actional agent is run on each Artix node that you wish to manage. Actional agents are used to provide instrumentation data back to the Actional server. Actional agents are provisioned from the Actional server to establish initial contact and send configuration to the Actional agent.

Artix interceptors

Artix interceptors are added to an endpoint's messaging chain that send the instrumentation data to the Actional agent using an Actional-specific API. These interceptors essentially push events to the Actional agent. The data is analyzed and stored in the Actional agent for retrieval by the Actional server.

Figure 3 shows an example system monitored in the **Actional Server Administration Console**.

Figure 3: *Actional Server Administration Console*



For more details on Artix integration with Progress Actional, see Part III.

AmberPoint

Integration between Artix and AmberPoint enables the AmberPoint SOA management system to monitor Artix services. An Artix AmberPoint Agent can be deployed in Artix endpoints that use SOAP over HTTP to enable reporting of performance metrics back to AmberPoint.

The Artix AmberPoint Agent enables the use of the following AmberPoint features:

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

For more details on Artix integration with AmberPoint, see Part IV.

BMC Patrol

Integration between Artix and BMC Patrol enables the BMC Patrol Enterprise Management System (EMS) to monitor Artix services. You can use the Artix integration with BMC Patrol to track key server metrics, such as server response times. You can also set up alarms and post events when a server crashes to enable specific recovery actions to be taken.

The Artix ESB C++ runtime integration with BMC Patrol, key server metrics are logged by the Artix performance logging plug-ins. Log file interpreting utilities are then used to analyze the logged data. Artix provides BMC Knowledge Modules (KM), which conform to standard BMC Patrol KM design and operation. These modules tell the BMC Patrol console how to interpret the data obtained from the Artix interceptors.

The Artix server metrics tracked by the Artix BMC Patrol integration include the number of invocations received, and the average, maximum and minimum response times. The Artix BMC Patrol integration also enables you to track these metrics for individual operations. Events can be generated when any of these parameters go out of bounds.

For more details on Artix integration with BMC Patrol, see Part V.

Part II

Java Management Extensions

In this part

This part contains the following chapters:

Monitoring and Managing with JMX

This chapter explains how to monitor and manage an Artix C++ runtime using Java Management Extensions (JMX). It applies to applications written using both C++ and Java API for XML-Based Remote Procedure Call (JAX-RPC).

In this chapter

This chapter discusses the following topics:

Introduction

Overview

You can use Java Management Extensions (JMX) to monitor and manage key Artix runtime components both locally and remotely. For example, using any JMX-compliant client, you can perform the following tasks:

- View bus status.
- Stop or start a service.
- Change bus logging levels dynamically.
- Monitor service performance details.
- View the interceptors for a selected port.

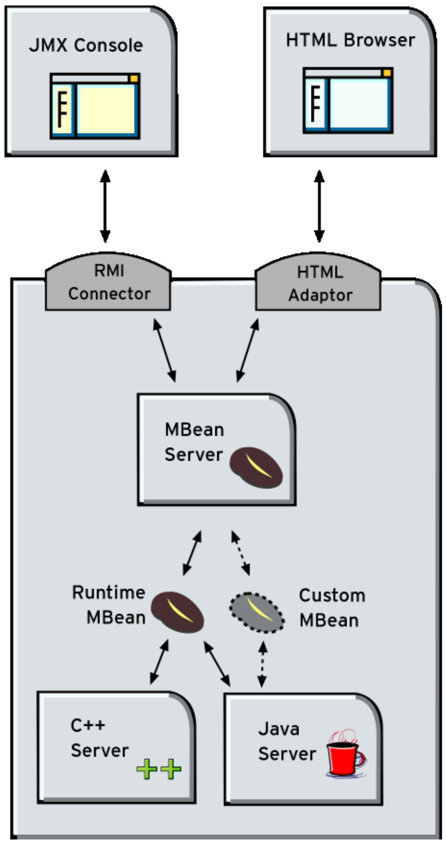
How it works

Artix has been instrumented to allow runtime components to be exposed as JMX Managed Beans (MBeans). This enables an Artix runtime to be monitored and managed either in process or remotely with the help of the JMX Remote API.

Artix runtime components can be exposed as JMX MBeans, out-of-the-box, for both C++ and JAX-RPC Artix servers. In addition, support for registering custom MBeans is also available. Java developers can create their own MBeans and register them either with their MBeanServer of choice, or with a default MBeanServer created by Artix (see "Relationship between runtime and custom MBeans" on page 30).

Figure 4 shows an overview of how the various components interact. The Java custom MBeans are optional components that can be added as required.

Figure 4: *Artix JMX Architecture*



What can be managed

Both Artix C++ and JAX-RPC servers can have their runtime components exposed as JMX MBeans. The following components can be managed:

- Bus
- Service
- Port

All runtime components are registered with an MBeanServer as Open Dynamic MBeans. This ensures that they can be viewed by third-party management consoles without any additional client-side support libraries.

All MBeans for Artix runtime components conform with Sun's JMX Best Practices document on how to name MBeans (see http://java.sun.com/products/JavaManagement/best-practices.html). Artix runtime MBeans use `com.iona.instrumentation` as their domain name when creating ObjectNames.

Note: An `MBeanServerConnection`, which is an interface implemented by the MBeanServer is used in the examples in this chapter. This ensures that the examples are correct for both local and remote access.

See also "Further information" on page 32 for details of how to access MBean Server hosting runtime MBeans either locally and remotely.

Relationship between runtime and custom MBeans

The Artix runtime instrumentation provides an out-of-the-box JMX view of C++ and JAX-RPC services. Java developers can also create custom JMX MBeans to manage Artix Java components such as services.

You may choose to write custom Java MBeans to manage a service because the Artix runtime is not aware of the current service's application semantics. For example, the Artix runtime can check service status and update performance counters, while a custom MBean can provide details on the status of a business loan request processing.

It is recommended that custom MBeans are created to manage application-specific aspects of a given service. Ideally, such MBeans should not duplicate what the runtime is doing already (for example, calculating service performance counters).

It is also recommended that custom MBeans use the same naming convention as Artix runtime MBeans. Specifically, runtime MBeans are named so that containment relationships can be easily established. For example:

```
// Bus :
com.iona.instrumentation:type=Bus,name=demos.jmx_runtime

Service :
com.iona.instrumentation:type=Bus.Service,name="{http://ws.iona.
    com}SOAPService",Bus=demos.jmx_runtime

// Port :
com.iona.instrumentation:type=Bus.Service.Port,name=SoapPort,Bus
    .Service="{http://ws.iona.com}SOAPService",Bus=demos.jmx_runt
    ime
```

Using these names, you can infer the relationships between ports, services and buses, and display or process a complete tree in the correct order. For example, assuming that you write a custom MBean for a loan approval Java service, you could name this MBean as follows:

```
com.iona.instrumentation:type=Bus.Service.LoanApprovalManager,na
    me=LoanApprovalManager,Bus.Service="{http://ws.iona.com}SOAPS
    ervice",Bus=demos.jmx_runtime
```

For details on how to write custom MBeans, see Developing Artix Applications in JAX-RPC.

Accessing the MBeanServer programmatically

Artix runtime support for JMX is enabled using configuration settings only. You do not need to write any additional Artix code. When configured, you can use any third party console that supports JMX Remote to monitor and manage Artix servers.

If you wish to write your own JMX client application, this is also supported. To access Artix runtime MBeans in a JMX client, you must first get a handle to the MBeanServer. The following code extract shows how to access the MBeanServer locally:

```
Bus bus = Bus.init(args);
MBeanServer mbeanServer =
    (MBeanServer)bus.getRegistry().getEntry(ManagementConstants.M
    BEAN_SERVER_INTERFACE_NAME);
```

The following shows how to access the MBeanServer remotely:

```
// The address of the connector server
String url = "service:jmx:rmi://host:1099/jndi/artix";
JMXServiceURL address = new JMXServiceURL(url);

// Create the JMXConnectorServer
JMXConnector cntor = JMXConnectorFactory.connect(address, null);

// Obtain a "stub" for the remote MBeanServer
MBeanServerConnection mbsc = cntor.getMBeanServerConnection();
```

Please see the following demo for a complete example on how to access, monitor and manage Artix runtime MBeans remotely:

```
InstallDir\cxx_java\samples\advanced\management\jmx_runtime
```

Further information

For further information, see the following URLs:

JMX

http://java.sun.com/products/JavaManagement/index.jsp

JMX Remote

http://www.jcp.org/aboutJava/communityprocess/final/jsr160/

Open Dynamic MBeans

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/openmbean/package-summary.html

ObjectName

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/ObjectName.html

MBeanServerConnection

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanServerConnection.html

MBeanServer

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanServer.html

Managed Bus Components

Overview

This section describes the attributes and methods that you can use to manage JMX MBeans representing Artix bus components. For example, you can use any JMX client to perform the following tasks:

- View bus attributes.
- Enable monitoring of bus services.
- Dynamically change logging levels for known subsystems.

If you wish to write your own JMX client, this section describes methods that you can use to access Artix logging levels and subsystems, and provides a JMX code example.

Bus MBean registration

When an Artix bus is initialized, a corresponding JMX MBean is created and registered for that bus with an MBeanServer.

JAX-RPC

For example, in an Artix Java application, this occurs after the following call:

```
String[] args = ...;
Bus serverBus = Bus.init(args);
```

C++

For example, in an Artix C++ application, this occurs after the following call:

```
Bus_var server_bus = Bus.init(argc, argv);
```

When a bus is shutdown, a corresponding MBean is unregistered from the MBeanServer.

Bus naming convention

An Artix bus `ObjectName` uses the following convention:

```
com.iona.instrumentation:type=Bus,name=busIdentifier
```

Bus attributes

The following bus component attributes can be managed by any JMX client:

Table 1: *Managed Bus Attributes*

Name	Description	Type	Read/Write
`scope`	Bus scope used to initialize a bus.	`String`	No
`identifier`	Bus identifier, typically the same as its scope.	`String`	No
`arguments`	Bus arguments, including the executable name.	`String[]`	No
`servicesMonitoring`	Used to enable/disable services performance monitoring.	`Boolean`	Yes
`services`	A list of object names representing services on this bus.	`ObjectName[]`	No

`servicesMonitoring` is a global attribute which applies to all services and can be used to change a performance monitoring status.

Note: By default, service performance monitoring is enabled when JMX management is enabled in a standalone server, and disabled in an `it_container` process.

When using a JMX console to manage a `it_container` server, you can enable performance monitoring by setting the `serviceMonitoring` attribute to `true`.

`services` is a list of object names that can be used by JMX clients to build a tree of components. Given this list, you can find all other registered service MBeans that belong to this bus.

For examples of bus attributes displayed in a JMX console, see "Managing Artix Services with JMX Consoles" on page 55.

Bus methods

If you wish to write your own JMX client, you can use the following bus methods to access logging levels and subsystems:

Table 2: *Managed Bus Methods*

Name	Description	Parameters	Return Type
`getLoggingLevel`	Returns a logging level for a subsystem.	`subsystem` (`String`)	`String`
`setLoggingLevel`	Sets a logging level for a subsystem.	`subsystem` (`String`), `level` (`String`)	`Boolean`
`setLoggingLevelPropagate`	Sets a logging level for a subsystem with propagation.	`subsystem` (`String`), `level` (`String`), `propagate` (`Boolean`)	`Boolean`

All the attributes and methods described in this section can be determined by introspecting `MBeanInfo` for the `Bus` component (see http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanInfo.htm)

Example JMX client

The following code extract from an example JMX client application shows how to access bus attributes and logging levels:

```
MBeanServerConnection mbsc = ...;
String busScope = ...;
ObjectName busName = new ObjectName("com.iona.instrumentation:type=Bus,name=" + busScope);

if (mbsc.isRegistered(busName)) {
    throw new MBeanException("Bus mbean is not registered");
}

// MBeanInfo can be used to check for all known attributes and methods
MBeanInfo info = mbsc.getMBeanInfo(busName);

// bus scope
String scope = (String)mbsc.getAttribute(busName, "scope");
// bus identifier
String identifier = (String)mbsc.getAttribute(busName, "identifier");
// bus arguments
String[] busArgs = (String[])mbsc.getAttribute(busName, "arguments");
```

```
// check servicesMonitoring attribute, then disable and reenable it
Boolean status = (Boolean)mbsc.getAttribute(busName, "servicesMonitoring");
if (!status.equals(Boolean.TRUE)) {
     throw new MBeanException("Service monitoring should be enabled by default");
}

mbsc.setAttribute(busName, new Attribute("servicesMonitoring", Boolean.FALSE));
status = (Boolean)mbsc.getAttribute(busName, "servicesMonitoring");
if (!status.equals(Boolean.FALSE)) {
    throw new MBeanException("Service monitoring should be disabled now");
}

mbsc.setAttribute(busName, new Attribute("servicesMonitoring", Boolean.TRUE));
status = (Boolean)mbsc.getAttribute(busName, "servicesMonitoring");
if (!status.equals(Boolean.TRUE)) {
    throw new MBeanException("Service monitoring should be reenabled now");
}

// list of service MBeans
ObjectName[] serviceNames = (ObjectName[])mbsc.getAttribute(busName, "services");

// logging
String level = (String)mbsc.invoke(
                               busName,
                               "getLoggingLevel",
                               new Object[] {"IT_BUS"},
                               new String[] {"subsystem"});
if (!level.equals("LOG_ERROR")) {
    throw new MBeanException("Wrong IT_BUS logging level");
}

level = (String)mbsc.invoke(
                          busName,
                          "getLoggingLevel",
                          new Object[] {"IT_BUS.INITIAL_REFERENCE"},
                          new String[] {"subsystem"});
if (!level.equals("LOG_ERROR")) {
    throw new MBeanException("Wrong IT_BUS.INITIAL_REFERENCE logging level");
}
level = (String)mbsc.invoke(
                          busName,
                          "getLoggingLevel",
                          new Object[] {"IT_BUS.CORE"},
                          new String[] {"subsystem"});
if (!level.equals("LOG_INFO_LOW")) {
    throw new MBeanException("Wrong IT_BUS.CORE logging level");
}
```

```
Boolean result = (Boolean)mbsc.invoke(
                          busName,
                          "setLoggingLevel",
                          new Object[] {"IT_BUS", "LOG_WARN"},
                          new String[] {"subsystem", "level"});

level = (String)mbsc.invoke(
                          busName,
                          "getLoggingLevel",
                          new Object[] {"IT_BUS"},
                          new String[] {"subsystem"});
if (!level.equals("LOG_WARN")) {
    throw new MBeanException("IT_BUS logging level has not been set properly");
}

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.INITIAL_REFERENCE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_WARN")) {
    throw new MBeanException("IT_BUS.INITIAL_REFERENCE logging level has not been set 
   properly");
}

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.CORE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_INFO_LOW")) {
    throw new MBeanException("IT_BUS.CORE logging level should not be changed");
}

// propagate
result =  (Boolean)mbsc.invoke(
                             busName,
                             "setLoggingLevelPropagate",
                             new Object[] {"IT_BUS", "LOG_SILENT", Boolean.TRUE},
                             new String[] {"subsystem", "level", "propagate"});

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS"},
                         new String[] {"subsystem"});
```

```
if (!level.equals("LOG_SILENT")) {
    throw new MBeanException("IT_BUS logging level has not been set properly");
}

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.INITIAL_REFERENCE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_SILENT")) {
    throw new Exception("IT_BUS.INITIAL_REFERENCE logging level has not been set properly");
}
level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.CORE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_SILENT")) {
    throw new MBeanException("IT_BUS.CORE logging level shouldve been set to LOG_SILENT");
}
```

Further information

For information on Artix logging levels and subsystems, see Configuring and Deploying Artix Solutions, C++ Runtime.

Managed Service Components

Overview

This section describes the attributes and methods that you can use to manage JMX MBeans representing Artix service components. For example, you can use any JMX client to perform the following tasks:

- View managed services.
- Dynamically change a service status.
- Monitor service performance data.
- Manage service ports.

The Artix locator and session manager services, have also been instrumented. These provide an additional set of attributes on top of those common to all services. For information on WS-RM persistence instrumentation, see Chapter 5.

If you wish to write your own JMX client, this section describes methods that you can use and provides a JMX code example.

Service MBean registration

When an Artix servant is registered for a service, a JMX Service MBean is created and registered with an MBeanServer.

JAX-RPC

For example, in an Artix Java application, this occurs after the following call:

```
Bus bus = Bus.init(args);

QName bankServiceName = new
    QName("http://www.iona.com/bus/tests", "BankService");
Servant servant = new SingleInstanceServant(new BankImpl(),
    serviceWsdlURL, bus);

bus.registerServant(servant, bankServiceName, "BankPort");
```

C++

For example, in an Artix C++ application, this happens after the following call:

```
Bus_var server_bus = Bus.init(argc, argv);

BankServiceImpl servant;
bus->register_servant(
    servant,
    wsdl_location,
    QName("http://www.iona.com/bus/tests", "BankService")
);
```

When a service is removed, a corresponding MBean is unregistered from the MBeanServer.

Service naming convention

An Artix service `ObjectName` uses the following convention:

```
com.iona.instrumentation:type=Bus.Service,name="{namespace}local
   name",Bus=busIdentifier
```

In this format, a `name` has an expanded service QName as its value. This value includes double quotes to permit for characters that otherwise would not be allowed.

Service attributes

The following service component attributes can be managed by any JMX client:

Table 3: *Managed Service Attributes*

Name	Description	Type	Read/Write
`name`	Service QName in expanded form.	`String`	No
`state`	Service state.	`String`	No
`serviceCounters`	Service performance data.	`CompositeData`	No
`ports`	A list of ObjectNames representing ports for this service.	`ObjectName[]`	No

`name` is an expanded QName, such as `{http://www.iona.com/bus/tests}BankService`.

`state` represents a current service state that can be manipulated by stop and start methods.

`ports` is a list of ObjectNames that can be used by JMX clients to build a tree of components. Given this list, you can find all other registered Port MBeans which happen to belong to this Service.

serviceCounters attributes

The following service performance attributes can be retrieved from the `serviceCounters` attribute:

Table 4: *serviceCounters Attributes*

Name	Description	Type
`averageResponseTime`	Average response time in milliseconds.	`Float`
`requestsOneway`	Total number of oneway requests to this service.	`Long`
`requestsSinceLastCheck`	Number of requests happened since last check.	`Long`
`requestsTotal`	Total number of requests (including oneway) to this service.	`Long`
`timeSinceLastCheck`	Number of seconds elapsed since last check.	`Long`
`totalErrors`	Total number of request-processing errors.	`Long`

For examples of service attributes displayed in a JMX console, see “Managing Artix Services with JMX Consoles” on page 55

Service methods

If you wish to write your own JMX client, you can use the following service methods to manage a specific service:

Table 5: *Managed Service Attributes*

Name	Description	Parameters	Return Type
`name`	Start (activate) a service.	None	`Void`
`state`	Stop (deactivate) a service.	None	`Void`

All the attributes and methods described in this section can be accessed by introspecting `MBeanInfo` for the Service component.

Example JMX client

The following code extract from an example JMX client application shows how to access service attributes and methods:

```
MBeanServerConnection mbsc = ...;

String busScope = ...;
ObjectName serviceName = new ObjectName("com.iona.instrumentation:type=Bus.Service" +
                     ",name=\"{http://www.iona.com/hello_world_soap_http}SOAPService\""
   +",Bus=" + busScope);

if (!mbsc.isRegistered(serviceName)) {
    throw new MBeanException("Service MBean should be registered");
}

// MBeanInfo can be used to check for all known attributes and methods
MBeanInfo info = mbsc.getMBeanInfo(serviceName);

// service name
String name = (String)mbsc.getAttribute(serviceName, "name");

// check service state attribute then reset it by invoking stop and start methods

String state = (String)mbsc.getAttribute(serviceName, "state");
if (!state.equals("ACTIVATED")) {
    throw new MBeanException("Service should be activated");
}

mbsc.invoke(serviceName, "stop", null, null);
```

```
state = (String)mbsc.getAttribute(serviceName, "state");
if (!state.equals("DEACTIVATED")) {
    throw new MBeanException("Service should be deactivated now");
}

mbsc.invoke(serviceName, "start", null, null);

state = (String)mbsc.getAttribute(serviceName, "state");
if (!state.equals("ACTIVATED")) {
    throw new MBeanException("Service should be activated again");
}

// check service counters

CompositeData counters = (CompositeData)mbsc.getAttribute(serviceName, "serviceCounters");
Long requestsTotal = (Long)counters.get("requestsTotal");
Long requestsOneway = (Long)counters.get("requestsOneway");
Long totalErrors = (Long)counters.get("totalErrors");
Float averageResponseTime = (Float)counters.get("averageResponseTime");
Long requestsSinceLastCheck = (Long)counters.get("requestsSinceLastCheck");
Long timeSinceLastCheck = (Long)counters.get("timeSinceLastCheck");

// ports
ObjectName[] portNames = (ObjectName[])mbsc.getAttribute(serviceName, "ports");
```

Further information

MBeanInfo

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanInfo.html

CompositeData

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/openmbean/CompositeData.html

Artix Locator Service

Overview

The Artix locator can also be exposed as a JMX MBean. A locator managed component is a service managed component that can be managed like any other bus service with the same set of attributes and methods. The Artix locator also exposes it own specifc set of attributes.

Locator attributes

An Artix locator MBean exposes the following locator-specific attributes:

Table 6: *Locator MBean Attributes*

Name	Description	Type
`registeredEndpoints`	Number of registered endpoints.	`Integer`
`registeredServices`	Number of registered services, less or equal to number of endpoints.	`Integer`
`serviceLookups`	Number of service lookup requests.	`Integer`
`serviceLookupErrors`	Number of service lookup failures.	`Integer`
`registeredNodeErrors`	Number of node (peer ping) failures.	`Integer`

Example JMX client

The following code extract from an example JMX client application shows how to access locator attributes and methods:

```
MBeanServerConnection mbsc = ...;
String busScope = ...;
ObjectName serviceName = new ObjectName("com.iona.instrumentation:type=Bus.Service" +
                     ",name=\"{http://ws.iona.com/2005/11/locator}LocatorService\""
   +",Bus=" + busScope);

// use common attributes and methods, see an example above

// Locator specific attributes
Integer regServices = (Integer)mbsc.getAttribute(serviceName, "registeredServices");
Integer endpoints = (Integer)mbsc.getAttribute(serviceName, "registeredEndpoints");
Integer nodeErrors = (Integer)mbsc.getAttribute(servicetName, "registeredNodeErrors");
Integer lookupErrors = (Integer)mbsc.getAttribute(serviceName, "serviceLookupErrors");
Integer lookups = (Integer)mbsc.getAttribute(serviceName, "serviceLookups");
```

Artix Session Manager Service

Overview

The Artix session manager can also be exposed as a JMX MBean. A session manager component is a service managed component that can be managed like any other bus service with the same set of attributes and methods. The Artix session manager also exposes it own specifc set of attributes.

Session manager attributes

An Artix session manager MBean exposes the following session manager-specific attributes:

Table 7: *Session Manager MBean Attributes*

Name	Description	Type
`registeredEndpoints`	Number of registered endpoints.	`Integer`
`registeredServices`	Number of registered services, less or equal to number of endpoints.	`Integer`
`serviceGroups`	Number of service groups.	`Integer`
`serviceSessions`	Number of service sessions	`Integer`

Example JMX client

The following code extract from an example JMX client application shows how to access session manager attributes and methods:

```
MBeanServerConnection mbsc = ...;
String busScope = ...;
ObjectName serviceName = new ObjectName("com.iona.instrumentation:type=Bus.Service" +

   ",name=\"{http://ws.iona.com/sessionmanager}SessionManagerService\"" +",Bus=" +
   busScope);
// use common attributes and methods, see an example above

// SessionManager specific attributes
Integer regServices = (Integer)mbsc.getAttribute(serviceName, "registeredServices");
Integer endpoints = (Integer)mbsc.getAttribute(serviceName, "registeredEndpoints");
Integer serviceGroups = (Integer)mbsc.getAttribute(serviceName, "serviceGroups");
Integer serviceSessions = (Integer)mbsc.getAttribute(serviceName, "serviceSessions");
```

Managed Port Components

Overview

This section describes the attributes that you can use to manage JMX MBeans representing Artix port components. For example, you can use any JMX client to perform the following tasks:

- Monitor managed ports.
- View message and request interceptors.

If you wish to write your own JMX client, this section also shows an example of accessing these attributes in JMX code.

Port MBean registration

Port managed components are typically created as part of a service servant registration. When service is activated, all supported ports will also be registered as MBeans.

When a service is removed, a corresponding Service MBean, as well as all its child Port MBeans are unregistered from the MBeanServer.

Naming convention

An Artix port `ObjectName` uses the following convention:

```
com.iona.instrumentation:type=Bus.Service.Port,name=portName,Bus
    .Service="{namespace}localname",Bus=busIdentifier
```

Port attributes

The following bus component attributes can be managed by any JMX client:

Table 8: *Supported Service Attributes*

Name	Description	Type	Read/Write
`name`	Port name.	`String`	No
`address`	Transport specific address representing an endpoint.	`String`	No
`interceptors`	List of interceptors for this port.	`String[]`	No

Table 8: *Supported Service Attributes*

Name	Description	Type	Read/Write
`transport`	An optional attribute representing a transport for this port.	`ObjectName[]`	No

interceptors

The `interceptors` attribute is a list of interceptors for a given port. Internally, `interceptors` is an instance of `TabularData` that can be considered an array/table of `CompositeData`. However, due to a current limitation of `CompositeData`, (no insertion order is maintained, which makes it impossible to show interceptors in the correct order), the interceptors are currently returned as a list of strings, where each `String` has the following format:

```
[name]: name [type]: type [level]: level [description]: optional
   description
```

In this format, *type* can be `CPP` or `Java`; *level* can be `Message` or `Request`.

It is most likely that this limitation will be fixed in a future JDK release, probably JDK 1.7 because the enhancement request has been accepted by Sun. In the meantime, interceptors details can be retrieved by parsing a returned `String` array.

For examples of port attributes displayed in a JMX console, see "Managing Artix Services with JMX Consoles" on page 55

Example JMX client

The following code extract from an example JMX client application shows how to access port attributes and methods:

```
MBeanServerConnection mbsc = ...;

String busScope = ...;
ObjectName portName = new ObjectName("com.iona.instrumentation:type=Bus.Service.Port" +
                     ",name=SoapPort" +

   ",Bus.Service=\"{http://www.iona.com/hello_world_soap_http}SOAPService\"" +",Bus=" +
   busScope);

if (!mbsc.isRegistered(portName)) {
    throw new MBeanException("Port MBean should be registered");
}

// MBeanInfo can be used to check for all known attributes and methods
MBeanInfo info = mbsc.getMBeanInfo(portName);

// port name
String name = (String)mbsc.getAttribute(portName, "name");

// port address
String address = (String)mbsc.getAttribute(portName, "address");

// check interceptors

String[] interceptors = (String[])mbsc.getAttribute(portName, "interceptors");
if (interceptors.length != 6) {
    throw new MBeanException("Number of port interceptors is wrong");
}

handleInterceptor(interceptors[0],
                  "MessageSnoop",
                  "Message",
                  "CPP");
handleInterceptor(interceptors[1],
                  "MessagingPort",
                  "Request",
                  "CPP");
handleInterceptor(interceptors[2],
                  "http://schemas.xmlsoap.org/wsdl/soap/binding",
                  "Request",
                  "CPP");
```

```
handleInterceptor(interceptors[3],
                  "TestInterceptor",
                  "Request",
                  "Java");
handleInterceptor(interceptors[4],
                  "bus_response_monitor_interceptor",
                  "Request",
                  "CPP");
handleInterceptor(interceptors[5],
                  "ServantInterceptor",
                  "Request",
                  "CPP");
```

For example, the `handleInterceptor()` function may be defined as follows:

```
private void handleInterceptor(String interceptor,
                               String name,
                               String level,
                               String type) throws Exception {
  if (interceptor.indexOf("[name]: " + name) == -1 ||
      interceptor.indexOf("[type]: " + type) == -1 ||
      interceptor.indexOf("[level]: " + level) == -1) {

      throw new MBeanException("Wrong interceptor details");
  }
  // analyze this interceptor further
}
```

CHAPTER 3

Configuring JMX in Artix C++

This chapter explains how to configure an Artix C++ runtime to be managed with Java Management Extensions (JMX).

In this chapter

This chapter discusses the following topic:

Artix JMX Configuration

Overview

This section explains the Artix configuration variable settings that you must configure to enable JMX monitoring of the Artix runtime, and access for remote JMX clients.

Enabling the management plugin

To expose the Artix runtime using JMX MBeans, you must enable a `bus_management` plug-in as follows:

```
jmx_local
{
   plugins:bus_management:enabled="true";
};
```

This setting enables local access to JMX runtime MBeans. The `bus_management` plug-in wraps runtime components into Open Dynamic MBeans and registers them with a local MBeanServer.

Configuring remote JMX clients

To enable remote JMX clients to access runtime MBeans, use the following configuration settings:

```
jmx_remote
{
   plugins:bus_management:enabled="true";
   plugins:bus_management:connector:enabled="true";
};
```

These settings allow for both local and remote access.

Specifying a remote access URL

Remote access is performed through JMX Remote, using an RMI Connector on a default port of `1099`. Using this configuration, you can use the following JNDI-based JMXServiceURL to connect remotely:

```
service:jmx:rmi:///jndi/rmi://host:1099/artix
```

Configuring a remote access port

To specify a different port for remote access, use the following configuration variable:

```
plugins:bus_management:connector:port="2000";
```

You can then use the following JMXServiceURL:

```
service:jmx:rmi:///jndi/rmi://host:2000/artix
```

Configuring a stub-based JMXServiceURL

You can also configure the connector to use a stub-based JMXServiceURL as follows:

```
jmx_remote_stub
{
   plugins:bus_management:enabled="true";
   plugins:bus_management:connector:enabled="true";
   plugins:bus_management:connector:registry:required="false";
};
```

See the javax.management.remote.rmi package for more details on remote JMX.

Publishing the JMXServiceURL to a local file

You can also request that the connector publishes its JMXServiceURL to a local file:

```
plugins:bus_management:connector:url:publish="true";
```

The following entry can be used to override the default file name:

```
plugins:bus_management:connector:url:file="../../service.url";
```

Further information

For further information, see the following:

RMI Connector

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/remote/rmi/RMIConnector.html

JMXServiceURL

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/remote/JMXServiceURL.html

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/remote/rmi/package-summary.html

Managing Artix Services with JMX Consoles

You can use third-party management consoles that support JMX Remote to monitor and manage Artix services (for example, JConsole and MC4J). You can view the status of a bus instance, stop or start a service, change bus logging levels, or view interceptor chains.

In this chapter

This chapter discusses the following topics:

Managing Artix Services with JConsole

Overview

You can use JConsole, which is provided with JDK 1.5, to monitor and manage Artix applications. JConsole displays Artix runtime managed components in a hierarchical tree, as shown in Figure 5.

Using JConsole

To use JConsole:

1. Start up JConsole using the following command:

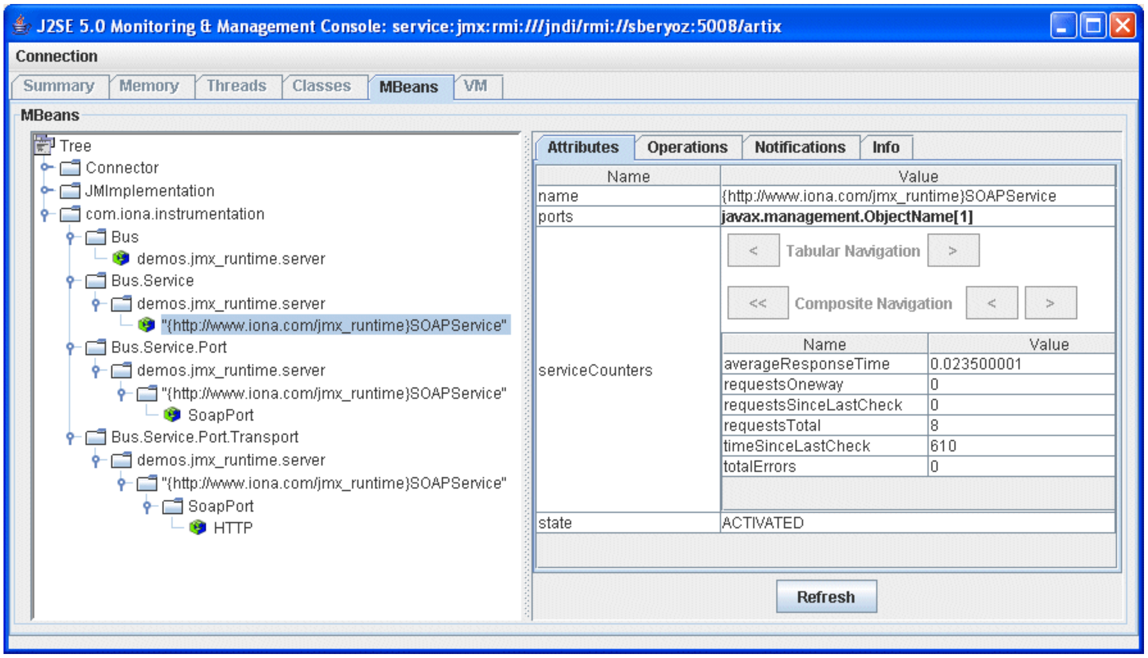
   ```
   JDK_HOME/bin/jconsole
   ```

2. Select the **Advanced** tab.
3. Enter or paste a JMXServiceURL (either the default URL, or one copied from a published `connector.url` file).

Managing services

Figure 5 shows the attributes displayed for a managed service component (for example, the `serviceCounters` performance metrics displayed in the right pane). For detailed information on these attributes, see "Service attributes" on page 40.

Figure 5: *Managed Service in JConsole*



J2SE 5.0 Monitoring & Management Console: service:jmx:rmi:///jndi/rmi://sberyoz:5008/artix

```
Tree
  Connector
  JMImplementation
  com.iona.instrumentation
    Bus
      demos.jmx_runtime.server
    Bus.Service
      demos.jmx_runtime.server
        "{http://www.iona.com/jmx_runtime}SOAPService"
    Bus.Service.Port
      demos.jmx_runtime.server
        "{http://www.iona.com/jmx_runtime}SOAPService"
          SoapPort
    Bus.Service.Port.Transport
      demos.jmx_runtime.server
        "{http://www.iona.com/jmx_runtime}SOAPService"
          SoapPort
            HTTP
```

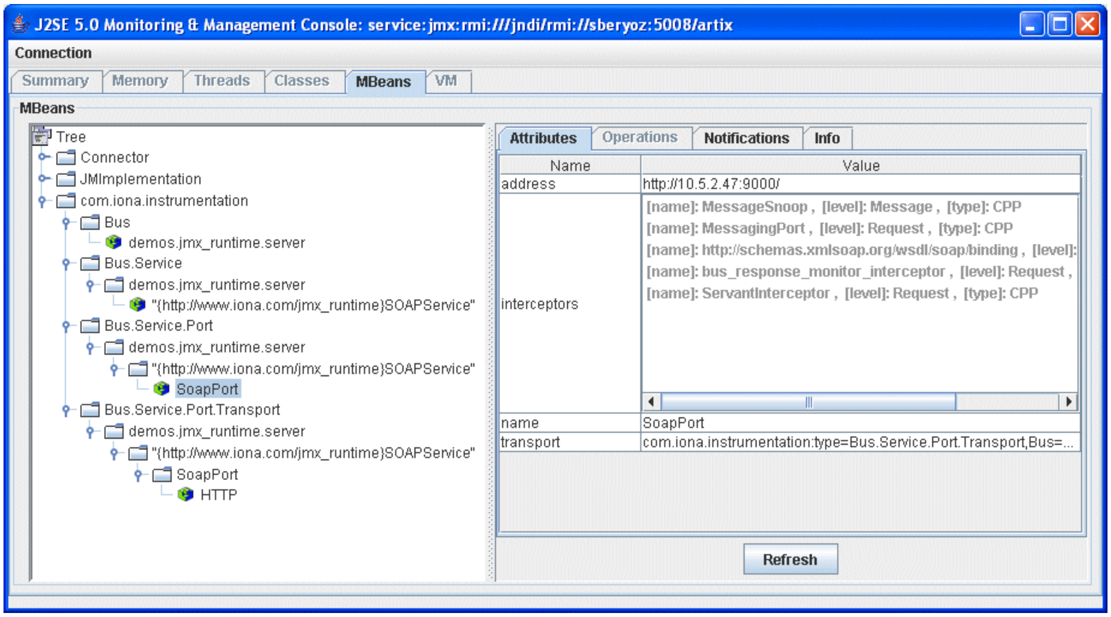
Name	Value
name	{http://www.iona.com/jmx_runtime}SOAPService
ports	**javax.management.ObjectName[1]**
serviceCounters	
state	ACTIVATED

Name	Value
averageResponseTime	0.023500001
requestsOneway	0
requestsSinceLastCheck	0
requestsTotal	8
timeSinceLastCheck	610
totalErrors	0

Managing ports

Figure 6 shows the attributes displayed for a managed port component (for example, the `interceptors` list displayed in the right pane). For detailed information on these attributes, see "Port attributes" on page 47.

Figure 6: *Managed Port in JConsole*



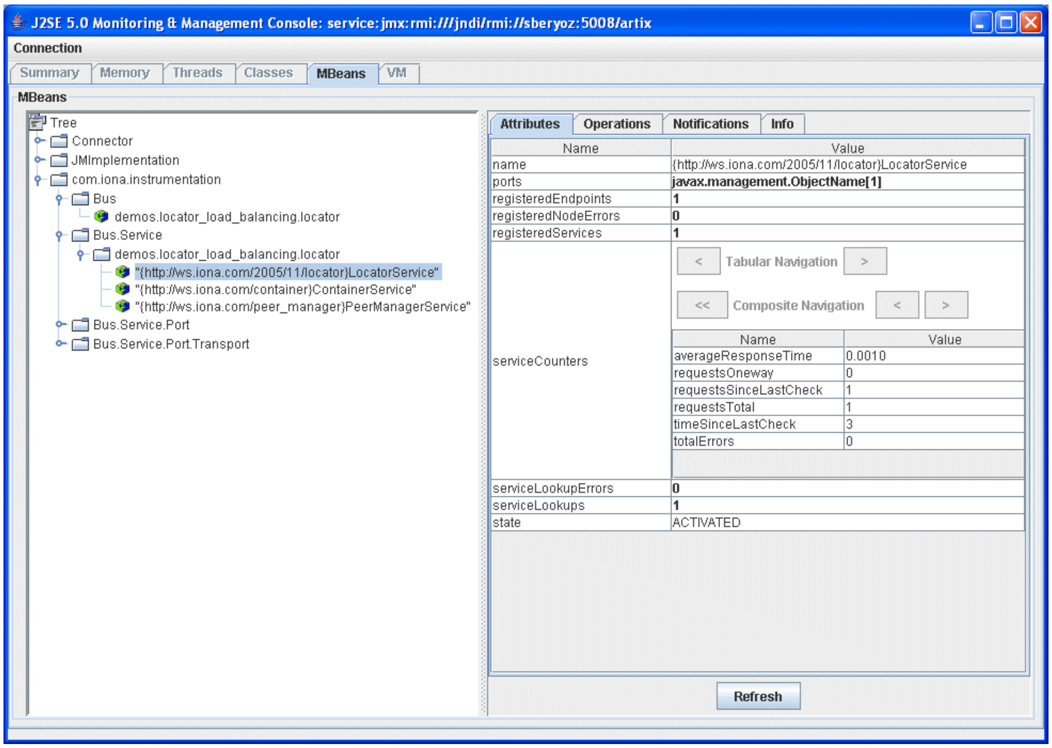
```
Tree
  Connector
  JMImplementation
  com.iona.instrumentation
    Bus
      demos.jmx_runtime.server
    Bus.Service
      demos.jmx_runtime.server
        "{http://www.iona.com/jmx_runtime}SOAPService"
    Bus.Service.Port
      demos.jmx_runtime.server
        "{http://www.iona.com/jmx_runtime}SOAPService"
          SoapPort
    Bus.Service.Port.Transport
      demos.jmx_runtime.server
        "{http://www.iona.com/jmx_runtime}SOAPService"
          SoapPort
            HTTP
```

Name	Value
address	http://10.5.2.47:9000/
interceptors	[name]: MessageSnoop , [level]: Message , [type]: CPP [name]: MessagingPort , [level]: Request , [type]: CPP [name]: http://schemas.xmlsoap.org/wsdl/soap/binding , [level]: [name]: bus_response_monitor_interceptor , [level]: Request , [name]: ServantInterceptor , [level]: Request , [type]: CPP
name	SoapPort
transport	com.iona.instrumentation:type=Bus.Service.Port.Transport,Bus=...

Managing containers

Figure 7 shows an example of a locator service deployed into an Artix container. For more information, see "Locator attributes" on page 44.

Figure 7: *Managed Locator in JConsole*



```
Tree
  Connector
  JMImplementation
  com.iona.instrumentation
    Bus
      demos.locator_load_balancing.locator
    Bus.Service
      demos.locator_load_balancing.locator
        "{http://ws.iona.com/2005/11/locator}LocatorService"
        "{http://ws.iona.com/container}ContainerService"
        "{http://ws.iona.com/peer_manager}PeerManagerService"
    Bus.Service.Port
    Bus.Service.Port.Transport
```

Name	Value
name	{http://ws.iona.com/2005/11/locator}LocatorService
ports	javax.management.ObjectName[1]
registeredEndpoints	1
registeredNodeErrors	0
registeredServices	1
serviceCounters	(see below)
serviceLookupErrors	0
serviceLookups	1
state	ACTIVATED

Name	Value
averageResponseTime	0.0010
requestsOneway	0
requestsSinceLastCheck	1
requestsTotal	1
timeSinceLastCheck	3
totalErrors	0

Note: When using a JMX console to manage a service running in an Artix container, set the `serviceMonitoring` attribute to `true` to enable service performance monitoring (see "Bus attributes" on page 34).

Further information

For more information on using JConsole, see the following:

http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

Managing Artix Services with the JMX HTTP adaptor

Overview

You can also manage Artix services using the default HTTP adaptor console that is provided with the JMX reference implementation. This console is browser-based, as shown in Figure 8.

Using the JMX HTTP adaptor

To use the JMX HTTP adaptor:

1. Specify following configuration settings:

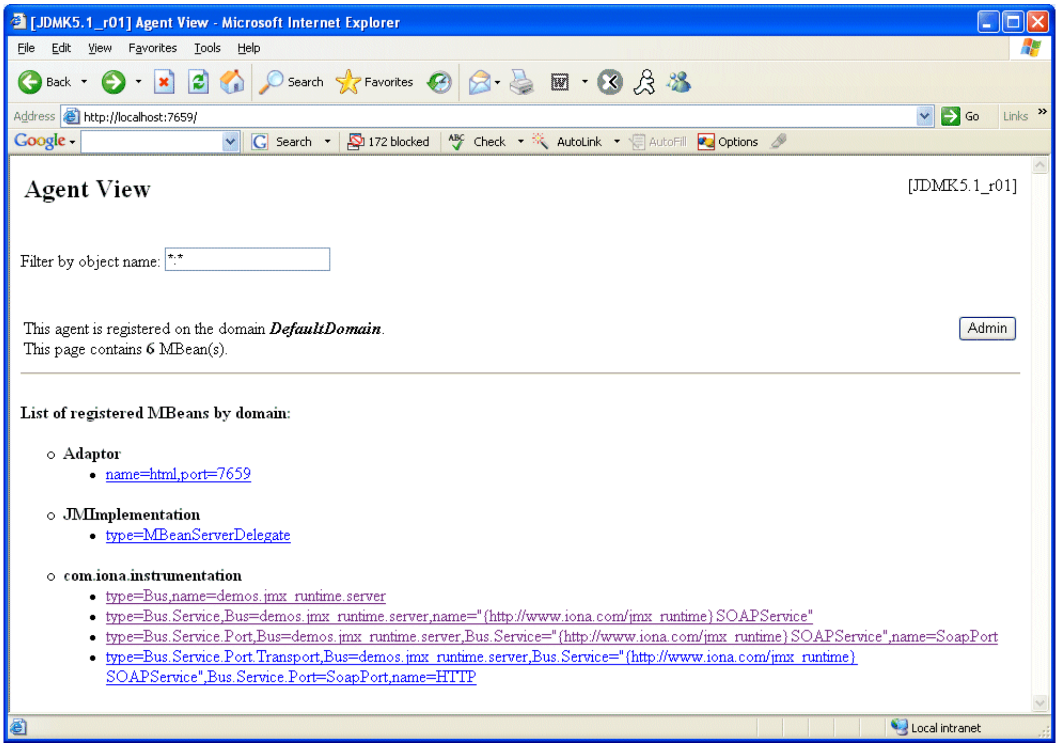
```
plugins:bus_management:http_adaptor:enabled="true";
plugins:bus_management:http_adaptor:port="7659";
```

2. Enter the following URL in your browser:

   `http://localhost:7659`

This displays the main HTTP adaptor management view, as shown in Figure 8.

Figure 8: *HTTP Adaptor Main View*


[JDMK5.1_r01] Agent View - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:7659/ Go Links
Google Search 172 blocked Check AutoLink AutoFill Options
Admin
Local intranet

Agent View [JDMK5.1_r01]

Filter by object name: *:*

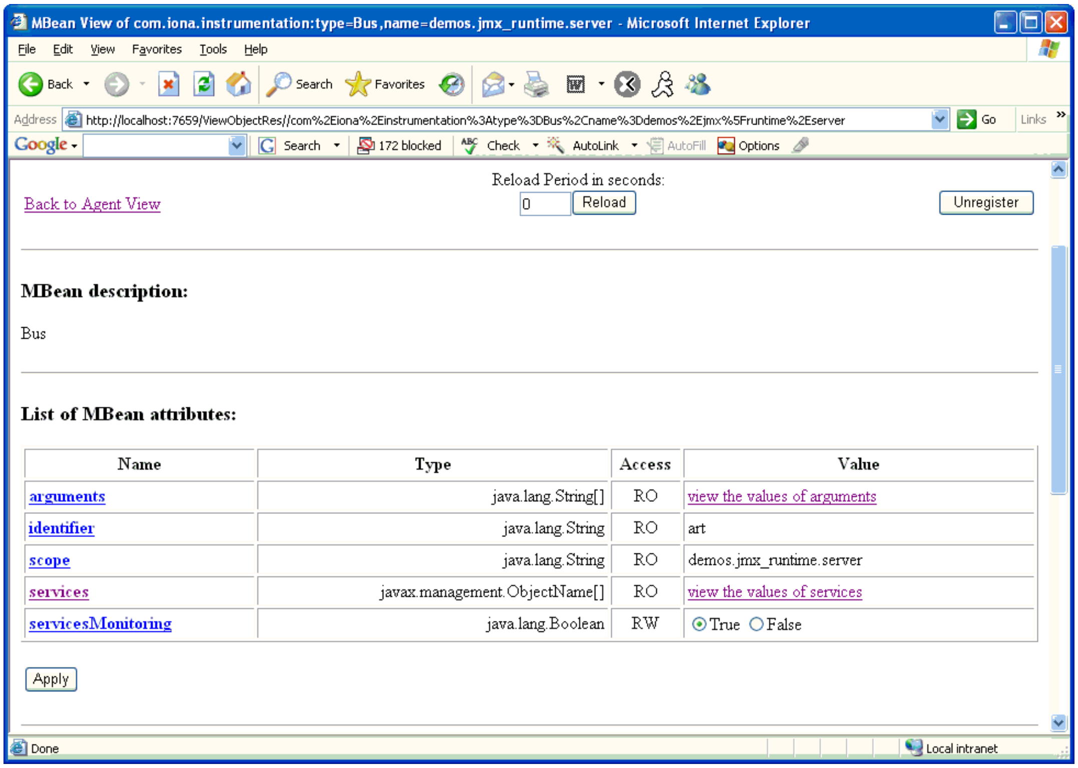
This agent is registered on the domain ***DefaultDomain***.
This page contains 6 MBean(s).

List of registered MBeans by domain:

- **Adaptor**
  - name=html,port=7659
- **JMImplementation**
  - type=MBeanServerDelegate
- **com.iona.instrumentation**
  - type=Bus,name=demos.jmx_runtime.server
  - type=Bus.Service,Bus=demos.jmx_runtime.server,name="{http://www.iona.com/jmx_runtime}SOAPService"
  - type=Bus.Service.Port,Bus=demos.jmx_runtime.server,Bus.Service="{http://www.iona.com/jmx_runtime}SOAPService",name=SoapPort
  - type=Bus.Service.Port.Transport,Bus=demos.jmx_runtime.server,Bus.Service="{http://www.iona.com/jmx_runtime}SOAPService",Bus.Service.Port=SoapPort,name=HTTP

Figure 9 shows the attributes displayed for a managed bus component (for example, the `services` that it includes). For detailed information on these attributes, see "Bus attributes" on page 34.

Figure 9: *HTTP Adaptor Bus View*


MBean View of com.iona.instrumentation:type=Bus,name=demos.jmx_runtime.server - Microsoft Internet Explorer

Address http://localhost:7659/ViewObjectRes//com%2Eiona%2Einstrumentation%3Atype%3DBus%2Cname%3Ddemos%2Ejmx%5Fruntime%2Eserver

Back to Agent View

Reload Period in seconds: 0 Reload

Unregister

MBean description:

Bus

List of MBean attributes:

Name	Type	Access	Value
arguments	java.lang.String[]	RO	view the values of arguments
identifier	java.lang.String	RO	art
scope	java.lang.String	RO	demos.jmx_runtime.server
services	javax.management.ObjectName[]	RO	view the values of services
servicesMonitoring	java.lang.Boolean	RW	[✓ True] False

Apply

Done | Local intranet

Further information

For further information on using the HTTP JMX adaptor, see the following:

http://java.sun.com/developer/technicalArticles/J2SE/jmx.html

Managing Artix Services with MC4J

Overview

You can use the open source MC4J management console to view service attributes and operations, stop or start a service, view interceptor chains, and change bus logging levels dynamically. This section uses the `jmx_runtime` Artix sample to show a detailed walk-through example of how to use MC4J to monitor and manage an Artix server.

Artix installs MC4J into the `*InstallDir*\cxx_java\mc4j` directory. This section uses the `jmx_runtime` Artix sample to show a detailed walk-through example of how to use MC4J to monitor and manage an Artix server.

Starting the MC4J console

To start the MC4J management console, perform the following steps:

1. Change directory to *InstallDir*`\cxx_java\bin`.
2. Run the following command:

Windows	`> start_mc4j.bat`
UNIX	`% ./start_mc4j`

Running the JMX sample

Before creating a new server connection in the MC4J console, do the following:

1. Change to the sample directory:

```
cd InstallDir\cxx_java\samples\advanced\management\jmx_runtime
```

2. Build the C++ or Java sample:

C++	`nmake`
Java	`ant`

3. Run the C++ or Java server:

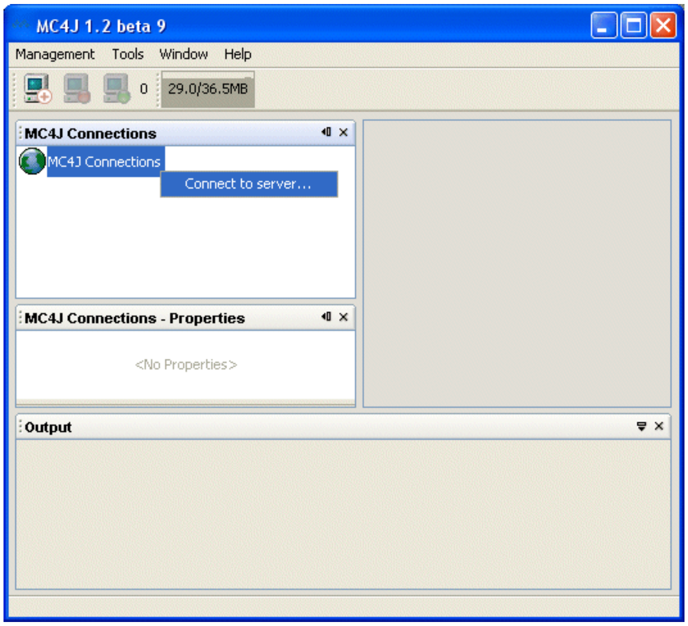
C++	`run_cxx_server.bat`
Java	`run_java_server.bat`

Creating a new server connection

To create a new server connection in the MC4J console:

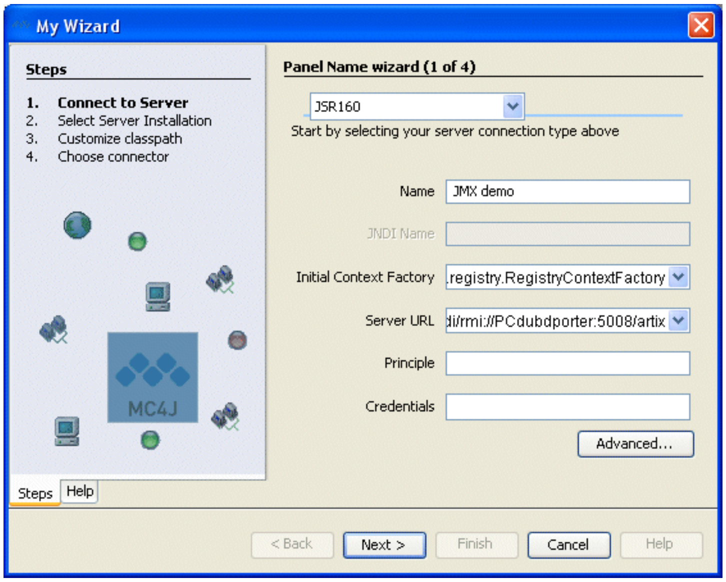
1. Select **MC4J Connections,** and right click, as shown in Figure 10.

Figure 10: *Connecting to a Server*



2. Click **Connection server…** to launch the **My wizard** dialog, as shown in Figure 11.

Figure 11: *Server Connection Details*



Panel Name wizard (1 of 4)

JSR160

Start by selecting your server connection type above

Name: JMX demo

JNDI Name:

Initial Context Factory: .registry.RegistryContextFactory

Server URL: di/rmi://PCdubdporter:5008/artix

Principle:

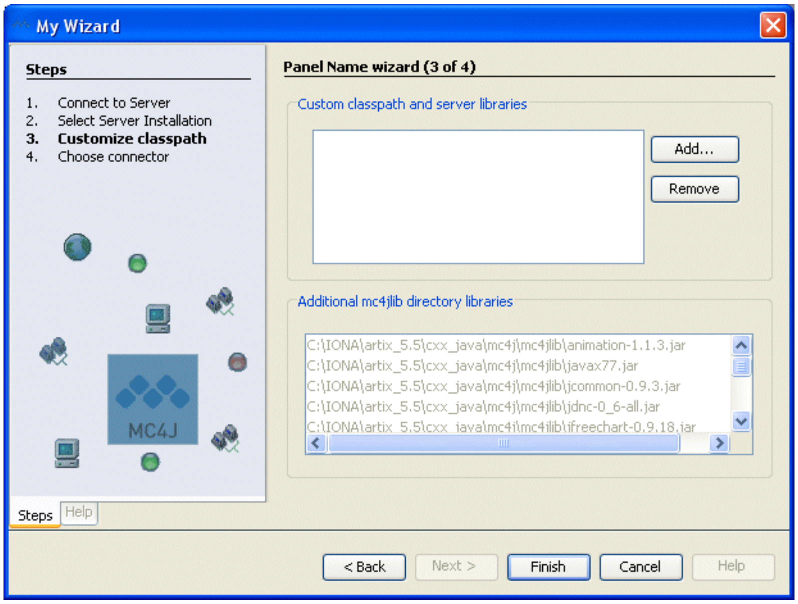
Credentials:

3. In the **My Wizard** dialog, select `JSR160` as your server connection type.
4. Enter `JMX demo` as your connection **Name**.
5. Enter the contents of the following file as the **Server URL**:

   `InstallDir``/cxx_java/samples/advanced/management/jmx_runtime/etc/connector.url`

6. Click **Next** to go to the next screen, as shown in Figure 12.

Figure 12: *Creation of Server Connection*



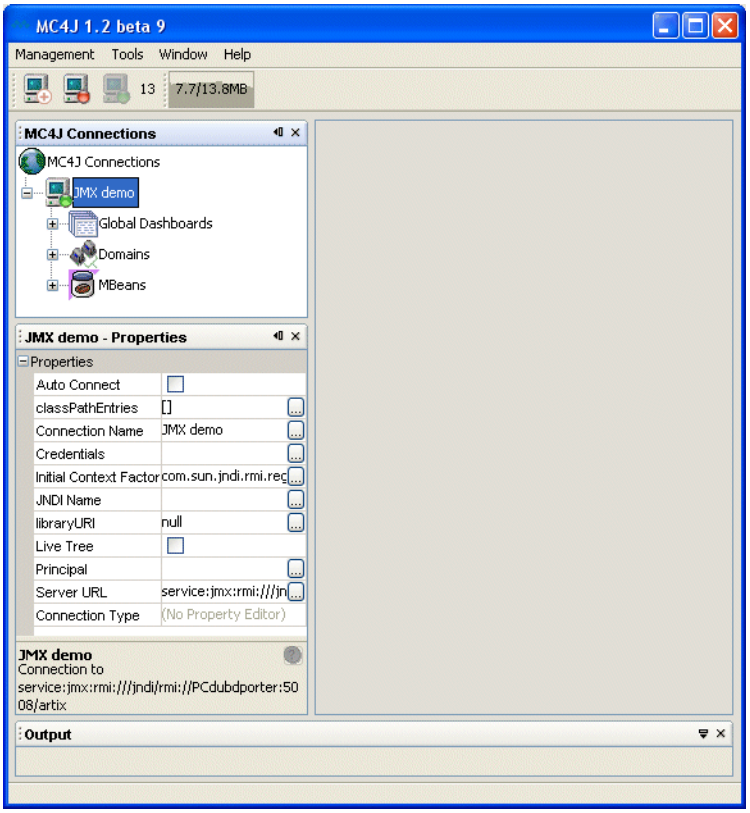
My Wizard
Steps
1. Connect to Server
2. Select Server Installation
3. Customize classpath
4. Choose connector
MC4J
Steps
Help
Panel Name wizard (3 of 4)
Custom classpath and server libraries
Add...
Remove
Additional mc4jlib directory libraries
< Back
Next >
Finish
Cancel
Help

C:\IONA\artix_5.5\cxx_java\mc4j\mc4jlib\animation-1.1.3.jar
C:\IONA\artix_5.5\cxx_java\mc4j\mc4jlib\javax77.jar
C:\IONA\artix_5.5\cxx_java\mc4j\mc4jlib\jcommon-0.9.3.jar
C:\IONA\artix_5.5\cxx_java\mc4j\mc4jlib\jdnc-0_6-all.jar
C:\IONA\artix_5.5\cxx_java\mc4j\mc4jlib\jfreechart-0.9.18.jar

7. Click **Finish** to finish the creation of a new server connection.

8. In the left panel of the MC4J console, a new server connection named `JMX demo` is created, as shown in Figure 13:

Figure 13: *New Server Connection*



Properties

Auto Connect	☐
classPathEntries	[]
Connection Name	JMX demo
Credentials	
Initial Context Factor	com.sun.jndi.rmi.reg
JNDI Name	
libraryURI	null
Live Tree	☐
Principal	
Server URL	service:jmx:rmi:///jn
Connection Type	(No Property Editor)

JMX demo
Connection to
service:jmx:rmi:///jndi/rmi://PCdubdporter:5008/artix

Monitoring and managing a service

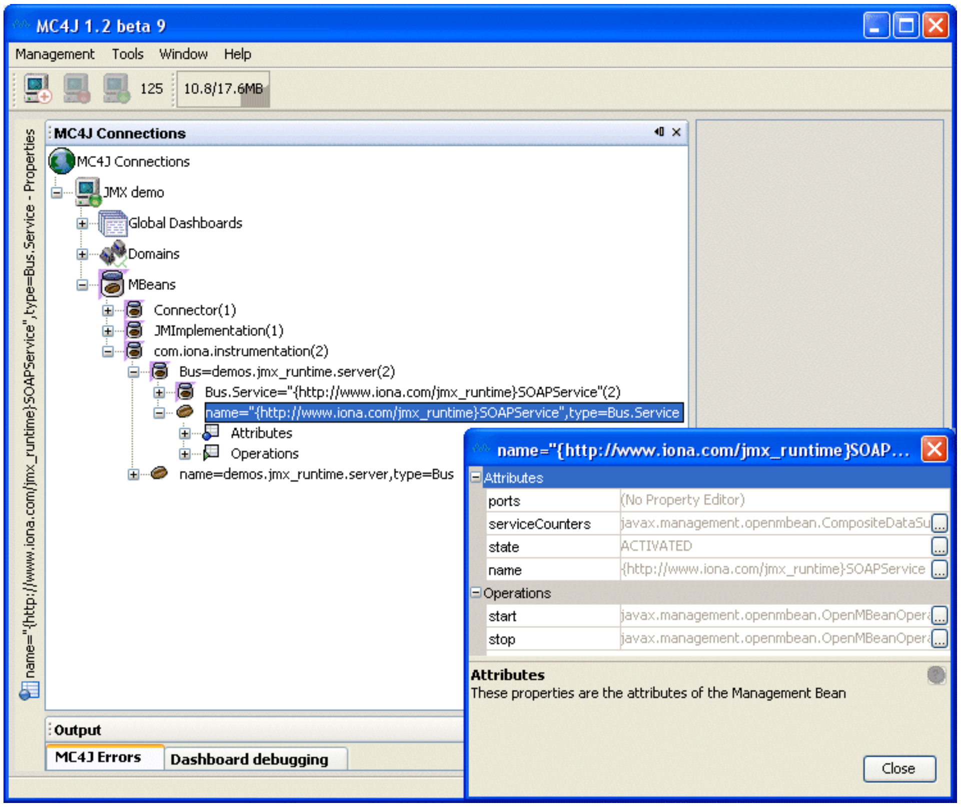
To monitor and manage an example service in the Mc4J console:

1. Expand the **MBeans** tree node in the left panel of MC4J.
2. Double click on the following tree node, as shown in Figure 14:

   ```
   name='{http://www.iona.com/jmx_runtime}SOAPService',type=Bus.
   Service
   ```

   This displays the attributes and operations of the `SOAPService` in the service properties dialog.

Figure 14: *Viewing Service Properties*



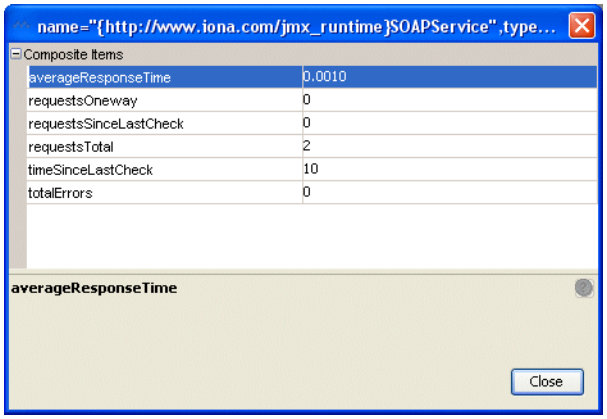
MC4J 1.2 beta 9
Management Tools Window Help
125 10.8/17.6MB
name="{http://www.iona.com/jmx_runtime}SOAPService",type=Bus.Service - Properties
MC4J Connections
MC4J Connections
JMX demo
Global Dashboards
Domains
MBeans
Connector(1)
JMImplementation(1)
com.iona.instrumentation(2)
Bus=demos.jmx_runtime.server(2)
Bus.Service="{http://www.iona.com/jmx_runtime}SOAPService"(2)
name="{http://www.iona.com/jmx_runtime}SOAPService",type=Bus.Service
Attributes
Operations
name=demos.jmx_runtime.server,type=Bus
Output
MC4J Errors
Dashboard debugging
name="{http://www.iona.com/jmx_runtime}SOAP...
Close

Attributes	
ports	(No Property Editor)
serviceCounters	javax.management.openmbean.CompositeDataSu
state	ACTIVATED
name	{http://www.iona.com/jmx_runtime}SOAPService
Operations	
start	javax.management.openmbean.OpenMBeanOper
stop	javax.management.openmbean.OpenMBeanOper

Attributes
These properties are the attributes of the Management Bean

3. Click the **...** button at the right of the `serviceCounters` attribute in the service properties dialog. This displays the details for the `serviceCounters` attribute, as shown in Figure 15.

Figure 15: *Viewing Service Counters Properties*



name="{http://www.iona.com/jmx_runtime}SOAPService",type...

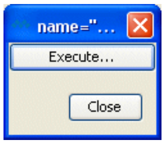
Composite Items

averageResponseTime	0.0010
requestsOneway	0
requestsSinceLastCheck	0
requestsTotal	2
timeSinceLastCheck	10
totalErrors	0

averageResponseTime

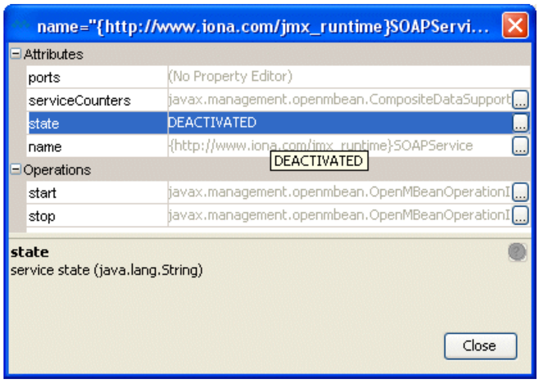
4. Click the **...** button at right of the **stop** operation on the service properties dialog. This displays a dialog for the **stop** operation, as shown in Figure 16.

Figure 16: *Stopping a Service*



5. Click **Execute...** to stop the service. In the `SOAPservice` properties dialog, the state attribute of the service becomes `DEACTIVATED`, as shown in Figure 17.

Figure 17: *Deactivated Service*


name="{http://www.iona.com/jmx_runtime}SOAPServi...

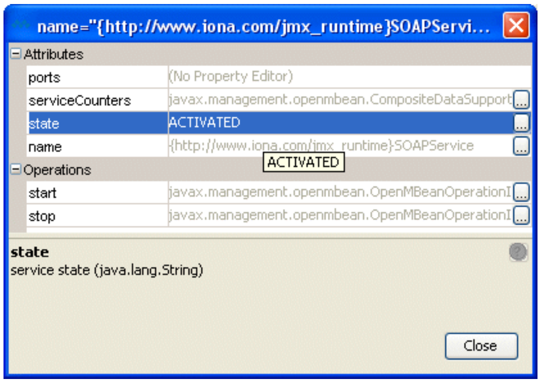
Attributes	
ports	(No Property Editor)
serviceCounters	javax.management.openmbean.CompositeDataSupport
state	DEACTIVATED
name	{http://www.iona.com/jmx_runtime}SOAPService
Operations	
start	javax.management.openmbean.OpenMBeanOperationI
stop	javax.management.openmbean.OpenMBeanOperationI

state
service state (java.lang.String)

6. Click the **...** button at the right of **start** operation on SOAP service properties. This displays a dialog for the **start** operation, which is the same as the one shown in Figure 16.

7. Click **Execute…** to restart the service. In the service properties dialog, the state of the `SOAPService` becomes `ACTIVATED`, as shown in Figure 18.

Figure 18: *Activated a Service*


name="{http://www.iona.com/jmx_runtime}SOAPServi…

Attributes

ports	(No Property Editor)
serviceCounters	javax.management.openmbean.CompositeDataSupport
state	ACTIVATED
name	{http://www.iona.com/jmx_runtime}SOAPService

ACTIVATED

Operations

start	javax.management.openmbean.OpenMBeanOperationI
stop	javax.management.openmbean.OpenMBeanOperationI

state
service state (java.lang.String)

Monitoring a service port

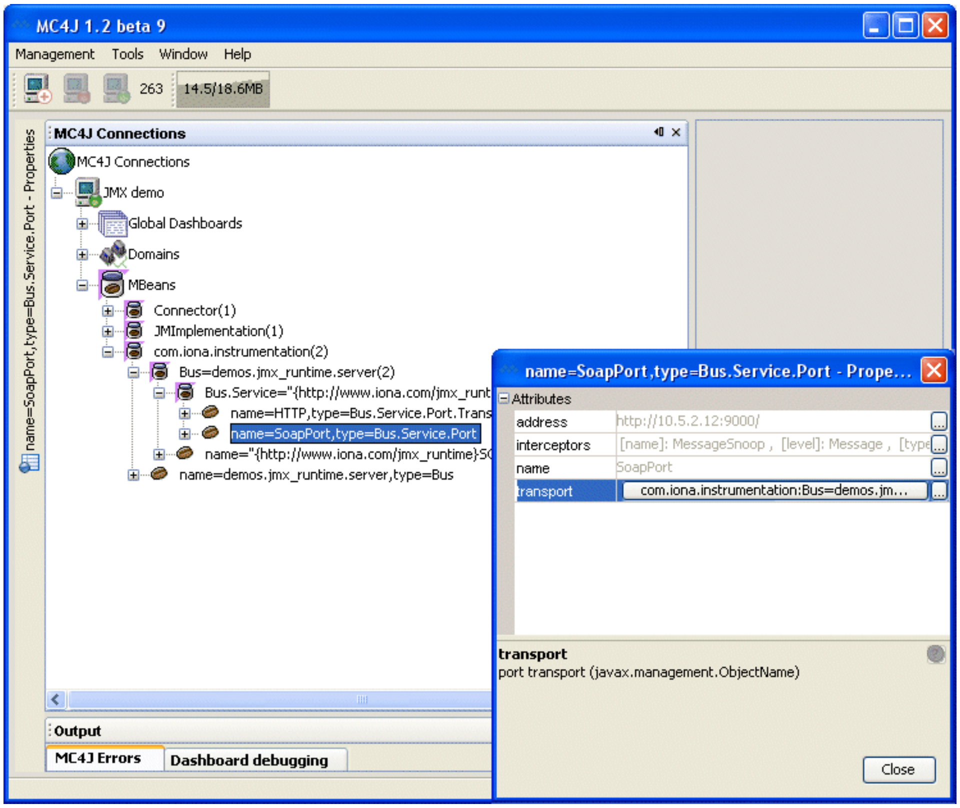
To monitor an example service port in the Mc4J console:

1. Click the following node in the left panel of the MC4J console:

   `name=SoapPort,tyoe=Bus.Service.Port`

   This displays the attributes for `SoapPort`, as shown in Figure 19.

Figure 19: *Viewing Port Properties*

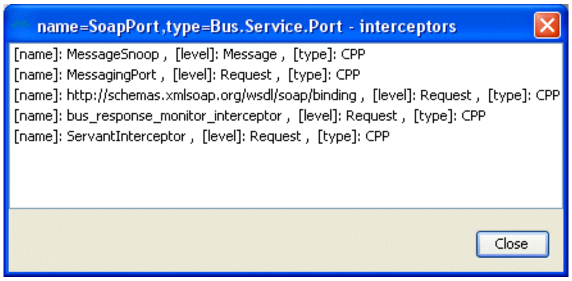

MC4J 1.2 beta 9
Management Tools Window Help
263 14.5/18.6MB
MC4J Connections
MC4J Connections
JMX demo
Global Dashboards
Domains
MBeans
Connector(1)
JMImplementation(1)
com.iona.instrumentation(2)
Bus=demos.jmx_runtime.server(2)
Bus.Service="{http://www.iona.com/jmx_runt
name=HTTP,type=Bus.Service.Port.Trans
name=SoapPort,type=Bus.Service.Port
name="{http://www.iona.com/jmx_runtime}S
name=demos.jmx_runtime.server,type=Bus
name=SoapPort,type=Bus.Service.Port - Properties
Output
MC4J Errors
Dashboard debugging
name=SoapPort,type=Bus.Service.Port - Prope...
Attributes
Close

address	http://10.5.2.12:9000/
interceptors	[name]: MessageSnoop , [level]: Message , [type
name	SoapPort
transport	com.iona.instrumentation:Bus=demos.jm...

transport
port transport (javax.management.ObjectName)

2. Click the **...** button at the right of the `interceptors` attribute in Figure 19. This displays the `interceptors` properties for the selected bus, as shown in Figure 20.

Figure 20: *Viewing Interceptor Properties*



```
[name]: MessageSnoop , [level]: Message , [type]: CPP
[name]: MessagingPort , [level]: Request , [type]: CPP
[name]: http://schemas.xmlsoap.org/wsdl/soap/binding , [level]: Request , [type]: CPP
[name]: bus_response_monitor_interceptor , [level]: Request , [type]: CPP
[name]: ServantInterceptor , [level]: Request , [type]: CPP
```

Further information

For full details on using the MC4J management console, see the MC4J documentation:

http://mc4j.org/confluence/display/MC4J/User+Guide

Managing Logging Levels with MC4J

Overview

This section uses the jmx_runtime Artix sample to show a detailed walk-through example of how to use the MC4J console to manage Artix bus logging levels dynamically at runtime.

Defined demo logging configuration

The following logging configuration is defined in the demos.jmx_runtime configuration scope:

Logging Subsystem	Logging Level
IT_BUS	FATAL+ERROR
IT_BUS.CORE	INFO

This means that the logging level for IT_BUS, and all of its child subsystems, is to display errors only. The only exception is IT_BUS.CORE, where logging is set to display informational messages.

Viewing logging levels for a subsystem

To view logging levels for a specified Artix logging subsystem in MC4J:

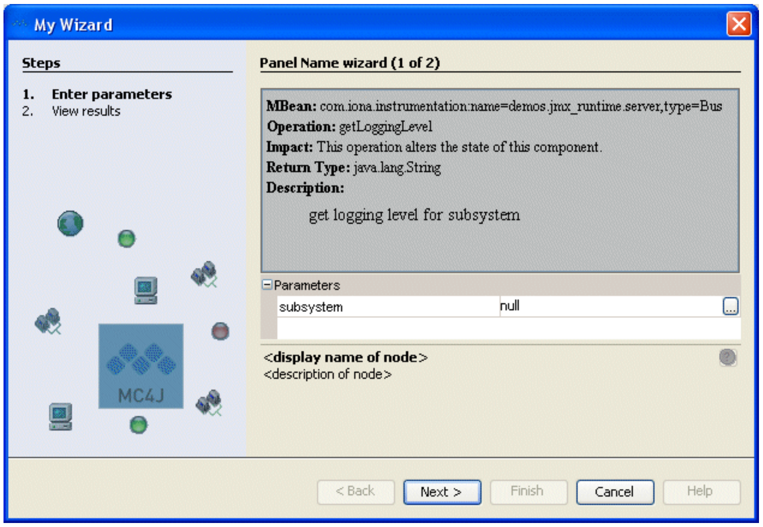
1. Expand the following tree node in the left panel of MC4J:

   ```
   name=demos.jmx_runtime.server,type=Bus
   ```

2. Expand the Operations node.

3. Double click `getLoggingLevel` to display the **My Wizard** screen.

Figure 21: *Using My Wizard to View Logging Levels*

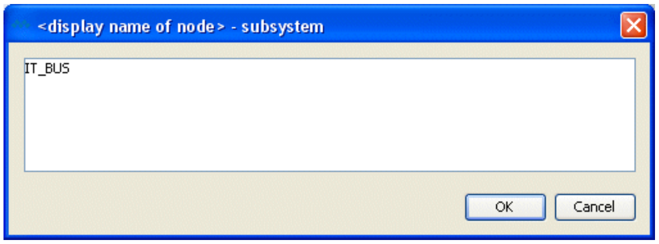


MBean: com.iona.instrumentation:name=demos.jmx_runtime.server,type=Bus
Operation: getLoggingLevel
Impact: This operation alters the state of this component.
Return Type: java.lang.String
Description:
get logging level for subsystem

Parameters	
subsystem	null

4. In the Parameters section, click the **...** button next to **subsystem**.
5. In the <display name of node> - subsystem dialog box, type **`IT_BUS`** and click **OK**.

Figure 22: *Providing a Subsystem Name*

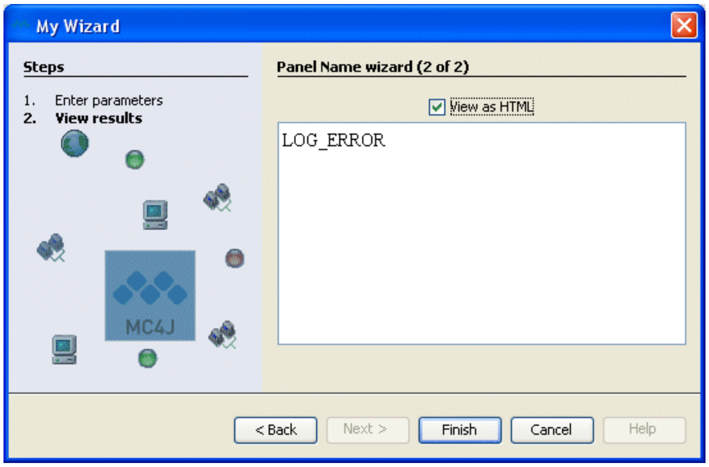


IT_BUS

6. Back in the **My Wizard** screen click **Next**.

7. The logging level for the `IT_BUS` subsystem is returned as `LOG_ERROR`.

Figure 23: *My Wizard Displays Logging Level*



LOG_ERROR

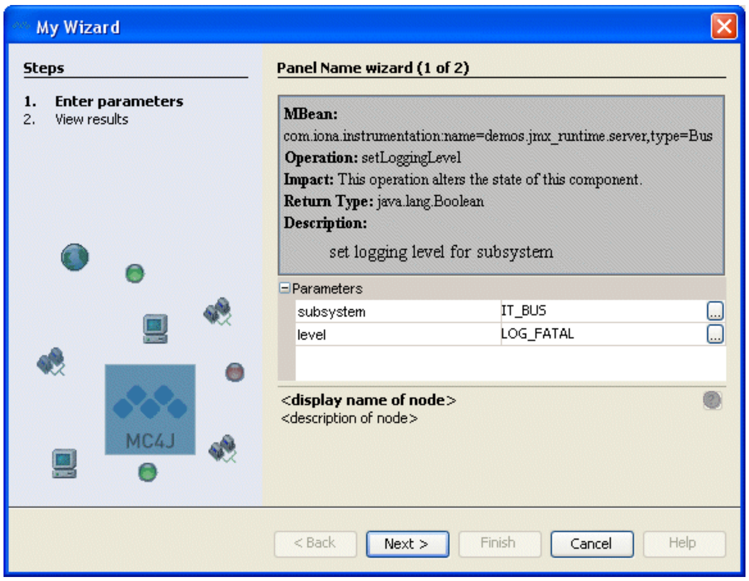
8. Now, use the **My Wizard** screen to enter a logging subsystem of `IT_BUS.CORE`.
9. Click **Next.** The logging level for `IT_BUS.CORE` is displayed as `LOG_INFO_LOW`. The logging level for `IT_BUS.CORE` has been configured differently from its `IT_BUS` parent (see "Defined demo logging configuration" on page 74).

Setting the logging level for a subsystem

To set the logging level for a specified logging subsystem:

1. Double click the `setLoggingLevel` node in the left panel of the MC4J console. This displays the **My Wizard** screen, as show in Figure 24.
2. Enter `IT_BUS` for the `subsystem`, and `LOG_FATAL` for the logging `level`, as show in Figure 24.

Figure 24: *Setting a Logging Level*



MBean:
com.iona.instrumentation:name=demos.jmx_runtime.server,type=Bus
Operation: setLoggingLevel
Impact: This operation alters the state of this component.
Return Type: java.lang.Boolean
Description:
set logging level for subsystem

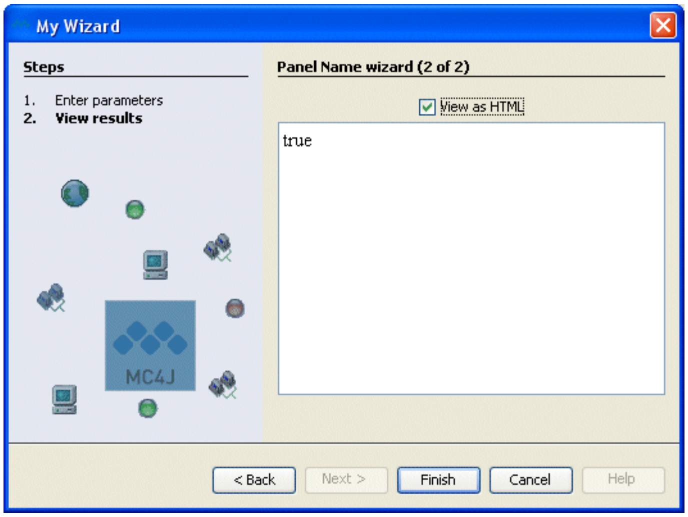
Parameters

subsystem	IT_BUS
level	LOG_FATAL

<display name of node>
<description of node>

3. Click **Next**. This displays `true`, as shown in Figure 25, which means that the logging level has been set successfully.

Figure 25: *Logging Level Set Successfully*



```
true
```

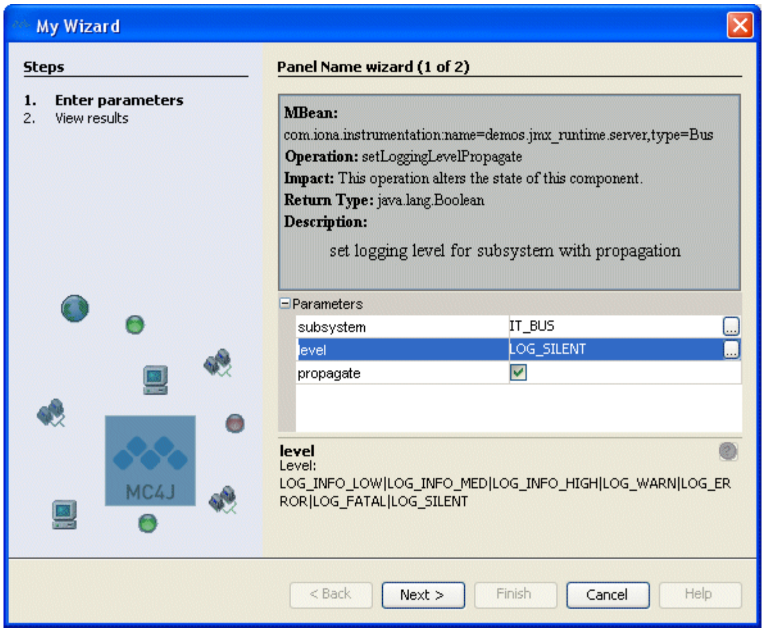
4. View the logging level of the `IT_BUS` subsystem to verify your setting (as described in "Viewing logging levels for a subsystem" on page 74). The logging level for `IT_BUS` is now `LOG_FATAL`.
5. View the logging level for the `IT_BUS.INITIAL_REFERENCE` subsystem. The logging level for `IT_BUS.INITIAL_REFERENCE` is also `LOG_FATAL`.
6. View the logging level for `IT_BUS.CORE`. The logging level of `IT_BUS.CORE` is still `LOG_INFO_LOW`. It does not inherit the `LOG_FATAL` level from its parent because its logging level has been configured separately (see "Defined demo logging configuration" on page 74).

Setting a logging level for a subsystem with propagation

To set a logging level to override a child subsystem with a separately configured logging level:

1. Double-click the `setLoggingLevelPropagate` tree node in the left panel of MC4J. This displays the **My Wizard** screen.
2. Enter `IT_BUS` as the subsystem, and `LOG_SILENT` as the logging level.

Figure 26: *Propagating a Logging Level*



MBean: com.iona.instrumentation:name=demos.jmx_runtime.server,type=Bus
Operation: setLoggingLevelPropagate
Impact: This operation alters the state of this component.
Return Type: java.lang.Boolean
Description:
set logging level for subsystem with propagation

Parameters

subsystem	IT_BUS
level	LOG_SILENT
propagate	✓

level
Level:
LOG_INFO_LOW|LOG_INFO_MED|LOG_INFO_HIGH|LOG_WARN|LOG_ERROR|LOG_FATAL|LOG_SILENT

3. Click **Next**. The returned value is `true`, which means that the logging level is set successfully.
4. View the logging level for `IT_BUS` (as described in "Viewing logging levels for a subsystem" on page 74). The logging level for `IT_BUS` is `LOG_SILENT`.

5. View the logging level for IT_BUS.INITIAL_REFERENCE. The logging level for IT_BUS.INITIAL_REFERENCE is also LOG_SILENT.
6. View the logging level for IT_BUS.CORE. The logging level for IT_BUS.CORE is also LOG_SILENT. Specifying propagation overrides log levels for all child logging subsystems.

Further information

For detailed information on Artix logging, see Configuring and Deploying Artix Solutions, C++ Runtime.

For more information on using MC4J, see:

http://mc4j.org/confluence/display/MC4J/Home

CHAPTER 5

Managing WS-RM Persistence with JMX

You can manage Web Services Reliable Messaging persistence in Artix using any JMX console.

In this chapter

This chapter discusses the following topics:

WS-RM Persistence Management

Overview

You can use any JMX console to view messages in the WS-RM persistence database both locally and remotely. You also can monitor the WS-RM persistence enabled endpoint, the WS-RM acksTo endpoint URI, and the client's RM source endpoint. This section explains the WS-RM persistence information that can be managed in a JMX console.

Managed WS-RM persistence components

The following WS-RM persistence components can be managed in a JMX console:

- Managed WS-RM persistence endpoints (`RMEndpointPersistentStore`)
- Managed WS-RM persistence sequences (`RMSequencePersistentStore`)

Managed WS-RM persistence endpoints

WS-RM persistence endpoint managed components are used to represent WS-RM persistence enabled endpoints. When a WS-RM persistence destination endpoint is created, it is registered as an MBean. When an WS-RM persistence destination endpoint is closed, the MBean is unregistered from the MBeanServer.

The MBean naming convention is as follows:

```
com.iona.instrumentation:type=Bus.Service.Port.EndpointPersistent,
name=WSRM_ENDPOINT_PERSISTENCE,
Bus.Service.Port=portName,
Bus.Service="{namespace}localname",
Bus=busIdentifier
```

WS-RM persistence endpoint attributes

You can view the following attributes for a WS-RM persistence endpoint in a JMX console:

Name	Description	Type
service name	WS-RM persistence enabled service name	`String`
port name	WS-RM persistence enabled port	`String`

Managed WS-RM persistence sequences

WS-RM persistence sequence managed components are used to represent WS-RM sequences. A destination sequence with a unique ID is created for each client. When a WS-RM persistence destination sequence is created, it is registered as an MBean. When a WS-RM persistence destination sequence is recovered from database, it is also registered as an MBean. When a WS-RM persistence destination sequence is terminated, it is unregistered from the MBeanServer.

The MBean naming convention is as follows:

```
com.iona.instrumentation:type=Bus.Service.Port.EndpointPersistent.SequencePersistent,
name=sequenceName,
Bus.Service.Port.EndpointPersistent=WSRM_ENDPOINT_PERSISTENCE,
Bus.Service.Port=portName,
Bus.Service="{namespace}localName",
Bus=busIdentifier
```

In this syntax, *sequenceName* includes the string `sequence_id` and the sequence ID.

WS-RM persistence sequence attributes

You can view the following attributes for a WS-RM persistence sequence in a JMX console:

Name	Description	Type
acksto uri	WS-RM acknowledgement URI	`String`
messages	Messages in the WS-RM persistence database	`String[]`
sequence id	Sequence unique ID representing a client	`String`

The messages attribute is a list of messages in the WS-RM persistence database. The messages are returned as a list of strings, where each string has the following format:

```
[message id]: messageId [message]: soapMessage
```

Viewing Messages in the WS-RM Persistence Database

Overview

Before you start viewing in the WS-RM persistence database, you must set your Artix configuration to enable JMX management for WS-RM persistence. This section uses the Artix WS-RM sample application to explain how to view and monitor messages in the WS-RM persistence database.

Enable JMX management for WS-RM persistence

To enable JMX management for WS-RM persistence in your Artix configuration file, perform the following steps:

1. Open the following file:

```
ArtixInstallDir\cxx_java\samples\advanced\wsrm\etc\wsrm.cfg
```

2. Edit the `demos.wsrm_persistence_enabled.server` scope as follows:

```
server {
    plugins:artix:db:home = "./server_db";
    plugins:bus_management:enabled="true";
    plugins:bus_management:connector:enabled="true";
    plugins:bus_management:connector:url:file="../../etc/connector.url";

   # optional port, default is 1099
   plugins:bus_management:connector:port="5008";
};
```

Note: Enabling JMX management for WS-RM persistence is similar to enabling JMX management for other Artix components.

Start the server

3. To start the server, go to the following directory:

```
InstallDir\cxx_java\samples\advanced\wsrm\bin\
```

4. Run the following command:

   `run_cxx_server_persistence.bat`

   This starts the server using following example command:

```
start server.exe -ORBname demos.wsrm_persistence_enabled.server
```

When the server runs, a file named `connector.url` is created in the `...samples\advanced\wsrm\etc\` directory.

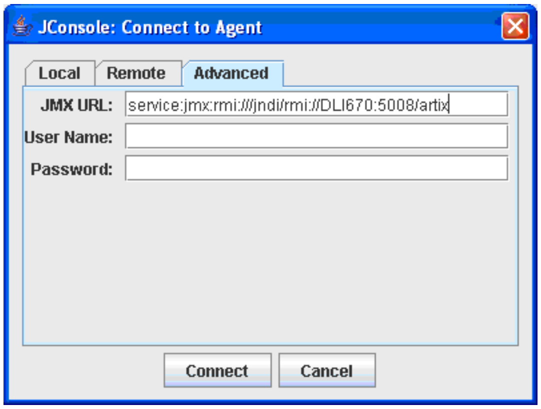
Start a JMX console

You can start any JMX console. For example, to start JConsole, execute the following command:

```
%jdk1.5_home%\bin\jconsole.exe
```

This displays the **JConsole: Connect to Agent** dialog, as shown in Figure 27.

Figure 27: *Connecting to a JMX Agent*



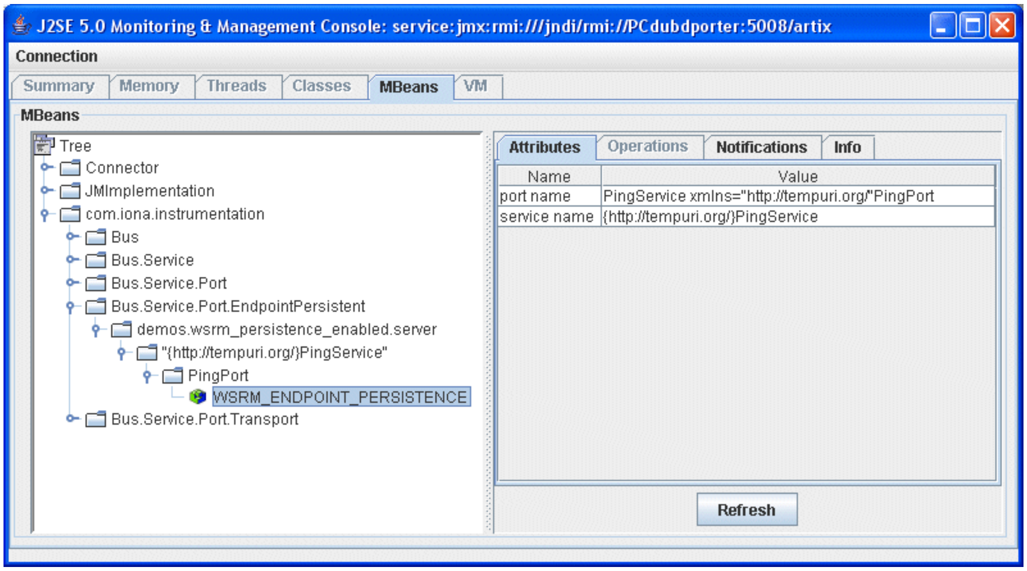
JMX URL: service:jmx:rmi:///jndi/rmi://DLI670:5008/artix
User Name:
Password:

Copy the contents of the `connector.url` file into the JMX URL field, and click **Connect**. This displays the **J2SE 5.0 Monitoring and Management Console**, as shown in Figure 28.

View WS-RM persistence enabled endpoints

You can view a WS-RM persistence enabled endpoint in the **MBeans** tab of the JMX console, as shown in Figure 28:

Figure 28: *WS-RM Persistence Enabled Endpoint*


J2SE 5.0 Monitoring & Management Console: service:jmx:rmi:///jndi/rmi://PCdubdporter:5008/artix
Connection
Summary
Memory
Threads
Classes
MBeans
VM
MBeans
Tree
Connector
JMImplementation
com.iona.instrumentation
Bus
Bus.Service
Bus.Service.Port
Bus.Service.Port.EndpointPersistent
demos.wsrm_persistence_enabled.server
"{http://tempuri.org/}PingService"
PingPort
WSRM_ENDPOINT_PERSISTENCE
Bus.Service.Port.Transport
Attributes
Operations
Notifications
Info
Refresh

Name	Value
port name	PingService xmlns="http://tempuri.org/"PingPort
service name	{http://tempuri.org/}PingService

In this example, PingPort is a WS-RM persistence enabled port. You can view the port and service name in the **Attributes** tab on the right of the console.

View messages in the WS-RM persistence database

To view messages in the WS-RM persistence database, perform the following steps:

1. Edit the client code in `...\samples\advanced\wsrm\cxx\client\PingClientSample.cxx` as follows:

```
int
  run_persistence_client(
    int argc,
    char* argv[]
)
...
  for (int i=0; i < 10; i++)
      {
       cout << "Invoking PingOneway " << i << endl;
       PingType param1;
       param1.setText("PingOneway message from client");
       client1.PingOneway(param1);
       cout << i << " PingOneway invoked" << endl;
      }
...
```

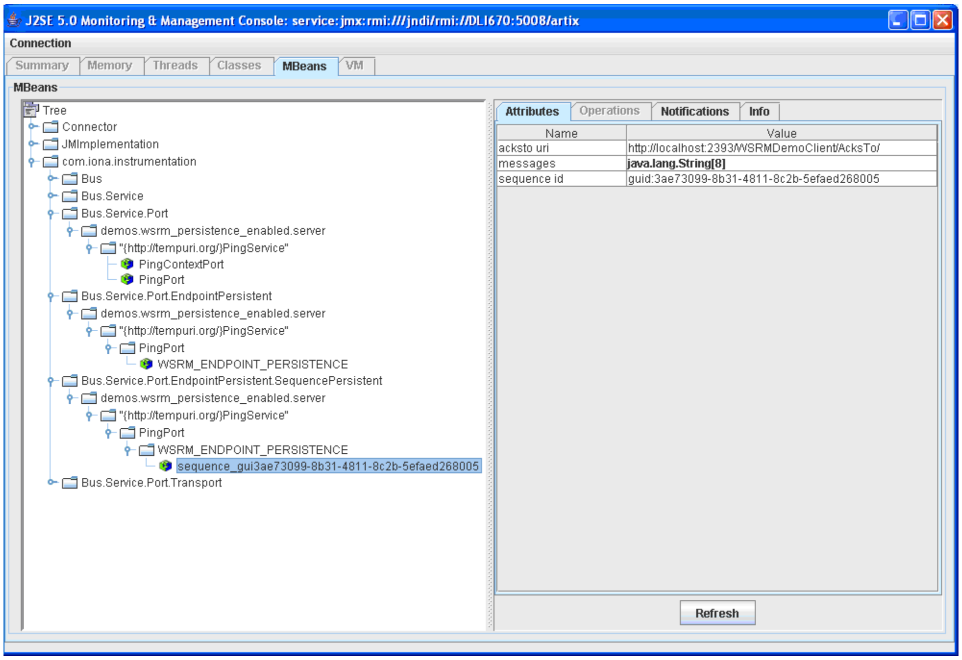
   This adds a loop to the client that invokes the server 10 times in order to easily view messages in WS-RM persistence database.

2. Start the client. For example, go to the `...\samples\advanced\wsrm\bin` directory, and run the following command:

```
run_cxx_client_persistence.bat
```

3. You can view the attributes for the WS-RM sequence in the JMX console, as shown in Figure 26. The WS-RM sequence name consists of the `sequence_guid` string and a sequence ID.

Figure 29: *WS-RM Sequence Attributes*

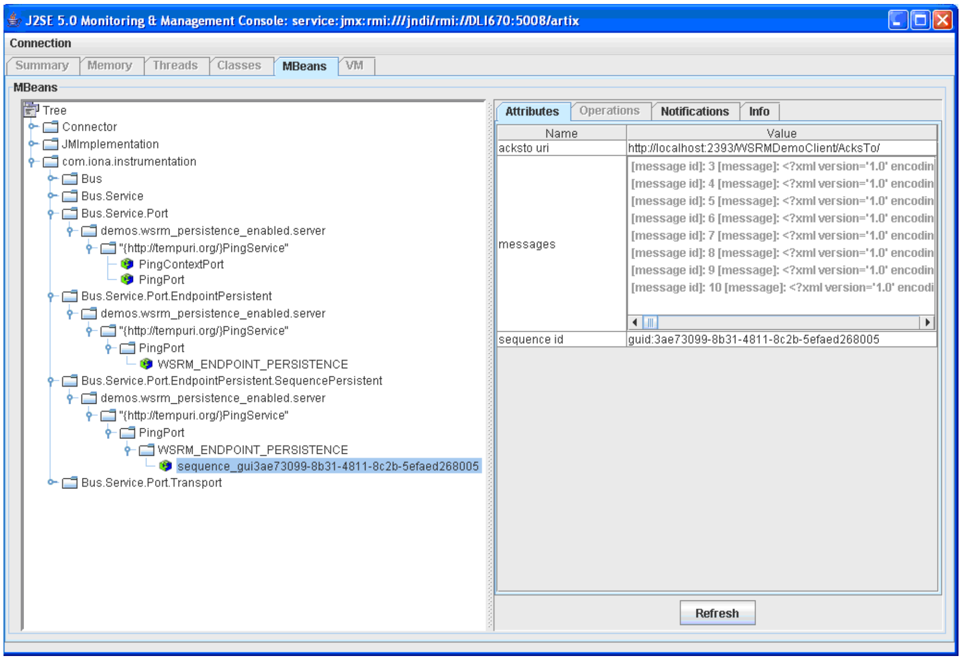


```
Tree
  Connector
  JMImplementation
  com.iona.instrumentation
    Bus
    Bus.Service
    Bus.Service.Port
      demos.wsrm_persistence_enabled.server
        "{http://tempuri.org/}PingService"
          PingContextPort
          PingPort
    Bus.Service.Port.EndpointPersistent
      demos.wsrm_persistence_enabled.server
        "{http://tempuri.org/}PingService"
          PingPort
            WSRM_ENDPOINT_PERSISTENCE
    Bus.Service.Port.EndpointPersistent.SequencePersistent
      demos.wsrm_persistence_enabled.server
        "{http://tempuri.org/}PingService"
          PingPort
            WSRM_ENDPOINT_PERSISTENCE
              sequence_gui3ae73099-8b31-4811-8c2b-5efaed268005
    Bus.Service.Port.Transport
```

Name	Value
acksto uri	http://localhost:2393/WSRMDemoClient/AcksTo/
messages	**java.lang.String[8]**
sequence id	guid:3ae73099-8b31-4811-8c2b-5efaed268005

4. You can view all the messages in WS-RM persistence database by clicking in the **Attributes** tab on the right of the console, as shown in Figure 30. Each message consists of a message ID and a SOAP message.

Figure 30: *Messages in the WS-RM Persistence database*



```
Tree
  Connector
  JMImplementation
  com.iona.instrumentation
    Bus
    Bus.Service
    Bus.Service.Port
      demos.wsrm_persistence_enabled.server
        "{http://tempuri.org/}PingService"
          PingContextPort
          PingPort
    Bus.Service.Port.EndpointPersistent
      demos.wsrm_persistence_enabled.server
        "{http://tempuri.org/}PingService"
          PingPort
            WSRM_ENDPOINT_PERSISTENCE
    Bus.Service.Port.EndpointPersistent.SequencePersistent
      demos.wsrm_persistence_enabled.server
        "{http://tempuri.org/}PingService"
          PingPort
            WSRM_ENDPOINT_PERSISTENCE
              sequence_gui3ae73099-8b31-4811-8c2b-5efaed268005
    Bus.Service.Port.Transport
```

Name	Value
acksto uri	http://localhost:2393/WSRMDemoClient/AcksTo/
messages	[message id]: 3 [message]: <?xml version='1.0' encodin [message id]: 4 [message]: <?xml version='1.0' encodin [message id]: 5 [message]: <?xml version='1.0' encodin [message id]: 6 [message]: <?xml version='1.0' encodin [message id]: 7 [message]: <?xml version='1.0' encodin [message id]: 8 [message]: <?xml version='1.0' encodin [message id]: 9 [message]: <?xml version='1.0' encodin [message id]: 10 [message]: <?xml version='1.0' encodi
sequence id	guid:3ae73099-8b31-4811-8c2b-5efaed268005

You can click the **Refresh** button to view the current messages in WS-RM persistence database.

Part III

Progress Actional

In this part

This part contains the following chapters:

CHAPTER 6

Artix–Actional Integration

Artix provides support for integration with Progress Actional SOA management products.

In this chapter

This chapter includes the following section:

Artix–Actional Interaction Architecture

Overview

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

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

This guide explains how to integrate Artix applications written in Java (JAX-RPC). For details on integrating JAX-WS applications, see the Artix Management Guide, Java Runtime.

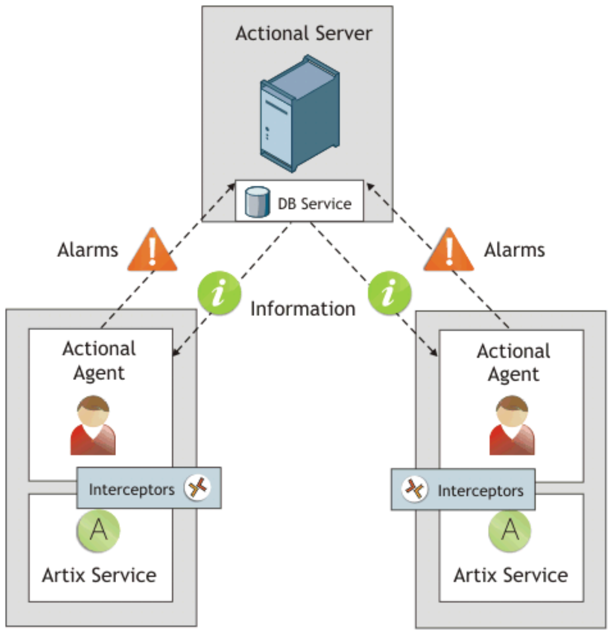
Artix–Actional integration architecture

The Actional SOA management system includes an Actional server and an Actional agent. The Actional agent is run on each node that you wish to manage. A node is defined as a system on the current network. A node with an Actional agent installed is referred to as an *instrumented node* or a *managed node*.

The managed node uses Actional's interceptor API to send monitoring data to the Actional agent. The Actional server pings the Actional agent periodically to retrieve the monitoring data. It analyzes this data and represents it in the Actional SOA management GUI tools. In addition, any alerts triggered at the Actional agent are sent immediately to the Actional server.

Figure 31 shows how Artix Web service applications are integrated with Actional using this architecture.

Figure 31: *Artix–Actional Integration Architecture*



The main components in this architecture are:

- "Actional server"
- "Actional agent"
- "Artix interceptors"
- "Actional agent interceptor API"
- "Artix service endpoints"
- "Service consumers"

Actional server

The Actional server is a central management server that manages nodes containing an Actional agent.

The Actional server hosts a database and pings Actional agents to obtain management data at configured time intervals. It analyzes the management data and displays it in an Actional console—for example, the **Actional Server Administration Console**. This is a Web application deployed on Apache Tomcat, runtime management and agent configuration modes.

By default, the Actional server uses port `4040`. The default Actional server database is Apache Derby.

Actional agent

An Actional agent is run on each Artix node that you wish to manage. Actional agents are used to provide instrumentation data back to the Actional server.

Actional agents are provisioned from the Actional server to establish initial contact and send configuration to the Actional agent. There is one Actional agent per managed node. By default, the Actional agent uses port `4041`.

Artix interceptors

At the level of a managed node, Artix interceptors send the instrumentation data to the Actional agent using an Actional-specific API. These interceptors essentially push events to the Actional agent.

The data is analyzed and stored in the Actional agent for retrieval later by the Actional server. However, any alerts triggered at the Actional agent are sent immediately to the Actional server.

Artix Java handlers

In Artix Java, interceptors are also known as *Java handlers*. For example, at the implementation level, Java handlers are used as follows:

1. Artix initializes a Java plug-in that loads a Java handler factory.
2. The handler factory creates client-side request and message handlers, and server-side request and message handlers.

3. When the Artix client-side request handler is invoked, Artix initiates an client Actional interaction object, and sets the following data on this object:
   - Service name
   - Port name
   - Operation name
   - Endpoint URL
   - IP address
   - Correlation ID
4. When client message handler is invoked, Artix gets the message payload and sets it in the client Actional interaction object.
5. On the Artix server side, when the request reaches the server-side message handler, Artix starts a server Actional interaction object and sets the message payload.
6. When the server-side request handler is invoked, Artix sets the same data listed in step 3 on the server Actional interaction object.

Actional agent interceptor API

The Actional Agent Interceptor SDK is an Actional-specific API used to send the management instrumentation data from the service endpoint to the Actional agent.

The Artix service application to be managed by Actional must use the Actional Agent Interceptor SDK to send monitoring data to the Actional agent. For detailed information on how to use this API, see the Actional product documentation.

Artix service endpoints

An Artix service endpoint is a service built using Artix, and described using WSDL. The endpoint can be implemented in Java (JAX-RPC). However, the main characteristic of an Artix service endpoint is that it can be described in WSDL, and classified as a service, which can be consumed.

Service consumers

Service consumers are clients that consume service endpoints by exchanging messages based on the service interface. Consumers can be built using Artix, or any product that supports the technology used by the endpoint. For example, a pure CORBA client could be a consumer for a CORBA endpoint. A .NET client could be a consumer for an Artix SOAP endpoint.

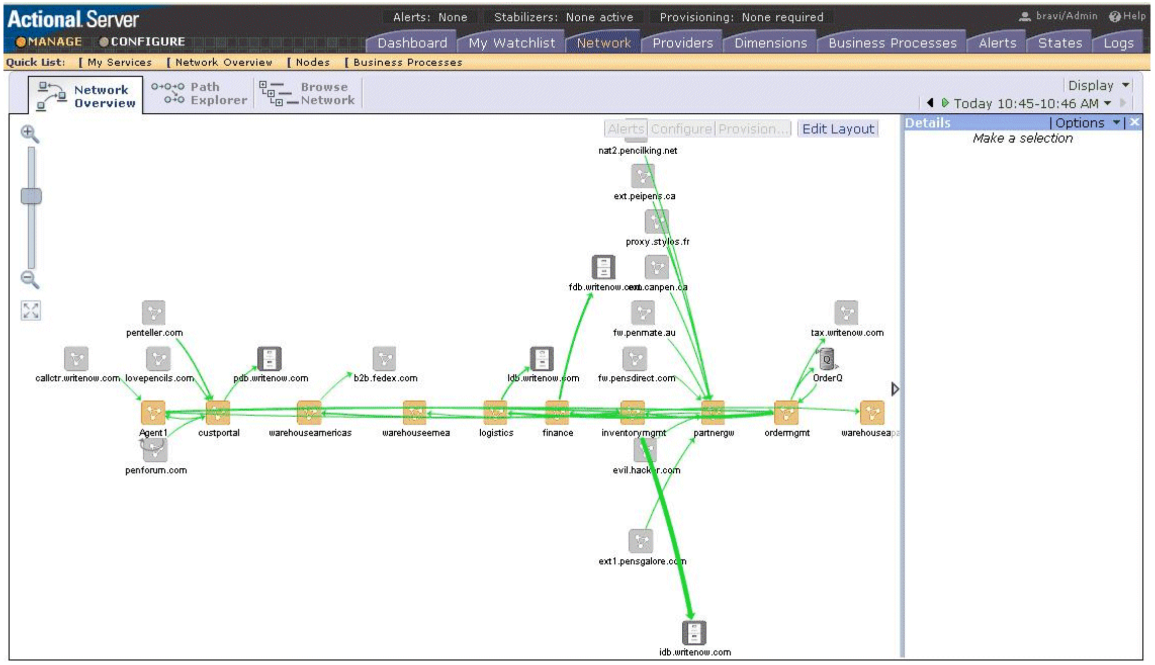
Actional SOA management system

In this document, Actional is the general term used to describe the Actional SOA management system in which all data is stored and viewed. This simplifies the architecture of Actional for the sake of this discussion.

Figure 32 shows an example of the **Actional Server Administration Console**. Managed nodes are displayed as orange boxes, and unmanaged nodes are displayed as grey boxes. The green arrow indicates the message flow through various nodes.

Clicking on each of the nodes shows more in-depth information regarding the response time, alerts and warnings, and so on. The organization of the information in this web console is in the form of *Node–Group–Service–Operation*. In Artix, this translates to *Node–Service–Port–Operation*.

Figure 32: *Actional Server Administration Console*



Further information

For detailed information on using Actional features, see the Actional product documentation.

For more information on Artix Java request/response handlers, see Developing Artix Applications with JAX-RPC.

Configuring Artix–Actional Integration

This chapter explains how to configure integration between Artix and Actional SOA management products, and shows examples from Artix-Actional integration demos.

In this chapter

This chapter includes the following sections:

Prerequisites

Overview

This section describes prerequisites for integration between Artix and Actional SOA management products.

The Actional for SOA Operations product is aimed at a technical audience (for example, system administrators managing services on the network). While the Actional for Continuous Service Optimization (Actional CSO) product is aimed at a business audience.

Supported product versions

Artix supports integration with the following Actional product versions:

- Actional for SOA Operations 7.1 and 7.2.
- Actional for Continuous Service Optimization 7.1 and 7.2.

Supported protocols and transports

The following protocols and transports are supported:

- SOAP over HTTP
- SOAP over JMS

Actional agents

The Actional agent component is also known as the Actional Point of Operational Visibility.

You must ensure that Actional agents have been set up on each Artix node that you wish to manage. The provisioning of Actional agents is performed using the Actional server. For some basic details, see “Configuring Actional for Artix Integration” on page 103.

For information on how to set up Actional agents on managed nodes, see the Actional product documentation.

Further information

For information on the full range of platform versions and database versions supported by Actional, see the Actional product documentation.

This Artix integration with Actional supports the full range of operating system platforms supported by Artix. For more details, see the Artix Installation Guide.

Configuring Actional for Artix Integration

Overview

These section provides some basic configuration guidelines for Actional agent and server configuration. For full details, see the Actional product documentation.

This basic configuration will help to set up the Artix–Actional integration demos. For information on how to run these demos, see the `readme.txt` files in the following directories:

```
ArtixInstallDir/cxx_java/samples/advanced/management/monitoring/actional_http_handler
ArtixInstallDir/cxx_java/samples/advanced/management/monitoring/actional_jms_handler
```

Actional agent configuration

No specific Actional agent configuration settings are required for integration with Artix. For example, for the purposes of the Actional-Artix integration demos, the Actional agent can be started with the default configuration settings.

Actional server configuration

The following sample configuration steps describe how to set up the Actional server to run an simple Artix-Actional demo:

1. Install the Actional server with typical installation options, and select the Apache Derby database.
2. Specify the following URL in your browser:

   `http://localhost:4040/lgserver`
3. If this is a new installation click **Start**, and follow new the Actional server setup steps.

   Otherwise, if the Actional server is already installed, perform the following steps:

   i. In the Actional console Web interface, select the **Configure** radio button in the top left of the screen.

   ii. Select **Platform** tab. This displays the general configuration settings.

Creating a managed node

To create a managed node for a simple Artix demo, perform the following steps:

1. In the Actional **Configure** view menu bar, open the **Network** tab. This displays the **Network Nodes**.
2. Select **Add**. This displays **Node Creation / Managing Agents**.
3. Click **Managed Node**.

Configuring a new node

To configure a managed node for the demo, perform the following steps in the wizard:

Step 1: New Node - Identification

1. Specify the **Name** as `agent1`.
2. Specify the **Display icon** as `auto-discover` (you can select `IONA Artix` from the drop down list, if desired).
3. Click **Next**.

Step 2: New Node - Management

1. Specify the **Transport** as `HTTP/S`.
2. Supply the Actional agent user name and password.
3. Ensure that **Override Agent Database** is checked.
4. Click **Next**.

Step 3: New Node - Agents

1. Specify the following URL:

   ```
   http://HostName:4041/lgagent
   ```

   You can specify a host name or an `IP_ADDRESS`.
2. Click **Add**. The agent URL is added.
3. Click **Next**.

Step 4: New Node - Endpoints

1. For **Endpoints**, add the hostname, fully qualified hostname, and IP address.
2. Click **Next**.

Step 5: New Node - Filters

1. Do not specify any filters for the demo.
2. Click **Next**.

Step 6: New Node - Trust Zone

1. Do not specify a trust zone the demo.
2. Click **Finish**

The node is created, and needs to be provisioned.

Provisioning a new node

To provision the new node, perform the following steps:

1. Select the **Deployment** tab from the **Configure** menu bar.
2. The **Provisioning** page is displayed, and `agent1` is listed as not provisioned.
3. Select the `agent1` check box.
4. Click **Provision**. This displays a message when complete: `Successfully provisioned`.
5. Click the **Manage** radio button on the Actional Web interface. You should see `agent1` added to the **Network Overview** screen.

Configuring Artix Java Services for Actional Integration

Overview

This section explains how to configure Artix Java (JAX-RPC) services for integration with Actional. It shows some examples from the Artix–Actional integration demos:

```
ArtixInstallDir/cxx_java/samples/advanced/management/monitoring/actional_http_handler
ArtixInstallDir/cxx_java/samples/advanced/management/monitoring/actional_jms_handler
```

Configuring the Artix monitoring plug-in

Configuring the Artix monitoring plug-in includes the following steps:

- Specifying the plug-in name
- Adding the Java handlers to the interceptor chain
- Configuring the monitoring tool

You can configure the monitoring plug-in by editing your Artix configuration (`artix.cfg`) file.

Specifying the plug-in name

To set the monitoring plug-in factory class, and load the plug-in name, add the following settings:

```
# Configure the plug-in factory class:
plugins:monitoring_plugin:classname =
    "com.iona.jbus.management.monitoring.interceptors.MonitoringPlugInFactory";

# Load the java plug-in:
orb_plugins = ["soap", "java"];

# Load the monitoring plug-in:
java_plugins = ["monitoring_plugin"];
```

Adding the monitoring handlers to the interceptor chain

You must specify monitoring handlers to the request-level and message-level interceptor lists, on both the client side and server side:

```
# Add the client-side handlers to the interceptors chain.
binding:artix:client_request_interceptor_list= "monitoring_handler";
binding:artix:client_message_interceptor_list= "monitoring_handler";

# Add the server-side handlers to the interceptors chain.
binding:artix:server_request_interceptor_list= "monitoring_handler";
binding:artix:server_message_interceptor_list= "monitoring_handler";
```

For more details on configuring binding lists and interceptors, see Artix Configuration Reference.

Configuring the monitoring tool

You must also configure the name of the reporting tool (in this case, `actional`). `actional` is currently the only supported value. For example:

```
plugins:monitoring_plugin:know_report_tool= "actional";
```

Optimizing your Actional integration

Artix provides the following configuration options to enable you to fine-tune the behavior of the monitoring plug-in.

Reporting the message payload

You can enable reporting of the message payload on the server side (for example, a SOAP message over HTTP). If this option is set to `false`, only the payload size is reported. The default value is:

```
plugins:monitoring_plugin:enable_si_payload = "true";
```

Specifying the maximum size of the payload

You can specify the maximum size in bytes of the message payload to report. If a message payload exceeds this value, only its size is reported, regardless of the value of the `enable_si_payload` option. An example setting is:

```
plugins:monitoring_plugin:max_reported_payload_size= "1024";
```

The default value is `-1` (unlimited).

Enabling a service facade

The service facade feature enables reporting of all interactions with an extra representation of the target service on the client side. This is also known informally as an *extra hop*. This is useful when it is impossible to report what service is being invoked by the client (for example, where a JMS queue exists in the invocation chain). The default value is:

```
plugins:monitoring_plugin:show_service_facade= "false";
```

Sample configuration

The following sample configuration shows some example settings in a `my_app` configuration scope:

```
my_app {

  monitoring_jms_handler {

    plugins:monitoring_plugin:classname =
    "com.iona.jbus.management.monitoring.interceptors.MonitoringPlugInFactory";

    orb_plugins = ["soap", "java"];
    java_plugins = ["monitoring_plugin"];

    # Name of the report tool
    plugins:monitoring_plugin:know_report_tool= "actional";

    # Enable the report of the payload (default = "true")
    plugins:monitoring_plugin:enable_si_payload = "true";

    # Maximum size of the reported payload (default is -1 unlimited)
    plugins:monitoring_plugin:max_reported_payload_size= "-1";

    client {
       binding:artix:client_request_interceptor_list= "monitoring_handler";
       binding:artix:client_message_interceptor_list= "monitoring_handler";

       # Enable service Facade representation
       plugins:monitoring_plugin:show_service_facade= "true";
     };

    server {
       binding:artix:server_request_interceptor_list= "monitoring_handler";
       binding:artix:server_message_interceptor_list= "monitoring_handler";
     };
};
```

Viewing Artix Endpoints in Actional

Overview

When your Artix service endpoints and consumers have been configured for integration with Actional, they can be monitored using the Actional SOA management tools.

For example, when you run the Artix–Actional SOAP over JMS demo, the **Actional Server Administration Console** displays the server queues and agent nodes. Invocations are displayed as arrows flowing to and from the queues. For details on how to run this demo, see the `readme.txt` file in the following directory:

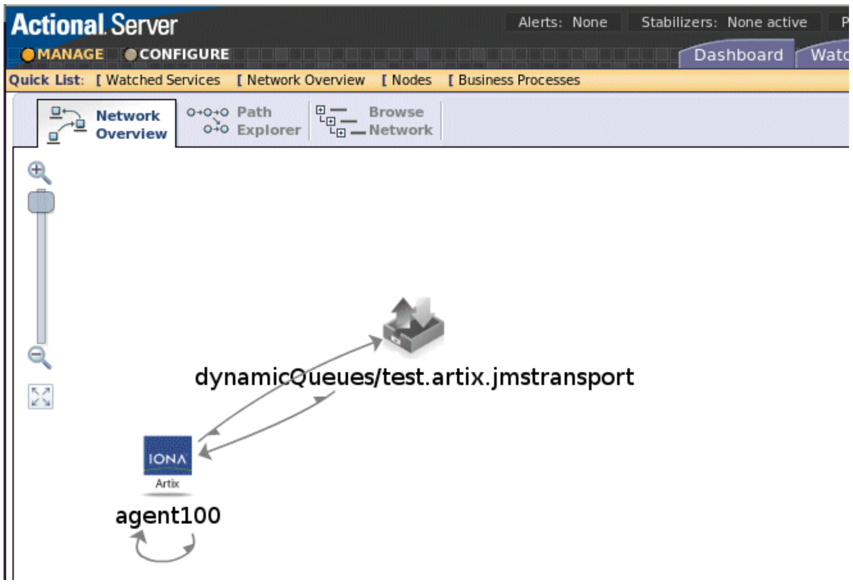
```
ArtixInstallDir/cxx_java/samples/advanced/management/monitoring/actional_jms_handler
```

Network overview

Figure 33 shows a running SOAP over JMS demo displayed in the **Network Overview** screen of the **Actional Server Administration Console**.

In Figure 33, the JMS queue is displayed on top, and the instrumented Artix application is displayed below. The interactions between the client and server applications are recorded by `agent100`, which is installed on the machine that runs the demo. This agent reports monitoring data back to the Actional server.

Figure 33: *Actional Server Network Overview*



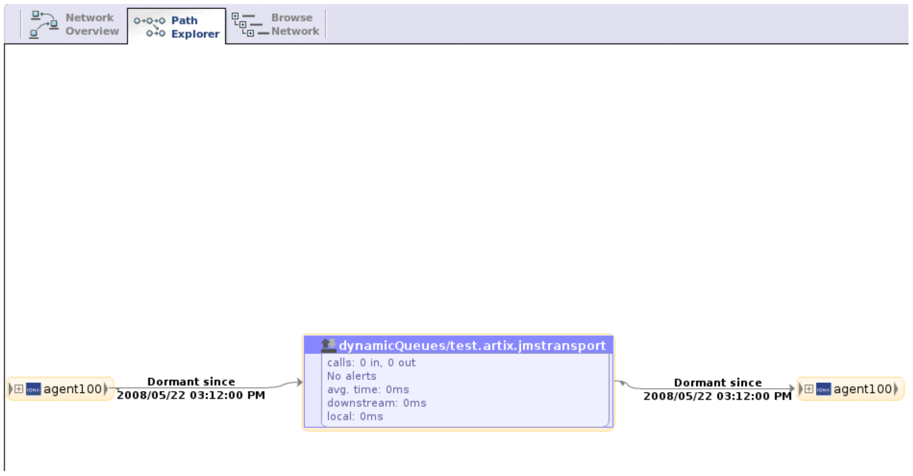
The arrows between the agent and the JMS queue represent the invocations: out to the queue from the client application, and back from the queue showing the message received by the service from the queue.

The arrow that loops from `agent100` back to itself is the extra hop, or service facade call. For more details, see "Enabling a service facade" on page 108.

Path Explorer

Figure 34 shows the example JMS queue displayed in the **Path Explorer** screen of the **Actional Server Administration Console**.

Figure 34: *Actional Server Path Explorer*

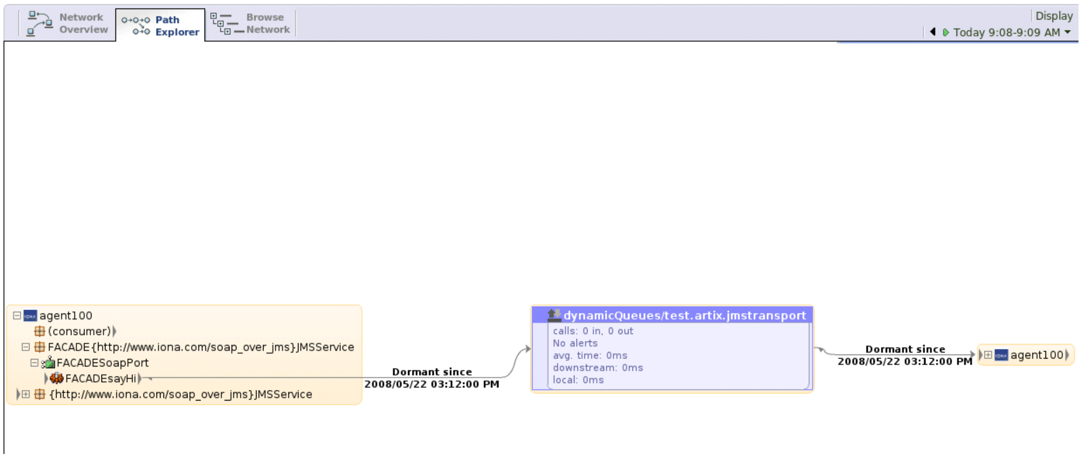


To view this screen, select the JMS queue object in Figure 33, and select the **Path Explorer** tab at the top left. This example shows the invocations from the point of view of the JMS queue.

Viewing a service facade

Figure 35 shows what is displayed when you expand `agent100` in Figure 34, and select the `FACADEsayHI` operation.

Figure 35: *Service Facade in Path Explorer*


Network Overview
Path Explorer
Browse Network
Display
Today 9:08-9:09 AM
agent100
(consumer)
FACADE{http://www.iona.com/soap_over_jms}JMSService
FACADESoapPort
FACADEsayHi
{http://www.iona.com/soap_over_jms}JMSService
Dormant since 2008/05/22 03:12:00 PM
dynamicQueues/test.artix.jmstransport
calls: 0 in, 0 out
No alerts
avg. time: 0ms
downstream: 0ms
local: 0ms
Dormant since 2008/05/22 03:12:00 PM
agent100

In Figure 35, the facade service appears to make a call to the JMS queue. However, the consumer is the actor that performs the invocation. Without the facade, you would only see an arrow from the consumer directly to the JMS queue. You would not know what service port or operation is invoked through the queue. This is because the queue acts as an opaque buffer for all messages.

Adding the service facade enables you to represent where the call is going. The small arrow head displayed to the right of `consumer` is the extra hop invocation from the consumer to the facade. The small arrow head displayed to the left of the `FACADEsayHi` operation is the other end of the invocation. The Actional console displays the call to the JMS queue as if it originates from the service facade.

Figure 36 shows the display when you expand the `agent100` object on the right of Figure 35. This represents the call from the JMS queue to the service being invoked. The names used for the facade in Figure 35 are constructed from the names for the service, endpoint and operation by prefixing them with `FACADE`.

Figure 36: *Service Facade in More Detail*

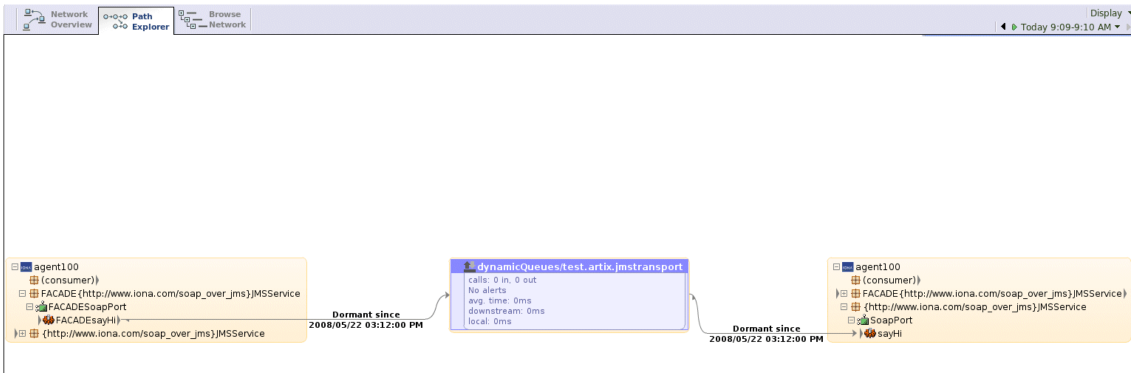
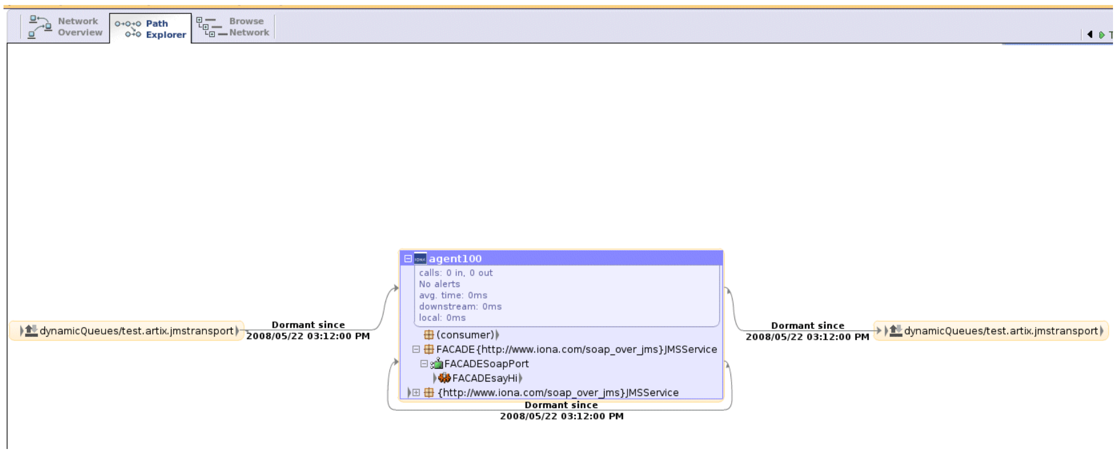

Network Overview
Path Explorer
Browse Network
Display
Today 9:09-9:10 AM
agent100
(consumer)
FACADE{http://www.iona.com/soap_over_jms}JMSService
FACADESoapPort
FACADEsayHi
{http://www.iona.com/soap_over_jms}JMSService
Dormant since 2008/05/22 03:12:00 PM
dynamicQueues/test.artix.jmstransport
calls: 0 in, 0 out
No alerts
avg. time: 0ms
downstream: 0ms
local: 0ms
Dormant since 2008/05/22 03:12:00 PM
agent100
(consumer)
FACADE{http://www.iona.com/soap_over_jms}JMSService
{http://www.iona.com/soap_over_jms}JMSService
SoapPort
sayHi

Figure 37 shows the interaction from the point of view of `agent100`. In this example, the agent has central position, and calls to and from the JMS queue are displayed. The arrow looping back to `agent100` is the internal extra hop interaction with the service facade.

Figure 37: *Service Facade Extra Hop*


Network Overview
Path Explorer
Browse Network
agent100
calls: 0 in, 0 out
No alerts
avg. time: 0ms
downstream: 0ms
local: 0ms
(consumer)
FACADE{http://www.iona.com/soap_over_jms}JMSService
FACADESoapPort
FACADEsayHi
{http://www.iona.com/soap_over_jms}JMSService
dynamicQueues/test.artix.jmstransport
Dormant since 2008/05/22 03:12:00 PM
Dormant since 2008/05/22 03:12:00 PM
Dormant since 2008/05/22 03:12:00 PM
dynamicQueues/test.artix.jmstransport

For more information on service facades, see "Enabling a service facade" on page 108.

Further information

Actional

For information on how to set up and run the Actional server, Actional agent, and Actional Server Administration Console, see the Actional product documentation.

Artix

For more information on Artix configuration, see the following:

- Configuring and Deploying Artix Solutions, C++ Runtime
- Artix Configuration Reference, C++ Runtime

CHAPTER 8

Artix–Actional Deployment Scenarios

This chapter gives general guidelines on deploying an Artix–Actional integration in example production environments.

In this chapter

This chapter includes the following sections:

Deployment with IBM WebSphere and J2EE Connector

Overview

This section gives basic guidelines for integrating Artix and Actional in an example deployment scenario that includes an IBM WebSphere 5.1 environment with an Artix J2EE Connector.

Artix J2EE Connector is a resource adaptor that enables J2EE applications and Artix Web services to talk to each other. It is also used to manage connections, transactions, and security.

For more details on WebSphere, see the IBM WebSphere product documentation. For more details on Artix J2EE Connector, see Artix for J2EE.

IBM WebSphere deployment

Follow these general guidelines when deploying IBM WebSphere:

- Before starting WebSphere, ensure that the Artix environment script (`artix_env`) has been sourced.
- Deploy the Artix J2EE Connector resource adapter archive (`artix.rar`) on WebSphere. Instructions for deploying on WebSphere are described in Artix for J2EE.
- Follow the instructions in the *Actional Interceptor Guide* to enable WebSphere instrumentation.
- Ensure that a copy of `actional-sdk.jar` is present in your `$WAS_HOME/classes` directory.

Artix J2EE Connector deployment

In this example deployment scenario, which includes an Artix J2EE Connector, you also need to update your Artix J2EE Connector classloader firewall configuration file (`artix_j2ee_ce.xml`).

- Add the following entries under the `ce:environment` element:

```
<ce:filter type="pattern" > com.actional. </ce:filter>
<ce:filter type="negative-pattern"> javax.xml.soap. </ce:filter>
<ce:filter type="negative-pattern"> com.iona.jbus.jms. </ce:filter>
<ce:filter type="negative-pattern"> com.iona.jbus.management. </ce:filter>
```

- Add the following entries under the `ce:loader` element (using the fully qualified path):

```
<ce:location> path/to/it_bus_management_monitoring.jar </ce:location>
<ce:location> $IT_PRODUCT_DIR/lib/sun/saaj/1.2.1/saaj-api.jar </ce:location>
```

Artix deployment

When deploying an Artix and Actional integration, you need to add some configuration entries to your Artix configuration file (`artix.cfg`). For example, you must configure the Artix monitoring plug-in; you can also set additional options such as payload reporting and service facade.

For full details, see "Configuring Artix Java Services for Actional Integration" on page 106.

Example Artix configuration scope

The following example shows a `j2ee` configuration scope from an Artix configuration file (`.cfg`):

```
j2ee {

   plugins:monitoring_plugin:classname="com.iona.jbus.management.monitoring.intercept
   ors.MonitoringPlugInFactory";

    plugins:monitoring_plugin:know_report_tool= "actional";

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

    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop", "iiop", "soap",
   "java"];

    java_plugins = ["monitoring_plugin"];

    binding:artix:client_request_interceptor_list= "monitoring_handler";
    binding:artix:client_message_interceptor_list= "monitoring_handler";
    binding:artix:server_request_interceptor_list= "monitoring_handler";
    binding:artix:server_message_interceptor_list= "monitoring_handler";

    plugins:monitoring_plugin:enable_si_payload="true";
    plugins:monitoring_plugin:max_reported_payload_size="-1";
    plugins:monitoring_plugin:show_service_facade="true";

    tx {

    orb_plugins = ["local_log_stream", "iiop_profile", "giop", "iiop",

    "ws_coordination_service", "soap", "ots", "java"];

     plugins:bus:default_tx_provider:plugin = "wsat_tx_provider";

        xa {
            poa:j2ee_rm:direct_persistent="true";
            poa:j2ee_rm:well_known_address:port="58502";
            initial_references:TransactionFactory:plugin = "ots_encina";
        };
    };
};
```

Native Deployment with IBM WebSphere

Overview

This section gives basic guidelines for integrating Artix and Actional in an example deployment scenario that includes a native IBM WebSphere 5.1 environment (without Artix J2EE Connector).

For more details on WebSphere, see the IBM WebSphere product documentation.

Native WebSphere deployment

Using Artix natively with WebSphere requires specific WebSphere configuration settings. For example, WebSphere classloaders do not use the system classpath. This means that you must specify classloader configuration to WebSphere without causing class clashes—WebSphere has some common components with Artix; but different versions.

Here are some important deployment considerations:

- The WebSphere server runs as a single process with multiple threads. Because Artix limits creation of only one Artix bus per process by default, you must initialize different Artix buses in WebSphere with different ORB IDs. Different ORB names are not sufficient.
- WebSphere reads the shared library path from the starting shell. You must start a shell environment, source your Artix environment, and then start WebSphere from that shell. If your Artix environment is not set before starting the WebSphere server, this results in failure.
- "WebSphere configuration steps" on page 120 describes how to add core Artix JARs to the WebSphere extended classloader, but it does not include all Artix JARs. If you use a non-standard Artix subsystem (for example, AmberPoint), you may need to add additional JARs to the list.

Before you begin

Follow these general guidelines:

- Before starting WebSphere, ensure that the Artix environment script (`artix_env`) has been sourced.
- Follow the instructions in the *Actional Interceptor Guide* to enable WebSphere instrumentation.
- Ensure that a copy of `actional-sdk.jar` is present in your `$WAS_HOME/classes` directory.

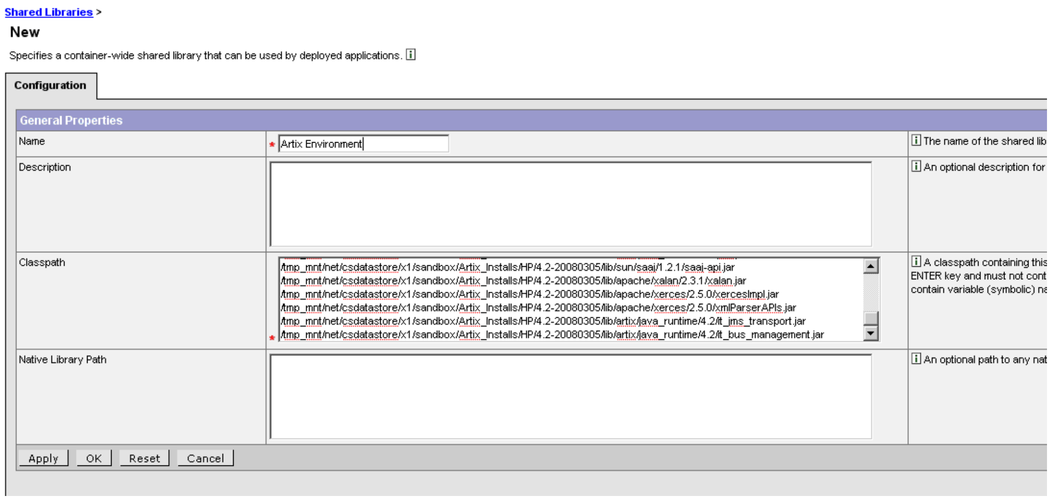
WebSphere configuration steps

In a native IBM WebSphere deployment, perform the following steps.

1. Start WebSphere server from a console that has already has the Artix environment set (using the `artix_env` script).
2. Launch the WebSphere administration console. This is generally available on: `*HostName*:9090/admin`. If no security is turned on, enter any user name to login
3. In the main menu on the left, select **Environment|Shared Libraries**.
4. On the **Shared Libraries** screen, in the `Scope` definition, select the `Server` radio button. This causes the shared library definition to applicable only the server level.
5. Click **Apply**.
6. To create a new shared library entry, select **New**.

7. In the **Configuration** tab, enter a shared library name in **Name** text box; for example, `Artix Environment` (see Figure 38).

Figure 38: *Specifying a New Shared Library*



Shared Libraries >

New

Specifies a container-wide shared library that can be used by deployed applications.

Configuration

General Properties

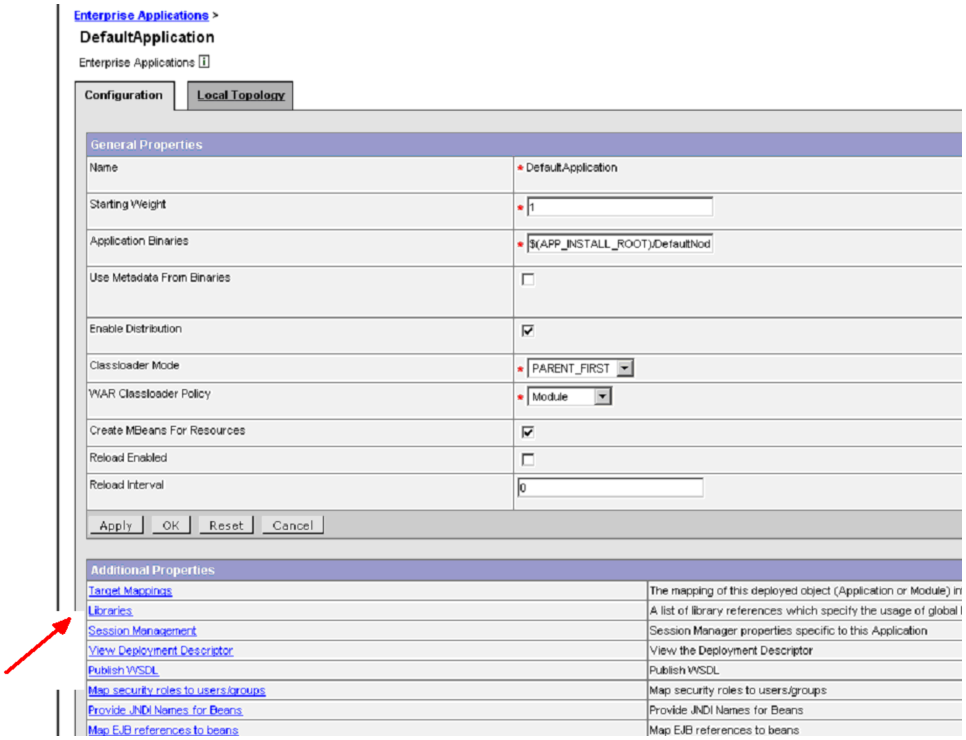
Name	* Artix Environment	The name of the shared lib
Description		An optional description for
Classpath	* /tmp_mnt/net/csdatastore/x1/sandbox/Artix_Installs/HP/4.2-20080305/lib/sun/saaj/1.2.1/saaj-api.jar /tmp_mnt/net/csdatastore/x1/sandbox/Artix_Installs/HP/4.2-20080305/lib/apache/xalan/2.3.1/xalan.jar /tmp_mnt/net/csdatastore/x1/sandbox/Artix_Installs/HP/4.2-20080305/lib/apache/xerces/2.5.0/xercesImpl.jar /tmp_mnt/net/csdatastore/x1/sandbox/Artix_Installs/HP/4.2-20080305/lib/apache/xerces/2.5.0/xmlParserAPIs.jar /tmp_mnt/net/csdatastore/x1/sandbox/Artix_Installs/HP/4.2-20080305/lib/artix/java_runtime/4.2/it_jms_transport.jar /tmp_mnt/net/csdatastore/x1/sandbox/Artix_Installs/HP/4.2-20080305/lib/artix/java_runtime/4.2/it_bus_management.jar	A classpath containing this ENTER key and must not cont contain variable (symbolic) na
Native Library Path		An optional path to any nat

8. In the **Classpath** text box, paste the full path to the following Artix JAR libraries:

```
ArtixInstallDir/etc
ArtixInstallDir/cxx_java/lib/apache/jakarta-log4j/1.2.6/log4j.jar
ArtixInstallDir/cxx_java/etc
ArtixInstallDir/cxx_java/lib/common/ifc/1.3/ifc.jar
ArtixInstallDir/cxx_java/lib/artix/java_runtime/5.1/it_bus.jar
ArtixInstallDir/cxx_java/lib/artix/java_runtime/5.1/it_bus-api.jar
ArtixInstallDir/cxx_java/lib/artix/java_runtime/5.1/it_context_library.jar
ArtixInstallDir/cxx_java/lib/ws_common/jaxrpc/1.3/it_jaxrpc.jar
ArtixInstallDir/cxx_java/lib/ws_common/saaj/1.3/it_saaj.jar
ArtixInstallDir/cxx_java/lib/ws_common/reflect/1.3/it_ws_reflect.jar
ArtixInstallDir/cxx_java/lib/ws_common/reflect/1.3/it_ws_reflect_types.jar
ArtixInstallDir/cxx_java/lib/ws_common/wsdl/1.3/it_wsdl.jar
ArtixInstallDir/cxx_java/lib/jaxrpc/jaxrpc/1.1/jaxrpc-api.jar
ArtixInstallDir/cxx_java/lib/artix/java_runtime/5.1/jms.jar
ArtixInstallDir/cxx_java/lib/sun/saaj/1.2.1/saaj-api.jar
ArtixInstallDir/cxx_java/lib/apache/xalan/2.3.1/xalan.jar
ArtixInstallDir/cxx_java/lib/apache/xerces/2.5.0/xercesImpl.jar
ArtixInstallDir/cxx_java/lib/apache/xerces/2.5.0/xmlParserAPIs.jar
ArtixInstallDir/cxx_java/lib/artix/java_runtime/5.1/it_jms_transport.jar
ArtixInstallDir/cxx_java/lib/artix/java_runtime/5.1/it_bus_management.jar
ArtixInstallDir/cxx_java/lib/artix/java_runtime/5.1/it_bus_management_monitoring.jar
ArtixInstallDir/cxx_java/lib/activemq/activemq/4.1.2.5/incubator-activemq-4.1.2.5.jar
```

9. Select **Apply** and **Save**.
10. In the main menu, select **Application**|**Enterprise Applications**.
11. Select the application that will use Artix natively (for example, `DefaultApplication` in Figure 39).
12. On the **Configuration** tab, in **General Properties**, set the **Classloader Mode** to `PARENT_LAST`.
13. Set the **WAR Classloader Policy** to `Application`.
14. Scroll down to **Additional Properties**, and select **Libraries**.

Figure 39: *Specifying Application Properties*



Enterprise Applications >

DefaultApplication

Enterprise Applications

General Properties	
Name	* DefaultApplication
Starting Weight	* 1
Application Binaries	* $(APP_INSTALL_ROOT)/DefaultNod
Use Metadata From Binaries	☐
Enable Distribution	☑
Classloader Mode	* PARENT_FIRST
WAR Classloader Policy	* Module
Create MBeans For Resources	☑
Reload Enabled	☐
Reload Interval	0

Additional Properties	
Target Mappings	The mapping of this deployed object (Application or Module) in
Libraries	A list of library references which specify the usage of global l
Session Management	Session Manager properties specific to this Application
View Deployment Descriptor	View the Deployment Descriptor
Publish WSDL	Publish WSDL
Map security roles to users/groups	Map security roles to users/groups
Provide JNDI Names for Beans	Provide JNDI Names for Beans
Map EJB references to beans	Map EJB references to beans

15. Select **Add**. This displays a list of predefined shared libraries, including the one you defined earlier.
16. Select the shared library you defined (for example, `Artix Environment`).
17. Select **Apply** and **Save**.

18. In the main menu, select **Servers|Application Servers|server1|Process Definition**.
19. In **Generic JVM Arguments**, add the following:

```
-Dorg.apache.commons.logging.LogFactory=org.apache.commons.logging.impl.LogFactoryImpl
```

20. Select **Apply** and **Save**.

Artix deployment

When deploying an Artix and Actional integration, you must specify some configuration entries in your Artix configuration file (`artix.cfg`). For example, you must configure the Artix monitoring plug-in; you can also set additional options such as payload reporting and service facade.

For full details, see "Configuring Artix Java Services for Actional Integration" on page 106.

Example Artix configuration scope

The following shows an example configuration scope from an Artix configuration file (`.cfg`):

```
demos {

    hello_world_soap_http {

   plugins:monitoring_plugin:classname="com.iona.jbus.management.monitoring.
   interceptors.MonitoringPlugInFactory";

    plugins:monitoring_plugin:know_report_tool= "actional";

    orb_plugins = ["local_log_stream", "xmlfile_log_stream", "soap", "java"];

    java_plugins = ["monitoring_plugin"];

    binding:artix:client_request_interceptor_list= "monitoring_handler";
    binding:artix:client_message_interceptor_list= "monitoring_handler";
    binding:artix:server_request_interceptor_list= "monitoring_handler";
    binding:artix:server_message_interceptor_list= "monitoring_handler";

    plugins:monitoring_plugin:enable_si_payload="true";
    plugins:monitoring_plugin:max_reported_payload_size="-1";
    plugins:monitoring_plugin:show_service_facade="false";
    };
};
```

Further information

Actional

For information on how to set up and run the Actional server, Actional agent, and Actional Server Administration Console, see the Actional product documentation.

Artix

For more information on Artix configuration, see the following:

- Configuring and Deploying Artix Solutions, C++ Runtime
- Artix Configuration Reference, C++ Runtime

IBM WebSphere

For more details on WebSphere, see the IBM WebSphere product documentation.

Artix J2EE Connector

For more details on Artix J2EE Connector, see Artix for J2EE.

Part IV
AmberPoint

In this part

This part contains the following chapters:

CHAPTER 9

Integrating with AmberPoint

Artix provides support for integration with the AmberPoint SOA management system. This chapter describes two approaches to integrating Artix services with AmberPoint.

In this chapter

This chapter includes the following sections:

AmberPoint Proxy Agent

Overview

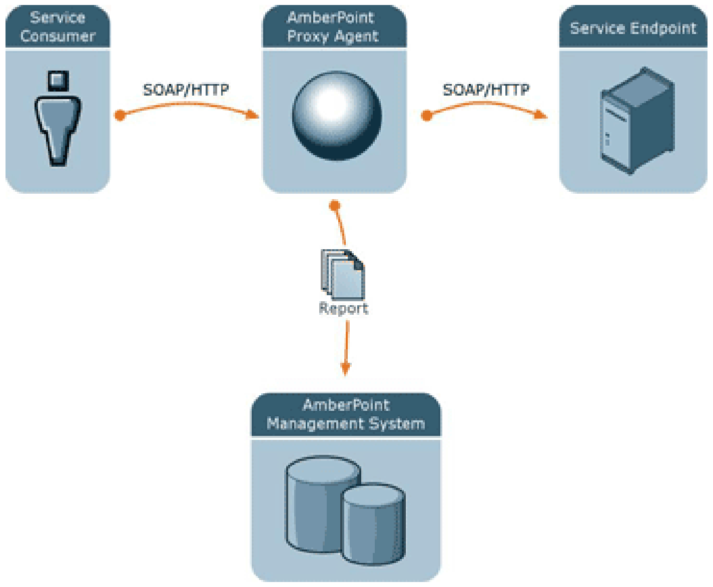
There are two possible approaches to integrating Artix with the AmberPoint SOA management system:

- AmberPoint Proxy Agent
- Artix AmberPoint Agent

AmberPoint Proxy Agent architecture

AmberPoint provides the AmberPoint Proxy Agent, which acts as a proxy for Web service endpoints by making the service endpoint WSDL available to the service consumer (client). Figure 40 shows a simple AmberPoint Proxy Agent architecture:

Figure 40: *AmberPoint Proxy Agent Integration*



In this architecture, the following restrictions apply:

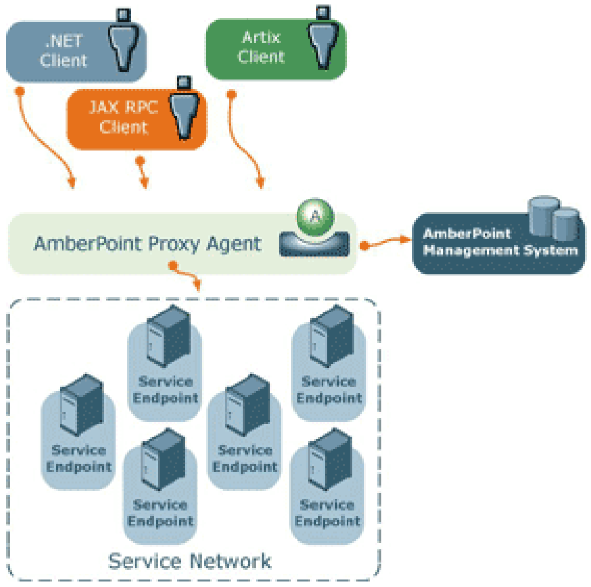
- All messages between the service consumer and service endpoint must be routed through the AmberPoint Proxy Agent.
- All messages must use SOAP over HTTP.
- The service consumer is unaware of the back-end service endpoint, and views its relationship as being with the proxy only.

If you can work within these limits, the AmberPoint monitoring and management features can be used out-of-the box with Artix. However, if you require a more flexible integration (for example, with increased performance and scalability), you should use the Artix AmberPoint Agent.

AmberPoint Proxy Agent in a service network

Figure 41 shows the AmberPoint Proxy Agent deployed in a service network with multiple service consumers and service endpoints.

Figure 41: *AmberPoint Proxy Agent Service Network*



Because all messages are routed through the AmberPoint Proxy Agent, the additional network hops may impact on performance. In addition, the proxy involves the risk of a single point of failure.

If these are important issues for your system, you should use the Artix AmberPoint Agent instead.

Further information

For information on using the AmberPoint Proxy Agent, see the AmberPoint product documentation.

Artix AmberPoint Agent

Overview

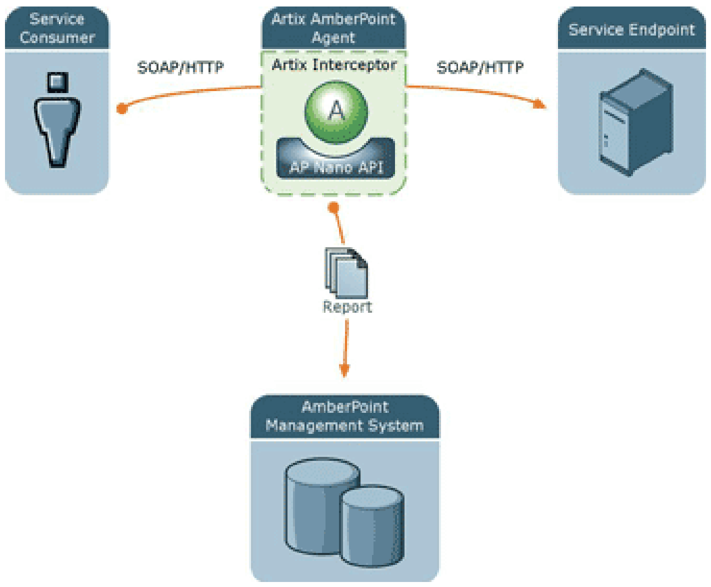
The Artix AmberPoint Agent enables Artix endpoints to be discovered and monitored by AmberPoint. This is the recommended approach to integrating Artix services with AmberPoint, and can be used with Artix services implemented in C++, JAX-RPC, JAX-WS, and scripting languages.

The Artix AmberPoint Agent can be deployed with Artix endpoints that use SOAP over HTTP to enable reporting of performance metrics back to AmberPoint. The Artix AmberPoint Agent offers significant benefits over the AmberPoint Proxy Agent. For example, these include increased performance and scalability, dynamic discovery, and the use of callbacks. This section describes the Artix AmberPoint Agent in detail.

Artix AmberPoint Agent architecture

Figure 42 shows how Artix can be integrated with AmberPoint using the Artix AmberPoint Agent.

Figure 42: *Artix AmberPoint Agent Integration*



The main components in this architecture are:

- "Artix AmberPoint Agent"
- "Artix interceptor"
- "Artix service endpoints"
- "Service consumers"
- "AmberPoint SOA Management System"
- "AmberPoint Nano Agent API"

Note: Integration with the Artix AmberPoint Agent currently applies to SOAP over HTTP, and services that have one endpoint only.

Artix AmberPoint Agent

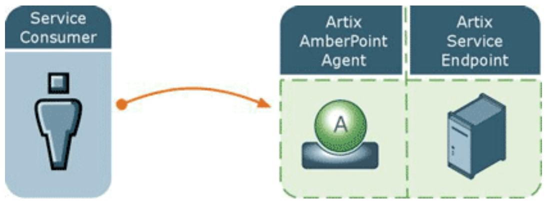
An Artix AmberPoint Agent consists of components developed by IONA and AmberPoint (the Artix interceptor, and the AmberPoint Nano Agent API). You can deploy multiple agents into your SOA network to capture data for the AmberPoint management system. Artix AmberPoint Agents gather performance data for all Artix endpoint types, as well as normal Web service endpoints.

Deployment modes

Artix AmberPoint Agents can be deployed in different ways in your system, for example:

- *Embedded in Artix consumers intercepting traffic*. This is suitable if Artix is deployed on the client side only, and the service endpoints do not support AmberPoint. This requires configuration for the consumer only.
- *Embedded in Artix service endpoints intercepting traffic*. This is suitable if Artix is used to implement the service endpoint. This works even when the consumers are third party products. This requires configuration for the service endpoint only. This is the most common and recommended approach, as shown in Figure 43.
- *Deployed as standalone Artix intermediaries (proxies) on your service network*. This option is suitable if you do not want touch your existing system and you do not want to update your endpoints or consumers. This approach is also necessary if Artix is not deployed at either the consumer or service endpoints.

Figure 43: *Artix AmberPoint Agent Embedded in Service Endpoint*



Artix interceptor

An Artix interceptor is deployed on the dispatch path of all messages exchanged between Artix service endpoints and consumers. It may be deployed in the same process as the consumer and/or the endpoint, or as an intermediary between the consumer and service.

The Artix interceptor captures all data in the dispatch path. The Artix interceptor then reports performance metrics using the AmberPoint nano agent API.

Artix service endpoints

An Artix service endpoint is a service built using Artix, and described using WSDL. The endpoint can be implemented using C++, JAX-RPC, JAX-WS, or even a scripting language, such as JavaScript. However, its main characteristic is that it can be described in WSDL, and classified as a service, which can therefore be consumed. The Artix AmberPoint Agent provides a WSDL contract describing the endpoint that is being monitored.

Service consumers

Service consumers are clients that consume service endpoints by exchanging messages based on the service interface. Consumers can be built using Artix, or any product that supports the technology used by the endpoint. For example, a pure CORBA client could be a consumer for a CORBA endpoint. A .NET client could be a consumer for an Artix SOAP endpoint.

AmberPoint SOA Management System

In this document, AmberPoint is the general term used to describe the system in which all performance metrics are stored and viewed. For the purposes of this document, all interactions are made using the AmberPoint Nano Agent API, and the AmberPoint graphical tools are used to view the Artix data. This simplifies the architecture of AmberPoint for the sake of this discussion.

AmberPoint Nano Agent API

The AmberPoint Nano Agent API is a Java public API provided by AmberPoint that enables customers to monitor their endpoints. This is the API that Artix uses to notify AmberPoint of the existence of the service endpoint. Artix also uses the AmberPoint nano agent API at runtime to report performance metrics about a previously registered endpoint.

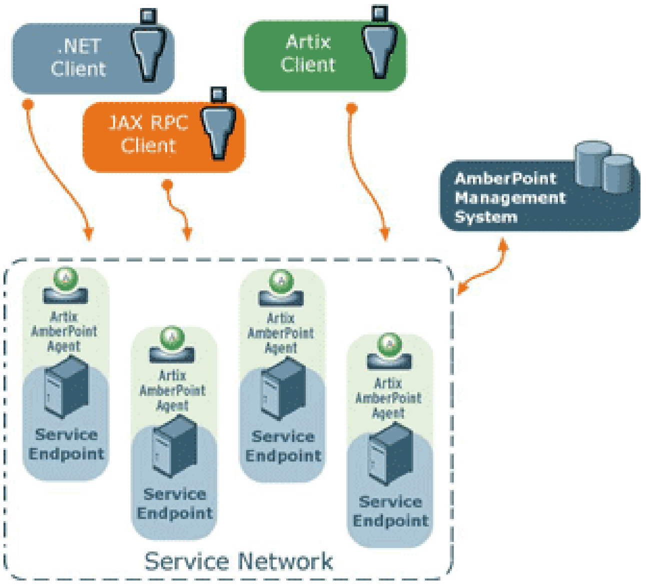
The AmberPoint Nano Agent API enables the Artix interceptor to do the following:

- Allow dynamic discovery of new Artix endpoints without manual registration of the endpoints by the user. This registration process assumes that the Artix interceptor has the required configuration for the nano agent to contact AmberPoint. When the Artix AmberPoint Agent becomes active, it uses the Nano Agent API to register a new endpoint.
- Allow periodic reporting of messages using the Artix interceptor. These reports contain performance data about the endpoint and the messages being exchanged.

Artix AmberPoint Agent in a service network

Figure 44 shows the Artix AmberPoint Agent deployed in a service network with multiple service consumers and service endpoints.

Figure 44: *Artix AmberPoint Agent Service Network*



This loosely-coupled architecture has the following benefits:

- Because the Artix AmberPoint Agent is collocated and embedded in the service endpoint, there are no additional network hops, so performance is maximized.
- Unlike with the AmberPoint Proxy Agent, there is no risk of a single point of failure, so reliability and scalability are also improved.
- An Artix AmberPoint Agent can be embedded into an Artix router.This enables it to dynamically discover and monitor the Artix service endpoints and consumers that the router creates and manages.
- Because the client is aware of the back-end service endpoint, the use of callbacks is supported.

Supported AmberPoint features

The Artix AmberPoint Agent enables the use of the following AmberPoint features:

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

Further information

For detailed information on using AmberPoint features, see the AmberPoint product documentation.

CHAPTER 10

Configuring the Artix AmberPoint Agent

This chapter explains how to set up integration with the Artix AmberPoint Agent, and shows examples from the Artix AmberPoint integration demos.

In this chapter

This chapter includes the following sections:

Installing AmberPoint

Overview

The Artix ESB C++ runtime supports integration with version 5.1 of the AmberPoint SOA management system. This section explains how to install AmberPoint to enable integration with the Artix AmberPoint Agent.

Installation steps

When installing the AmberPoint runtime, perform the following steps:

1. In the AmberPoint installation wizard, choose a suitable HTTP port number for the J2EE application server in which the AmberPoint server will be deployed (for example, `9090`).
2. AmberPoint comes bundled with Tomcat application server, so for the demo purposes, choose to install Tomcat.
3. Select **Deploy AmberPoint into the container**.
4. Select **Install a Java VM specifically for this application**.
5. Select **Deploy a new sphere with the SOA Management System**. This deploys the persistence runtime into the J2EE application server, and configures it to use the embedded Tomcat HSQL relational database management system.
6. You can also install AmberPoint sample Web services, but these are not required.
7. Provide a user name and password with administrative privileges (for example, `admin`/`admin`).
8. When installation is complete, copy the AmberPoint Nano Agent Server into the deployment directory of the application server. For example, for Tomcat, use the following command:

```
copy AP_InstallDir/add_ons/socket_converter/apsocketconverter.war
    AP_InstallDir/server/webapps
```

If you are not using Tomcat, use the vendor's visual tools to deploy `apsocketconverter.war` into the application server.

Configuring AmberPoint for Artix Integration

Overview

This section explains how to configure the AmberPoint SOA management system for integration with Artix.

Starting the AmberPoint Server

When you have completed the AmberPoint installation steps, run the AmberPoint server using Window's **Start** menu.

Alternatively, execute the following script:

Windows	`AP_InstallDir\server\bin\startup.bat`
UNIX	`AP_InstallDir/server/bin/startup.sh`

You can see how your application server starts up and deploys the AmberPoint server in the log files in the `AP_InstallDir/server/logs` directory.

Configuring the AmberPoint Nano Agent Sever

When the application server has started and deployed all the AmberPoint `.war` files, perform the following steps:

1. Open a web browser and specify the following URL:

   `http://hostname:port/apasc/`
2. Login using the admin user name and password that you provided when installing AmberPoint.
3. When logged in, click **Network|Infrastructure** in the tabbed menu. This displays a list of registered **Deployments** with this application server's container.
4. Ensure that one of the deployed items is named `apsocketconverter` and has a green button beside it This indicates that the AmberPoint Nano Agent Server has been successfully deployed and is ready to be configured.

5. In the left pane, click the **Register** button.

   From the drop-down menu, select **Message Source**|**Simple Message Source**: This displays the **Register Message Source** form.

6. In the **Register Message Source** form, enter the following:

Name	`Artix Message Source`
Type of Message Source	`File`
Start At	`At present`
Location	`*AmberPointInstallDir*\server\amberpoint\apsocketconverter\logdir`

   The source **Name** can be any string value. The **Location** specifies the location of the log file for incoming messages. The default **Criteria for this policy** applies this message source to all active services that this AmberPoint system is aware of.

7. Without modifying the **Criteria for this policy,** click **Preview Services** to see which services this message Source applies to. If you have no services currently registered, only one service named **MonitorEnabler** is displayed.

8. Click the **Go** button at the top left of the screen, and wait until the **Policy Status** is `Applied`.

9. Return to a command window to build an Artix AmberPoint demo (see "Configuring Artix C++ Services for AmberPoint Integration" on page 30).

Configuring the AmberPoint port

If the default AmberPoint Nano Agent Server port (`33333`) does not suit your setup, change the following attributes to the new port number:

- `messageLogWriter logLocation` in your Artix `apobserver.configuration` file
- `messageLogReader logLocation` in:

```
AP_InstallDir/server/webapps/apsocketconverter.war@/WEB-INF/
    application/resources/readerConfig.xml
```

Whenever you update values in the Artix `apobserver.configuration` file, you must restart the services already being monitored by the Artix AmberPoint Agent for the changes to take effect.

If you update the Nano Agent Server port, you may need to restart the application server for changes to take effect (except for those servers that support hot deployment).

For example, these settings appear as follows in the Artix `apobserver.configuration` file:

```
...
<ap:messageLogWriter
    logWriterImplClass="com.amberpoint.msglog.socketimpl.SocketLogWriter"
        logName="{hostname}" <!-- default = localhost -->
        logLocation="{port}" <!-- default = 33333 -->
        syncEverySoManyEntries="50">
</ap:messageLogWriter>
  ...
<ap:hostMapper algorithm="asSent" urlProperty="ap:requestURL"/>
  ...
<ap:hostMapper algorithm="asSent" urlProperty="ap:wsdlUrl"/>
  ...
```

Configuring Artix C++ Services for AmberPoint Integration

Overview

This section explains how to configure Artix C++ and JAX-RPC services to support the Artix AmberPoint Agent. It describes Artix AmberPoint demo configuration settings in detail. However, if your AmberPoint installation and demo run on the same host, you do not need to make any configuration changes to run the demo. If you wish to run the demo now, skip this section, and see the `readme.txt` in the following directory:

```
ArtixInstallDir/cxx_java/samples/integration/amberpoint
```

This `amberpoint` demo is based on the `.../samples/routing/content_based` demo, with some modifications to enable Artix and AmberPoint integration.

Configuring the AmberPoint Nano Agent plug-in

You must enable the AmberPoint Nano Agent plug-in for the Artix runtime. For example, the configuration scope in which the demo servers run includes an Artix plug-in named `ap_nano_agent`. This is loaded into the Artix runtime, and enables discovery and monitoring by AmberPoint of services and consumers running inside Artix processes.

```
demos {
    content_based {
      orb_plugins = ["xmlfile_log_stream", "soap", "at_http", "ap_nano_agent"];
    ...
   }
 ...
}
```

In this demo, there are three server instances, each exposing the same interface but running under different service and endpoint name pairs. These are as follows:

```
{TargetService1, TargetPort1}
{TargetService2, TargetPort2}
{TargetService3, TargetPort3}
```

Configuring the Artix router

To enable router support, you must also add the AmberPoint Nano Agent plug-in to the router's configuration. For example, the demo configuration scope in which the Artix router runs includes additional configuration for the Artix `routing` plug-in. Its `orb_plugins` list includes the `ap_nano_agent` plug-in, which enables the router's endpoints and consumers to be discovered and monitored by AmberPoint.

```
demos {
  content_based {
    ...
    router {
     orb_plugins = [ "xmlfile_log_stream", "ap_nano_agent", "routing" ];
      plugins:routing:use_pass_through="false";
     ...
    }
  }
}
```

The `ap_nano_agent` plug-in must precede the `routing` plug-in. This is because the Artix AmberPoint Agent must register itself in the interceptor chain before the routing plug-in instantiates and activates the services that it manages.

Setting `plugins:routing:use_pass_through` to `false` disables passing data through the router without parsing. The `ap_nano_agent` plug-in requires that the underlying payload is parsed in the Artix type format.

Configuring the consumer hostname

`plugins:ap_nano_agent:hostname_address:publish_hostname` specifies the form in which the Artix AmberPoint Agent resolves the host address that an Artix service consumer (proxy) runs on. This variable takes the following values:

`unqualified`	The host name in short form, without the domain name (*`hostname`*).
`ipaddress`	The host name in the form of an IP address (for example, `123.4.56.789`). This is the default.
`canonical`	The host name takes a fully qualified form (*`hostname.domainname`*).
`true`	same as `unqualified`
`false`	same as `ipaddress`

`plugins:ap_nano_agent:hostname_address:local_hostname` is an arbitrary string used as the client hostname instead of trying to resolve it using the underlying IP runtime. This is undefined by default.

To report the correct service consumer address invoking to an Artix service monitored by this agent, specify the following setting in the client and server configuration scope:

```
plugins:bus:register_client_context="true";
```

Configuring the service hostname

The server-side host name resolution is driven by the specific transport. Because the HTTP transport is the only one currently supported the following variables must be configured:

- `policies:soap:server_address_mode_policy:publish_hostname`
- `policies:at_http:server_address_mode_policy:publish_hostname`

Possible values are the same as those for `plugins:ap_nano_agent:hostname_address:publish_hostname`.

These variables specify the format that a service endpoint address is published to service consumers. AmberPoint discovers Artix services by consuming a published WSDL contract. It correlates the address in the WSDL with the inflow of log messages that describe operations invoked on an endpoint. This means that you must synchronize these configuration values with the configuration values of the AmberPoint Client Nano Agent.

Configuring the AmberPoint hostname

The default Artix hostname resolution setting is `ipaddress`, which is the same as that for the configuration of AmberPoint Client Nano Agent. However, if you change the Artix hostname resolution, you must also update the AmberPoint Client Nano Agent configuration file. For example:

```
ArtixInstallDir/cxx_java/etc/amberpoint/5.1/nanoagent/conf/apobserver.configuration
```

To update the hostname resolutions setting, open the file in a text editor and find the two occurrences of the `hostMapper algorithm` attribute.

You must update the value of `hostMapper algorithm` attribute if you change the value of `policies:soap:server_address_mode_policy:publish_hostname` and `policies:at_http:server_address_mode_policy:publish_hostname` configuration variables.

The equivalent AmberPoint values are as follows:

Artix publish_hostname variable	AmberPoint hostMapper algorithm
`ipaddress`	`useIpAddr` or `asSent`
`canonical`	`useFQN` or `asSent`
`unqualified`	`asSent`

To avoid updating the AmberPoint Nano Agent Client configuration each time you change the Artix configuration, simply use `hostMapper algorithm="asSent"`.

If you are running your Artix services and the AmberPoint Nano Agent Server on different machines, you must also update the `messageLogWriter logName` attribute to point the host name or IP address where the Nano Agent Server is running.

Configuring the AmberPoint port

If the default AmberPoint Nano Agent Server port (`33333`) does not suit your setup, you can update your AmberPoint configuration file to the new port number. For more details, see "Configuring the AmberPoint port" on page 29.

Viewing Artix services in AmberPoint

When you run the demo, and start the Artix router and servers, and make client invocations to the router, these calls are in turn forwarded on to the servers.

AmberPoint dependency diagrams

While the demo is running, in the AmberPoint GUI, select the **Network|Services|Dependencies** screen. AmberPoint tracks the call flow, as it happens, between Artix services with the Artix AmberPoint Agent in their runtime. The dependency flow diagram is a directed graph, and can be of any complexity. For example, a client makes three calls to the source service implemented by the router. Each call is routed to the intended destination service, defined by the routing rules. Each `TargetService` receives a single call out of the three made. And each dependency tracking is shown in relation to the service selected in the **Selector** list, which is referred as a primary service.You can manually create dependencies between services using the AmberPoint tools if so desired. See the

AmberPoint user documentation for details on what you can do with dependency diagrams (for example, using the **Network|Services|Dependencies** screen).

AmberPoint performance diagrams

You can use the AmberPoint **Performance|Activity** screen to view performance statistics. See the AmberPoint user documentation for details on what you can do with performance statistics.

AmberPoint logging policies

You can collect call logs by adding an AmberPoint logging policy using the **Exceptions|Services** screen. To add an AmberPoint logging policy, click the **Add Logging Policy** button at the top of the screen. This displays the **Add Policy** form,. Use this form to specify a meaningful name, and tune its parameters to your needs. If you wish to log messages for all available services, edit the policy rules at the bottom of this form.

When the log policy is created, you must wait until it is applied, like when you created a **Message Source** (see "Configuring the AmberPoint Nano Agent Sever" on page 27). After the log policy has been applied and turns green, send some more traffic using the demo. You can then watch the **Message Log** using the **Exceptions|Services|Message Log** tab.

Further information

There are many other AmberPoint features that you can use with Artix. For example, when AmberPoint has captured the Artix traffic, you can use its runtime to define customers and their SLAs, and map these SLAs to the services in the network. You can also create reactions (alerts) if an SLA violation has occurred and so on. See the AmberPoint user documentation for more details.

Artix AmberPoint demo

For more details on the Artix AmberPoint integration demo, see:

```
ArtixInstallDir\cxx_java\samples\integration\amberpoint\README.txt
```

Artix C++ configuration

- Configuring and Deploying Artix Solutions, C++ Runtime
- Artix Configuration Reference, C++ Runtime

Part V
BMC Patrol

In this part

This part contains the following chapters:

Integrating with BMC Patrol™

This chapter introduces Artix ESB's integration with the BMC Patrol™ Enterprise Management System. It describes the requirements and main components of this integration.

In this chapter

This chapter contains the following sections:

Introduction

Overview

Artix ESB supports integration with Enterprise Management Systems such as BMC Patrol. This section includes the following topics:

- "The application life cycle"
- "Enterprise Management Systems"
- "Artix BMC Patrol features"
- "How it works"

The application life cycle

Most enterprise applications go through a rigorous development and testing process before they are put into production. When applications are in production, developers rarely expect to manage those applications. They usually move on to new projects, while the day-to-day running of the applications is managed by a production team. In some cases, the applications are deployed in a data center that is owned by a third party, and the team that monitors the applications belongs to a different organization.

Enterprise Management Systems

Different organizations have different approaches to managing their production environment, but most will have at least one *Enterprise Management System* (EMS).

For example, the main Enterprise Management Systems include BMC Patrol™ and IBM Tivoli™. These systems are popular because they give a top-to-bottom view of every part of the IT infrastructure.

This means that if an application fails because the `/tmp` directory fills up on a particular host, for example, the disk space is reported as the fundamental reason for the failure. The various application errors that arise are interpreted as symptoms of the underlying problem with disk space. This is much better than being swamped by an event storm of higher-level failures that all originate from the same underlying problem. This is the fundamental strength of integrated management.

Artix BMC Patrol features

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

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

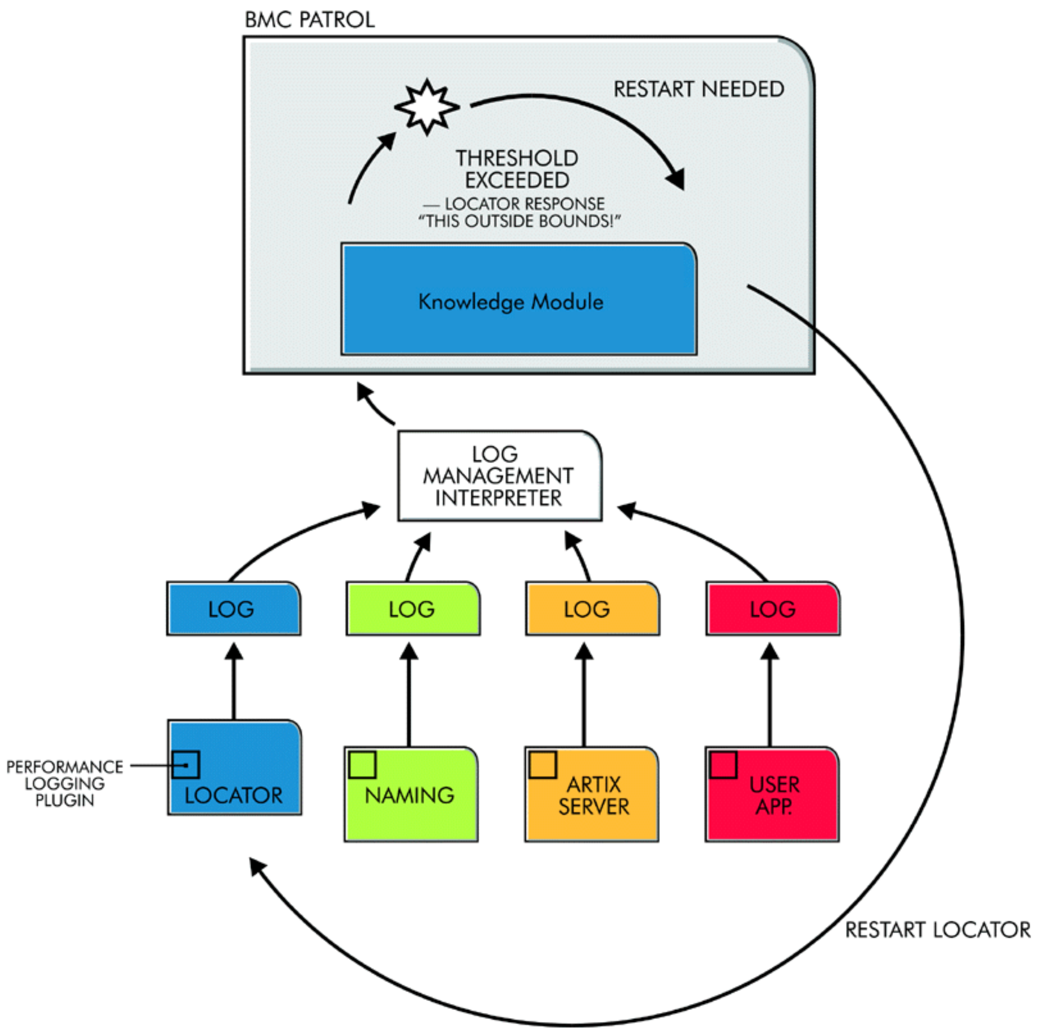
The server metrics tracked by the Artix BMC Patrol integration include the number of invocations received, and the average, maximum and minimum response times. The Artix BMC Patrol integration also enables you to track these metrics for individual operations. Events can be generated when any of these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

How it works

In the BMC Patrol integration, key server metrics are logged by the Artix performance logging plug-ins. Log file interpreting utilities are then used to analyze the logged data.

Artix also provides Knowledge Modules, which conform to standard BMC Knowledge Module design and operation. These modules tell the BMC Patrol console how to interpret the logging data received from the Artix services. Figure 45 on page 40 shows a simplified view of how the Knowledge Modules work. In this example, an alarm is triggered in the BMC Patrol console when a locator becomes unresponsive, and this results in an action to restart the locator.

Figure 45: *Overview of the Artix BMC Patrol Integration*



The performance logging plug-ins collect data relating to server response times and log it periodically in the performance logs. The Knowledge Module executes parameter collection periodically on each host, using the log file interpreter running on each host to collect and summarize the logged data.

The Knowledge Module compares the response times and other values against the defined alarm ranges, and issues an alarm event if a threshold has been breached. These events can be analyzed and appropriate action taken automatically (for example, restart a server). Alternatively, the user can intervene manually and execute a BMC Patrol menu command to stop, start or restart the offending server.

The Artix BMC Patrol Integration

Overview

This section describes the requirements and main components of the Artix BMC Patrol integration. It includes the following topics:

- "BMC Patrol requirements"
- "Main components"
- "Example metrics"
- "Further information"

BMC Patrol requirements

To use the Artix BMC Patrol integration, you must have BMC Patrol 3.4 or higher. The BMC Patrol integration is compatible with the BMC Patrol 7 Central Console.

Main components

The BMC Patrol integration consists of the following Knowledge Modules (KM):

- `IONA_SERVERPROVIDER`
- `IONA_OPERATIONPROVIDER`

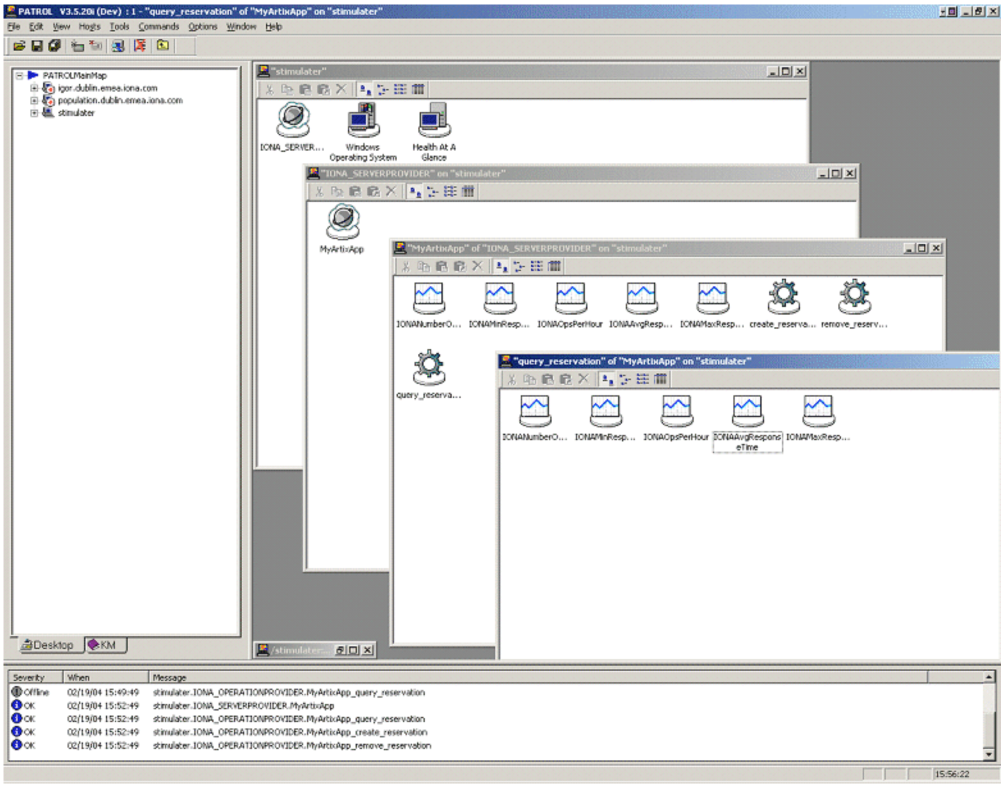
`IONA_SERVERPROVIDER.km` tracks key metrics associated with your Artix servers on a particular host. It also enables servers to be started, stopped, or restarted, if suitably configured.

`IONA_OPERATIONPROVIDER.km` tracks key metrics associated with individual operations on each server.

Example metrics

Figure 46 shows an example of the `IONA_SERVERPROVIDER` Knowledge Module displayed in BMC Patrol. The window in focus shows the IONA performance metrics that are available for an operation named `query_reservation`, running on a machine named `stimulator`.

Figure 46: *IONA Server Running in BMC Patrol*


PATROL V3.5.20i (Dev) : 1 - "query_reservation" of "MyArtixApp" on "stimulater"
File Edit View Hosts Tools Commands Options Window Help
PATROLMainMap
igor.dublin.emea.iona.com
population.dublin.emea.iona.com
stimulater
"stimulater"
IONA_SERVER...
Windows Operating System
Health At A Glance
"IONA_SERVERPROVIDER" on "stimulater"
MyArtixApp
"MyArtixApp" of "IONA_SERVERPROVIDER" on "stimulater"
IONANumberO...
IONAMinResp...
IONAOpsPerHour
IONAAvgResp...
IONAMaxResp...
create_reserva...
remove_reserv...
query_reserva...
"query_reservation" of "MyArtixApp" on "stimulater"
IONANumberO...
IONAMinResp...
IONAOpsPerHour
IONAAvgResponseTime
IONAMaxResp...
Desktop
KM
/stimulater...
15:56:22

```
Severity  When               Message
Offline   02/19/04 15:49:49  stimulater.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation
OK        02/19/04 15:52:49  stimulater.IONA_SERVERPROVIDER.MyArtixApp
OK        02/19/04 15:52:49  stimulater.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation
OK        02/19/04 15:52:49  stimulater.IONA_OPERATIONPROVIDER.MyArtixApp_create_reservation
OK        02/19/04 15:52:49  stimulater.IONA_OPERATIONPROVIDER.MyArtixApp_remove_reservation
```

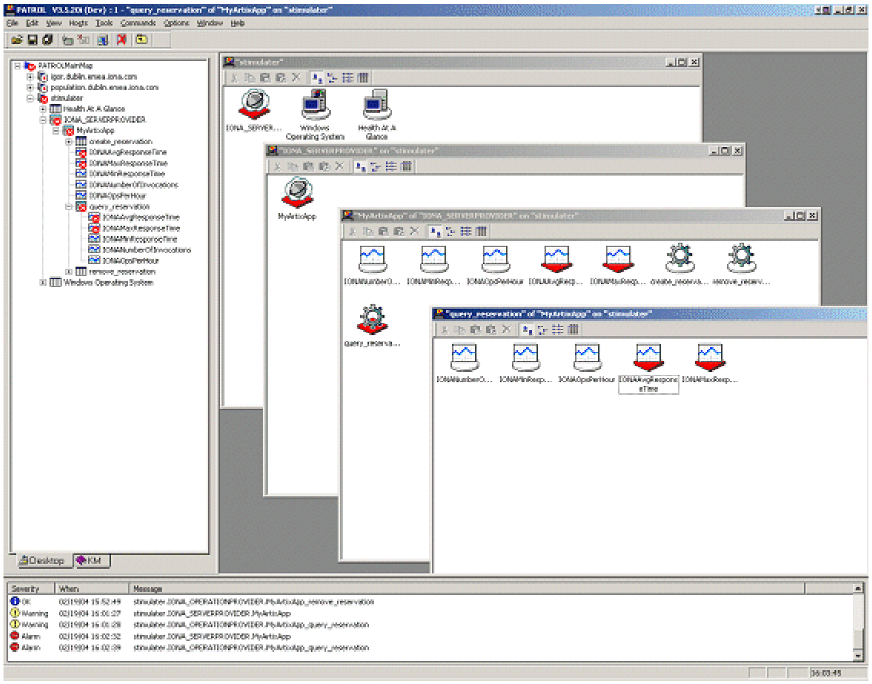
The IONA server performance metrics include the following:

- `IONAAvgResponseTime`
- `IONAMaxResponseTime`
- `IONAMinResponseTime`
- `IONANumInvocations`
- `IONAOpsPerHour`

For more details, see "Using the Artix Knowledge Module" on page 52.

Figure 47 shows alarms for server metrics, for example, `IONAAvgResponseTime`. This measures the average response time of all operations on this server during the last collection cycle.

Figure 47: *BMC Patrol Displaying Alarms*



```
Severity   When               Message
OK         02/19/04 15:52:49  stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_remove_reservation
Warning    02/19/04 16:01:27  stimulator.IONA_SERVERPROVIDER.MyArtixApp
Warning    02/19/04 16:01:28  stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation
Alarm      02/19/04 16:02:32  stimulator.IONA_SERVERPROVIDER.MyArtixApp
Alarm      02/19/04 16:02:39  stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation
```

Further information

For a detailed description of Knowledge Modules, see your BMC Patrol documentation.

Configuring Artix for BMC Patrol

This chapter explains the steps that you need to perform in your IONA product to configure integration with BMC Patrol.

In this chapter

This chapter contains the following sections:

Setting up your Artix Environment

Overview

The best way to learn how to use the BMC Patrol integration is to start with a host that has both BMC Patrol and Artix installed. This section explains how to make your Artix servers visible to BMC Patrol. It includes the following topics:

- "EMS configuration files"
- "Creating a servers.conf file"
- "Creating a server_commands.txt file"
- "Further information"

EMS configuration files

You need to create two text files that are used to configure the BMC Patrol integration:

- `servers.conf`
- `server_commands.txt`

These files are used to track your Artix applications in BMC Patrol. You will find starting point files in the `IONA_km.zip` located in the following directory of your Artix installation:

```
ArtixInstallDir\cxx_java\management\BMC\IONA_km.zip
```

When you unzip, the starting point files are located in the `lib/iona/conf` directory.

Creating a servers.conf file

The `servers.conf` file is used to instruct BMC Patrol to track your Artix servers. It contains the locations of performance log files for specified applications. Each entry must take the following format:

```
my_application, 1, /path/to/myproject/log/myapplication_perf.log
```

This example entry instructs BMC Patrol to track the `myapplication` server, and reads performance data from the following log file:

```
/path/to/myproject/log/myapplication_perf.log
```

You must add entries for the performance log file of each Artix server on this host that you wish BMC Patrol to track. BMC Patrol uses the `servers.conf` file to locate these log files, and then scans the logs for information about the server's key performance indicators.

The following example is taken from the Artix Java sample application for BMC Patrol integration:

```
management-bmc-patrol-demo-server,1,%ARTIX_HOME%\java\samples\advanced
    \management\bmc-patrol\BMCCounterServer.log
management-bmc-patrol-demo-client,1,%ARTIX_HOME%\java\samples\advanced
    \management\bmc-patrol\BMCCounterClient.log
```

Creating a server_commands.txt file

The `server_commands.txt` file is used to instruct BMC Patrol how to start, stop, and restart your Artix servers. It contains the locations of the relevant scripts for specified servers. Each entry must take the following format:

```
myapplication,start=/path/to/myproject/bin/start_myapplication.sh
myapplication,stop=/path/to/myproject/bin/stop_myapplication.sh
myapplication,restart=/path/to/myproject/bin/restart_myapplication.sh
```

In this example, each entry specifies a script that can be used to stop, start, or restart the `myapplication` server. When BMC Patrol receives an instruction to start `myapplication`, it looks up the `server_commands.txt` file, and executes the script specified in the appropriate entry.

You must add entries that specify the relevant scripts for each server on this host that you wish BMC Patrol to track.

Copy the EMS files to your BMC installation

When you have added content to your `servers.conf` and `server_commands.txt` files, copy these files into your BMC installation, for example:

```
$PATROL_HOME/lib/iona/conf
```

This enables tracking of your Artix server applications in BMC Patrol.

Further information

For details of how to configure your Artix servers to use performance logging, see "Configuring an Artix Production Environment" on page 60.

For a complete explanation of configuring performance logging, see Configuring and Deploying Artix Solutions, C++ Runtime.

CHAPTER 13

Using the Artix BMC Patrol Integration

This chapter explains the steps the that you must perform in your BMC Patrol environment to monitor Artix applications. It also describes the Artix Knowledge Module and how to use it to monitor servers and operations. It assumes that you already have a good working knowledge of BMC Patrol.

In this chapter

This chapter contains the following sections:

Setting up your BMC Patrol Environment

Overview

To enable monitoring of the Artix servers on your host, you must first perform the following steps in your BMC Patrol environment:

1. "Install the Knowledge Module"
2. "Set up your Java environment"
3. "Set up your EMS configuration files"
4. "View your servers in the BMC Console"

Install the Knowledge Module

The Artix BMC Patrol Knowledge Module is shipped in two formats:

Windows	*ArtixInstallDir*\cxx_java\management\BMC\IONA_km.zip
UNIX	*ArtixInstallDir*/cxx_java/management/BMC/IONA_km.tgz

To install the Artix Knowledge Module:

Windows

Use WinZip to unzip `IONA_km.zip`. Extract this file into your `%PATROL_HOME%` directory.

If this is successful, the following directory is created:

```
%PATROL_HOME%\lib\iona
```

UNIX

Copy the `IONA_km.tgz` file into `$PATROL_HOME`, and enter the following commands:

```
$ cd $PATROL_HOME
$ gunzip IONA_km.tgz
$ tar xvf IONA_km.tar
```

Set up your Java environment

The Artix Knowledge Module requires a Java Runtime Environment (JRE). If your BMC Patrol installation already has a `$PATROL_HOME/lib/jre` directory, it should work straightaway. If not, you must setup a JRE (version 1.3.1 or later) on your machine as follows:

1. Copy the `jre` directory from your Java installation into `$PATROL_HOME/lib`. You should now have a directory structure that includes `$PATROL_HOME/lib/jre`.
2. Confirm that you can run `$PATROL_HOME/lib/jre/bin/java`.

Set up your EMS configuration files

In Chapter 12, you generated the following EMS configuration files:

- `servers.conf`
- `server_commands.txt`

Copy these generated files to `$PATROL_HOME/lib/iona/conf`.

View your servers in the BMC Console

To view your servers in the **BMC Console**, and check that your setup is correct:

1. Start your **BMC Console** and connect to the **BMC Patrol Agent** on the host where you have installed the IONA Knowledge Module.
2. In the **Load KMs** dialog, open the `$PATROL_HOME/lib/knowledge` directory, and select the `IONA_SERVER.kml` file. This will load the `IONA_SERVERPROVIDER.km` and `IONA_OPERATIONPROVIDER.km` Knowledge Modules.
3. In your **Main Map**, the list of servers that were configured in the `servers.conf` file should be displayed. If they are not currently running, they are shown as offline.

You are now ready to manage these servers using BMC Patrol.

Using the Artix Knowledge Module

Overview

This section describes the Artix Knowledge Module and explains how to use it to monitor servers and operations. It includes the following topics:

- "Server Provider parameters"
- "Monitoring servers"
- "Monitoring operations"
- "Operation parameters"
- "Starting, stopping and restarting servers"
- "Troubleshooting"

Server Provider parameters

The `IONA_SERVERPROVIDER` class represents instances of Artix server or client applications. The parameters exposed in the Knowledge Module are shown in Table 9.

Table 9: *Artix Server Provider Parameters*

Parameter Name	Default Warning	Default Alarm	Description
IONAAvgResponseTime	1000–5000	> 5000	The average response time (in milliseconds) of all operations on this server during the last collection cycle.
IONAMaxResponseTime	1000–5000	> 5000	The slowest operation response time (in milliseconds) during the last collection cycle.
IONAMinResponseTime	1000–5000	> 5000	The quickest operation response time (in milliseconds) during the last collection cycle.
IONANumInvocations	10000–100000	> 100000	The number of invocations received during the last collection period.
IONAOpsPerHour	1000000–10000000	> 10000000	The throughput (in Operations Per Hour) based on the rate calculated from the last collection cycle.

Monitoring servers

You can use the parameters shown in Table 9 to monitor the load and response times of your Artix servers.

The Default Alarm ranges can be overridden on any particular instance, or on all instances, using the BMC Patrol 7 Central console. You can do this as follows:

1. In the **PATROL Central** console's **Main Map**, right click on the selected parameter and choose the **Properties** menu item.
2. In the **Properties** pane, select the **Customization** tab.
3. In the **Properties** drop-down list, select ranges.
4. You can customize the alarm ranges for this parameter on this instance. If you want to apply the customization to all instances, select the **Override All Instances** checkbox.

Note: The `IONANumInvocations` parameter is a raw, non-normalized metric and can be subject to sampling errors. To minimize this, keep the performance logging period relatively short, compared to the poll time for the parameter collector.

Monitoring operations

In the same way that you can monitor the overall performance of your servers and clients, you can also monitor the performance of individual operations. In Artix, an operation relates to a WSDL operation defined on a port.

In many cases, the most important metrics relate to the execution of particular operations. For example, it could be that the `make_reservation()`, `query_reservation()` calls are the operations that you are particularly interested in measuring. This means updating your `servers.conf` file as follows:

```
mydomain_myserver,1,/var/mydomain/logs/myserver_perf.log,[make_reservation,query_reservation]
```

In this example, the addition of the bold text enables the `make_reservation` and `query_reservation` operations to be tracked by BMC Patrol.

Operation parameters

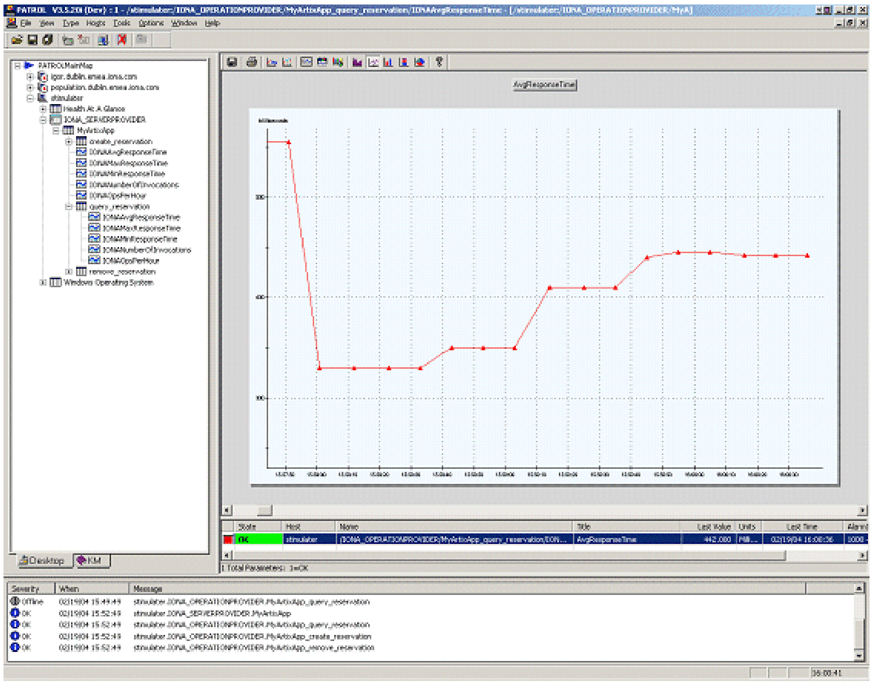
Table 10 shows the Artix parameters that are tracked for each operation instance:

Table 10: *Artix Operation Provider Parameters*

Parameter Name	Default Warning	Default Alarm	Description
IONAAvgResponseTime	1000–5000	> 5000	The average response time (in milliseconds) for this operation on this server during the last collection cycle.
IONAMaxResponseTime	1000–5000	> 5000	The slowest invocation of this operation (in milliseconds) during the last collection cycle.
IONAMinResponseTime	1000–5000	> 5000	The quickest invocation (in milliseconds) during the last collection cycle.
IONANumInvocations	10000–100000	> 100000	The number of invocations of this operation received during the last collection period.
IONAOpsPerHour	1000000–100000000	> 10000000	The number of operations invoked in a one hour period based on the rate calculated from the last collection cycle.

Figure 48 shows BMC Patrol graphing the value of the `IONAAvgResponseTime` parameter on a `query_reservation` operation call.

Figure 48: *Graphing for IONAAvgResponseTime*



	State	Host	Name	Title	Last Value	Units	Last Time	Alarm
🟥	OK	stimulator	/IONA_OPERATIONPROVIDER/MyArtixApp_query_reservation/ION...	AvgResponseTime	442.000	Mill...	02/19/04 16:00:36	1000 -

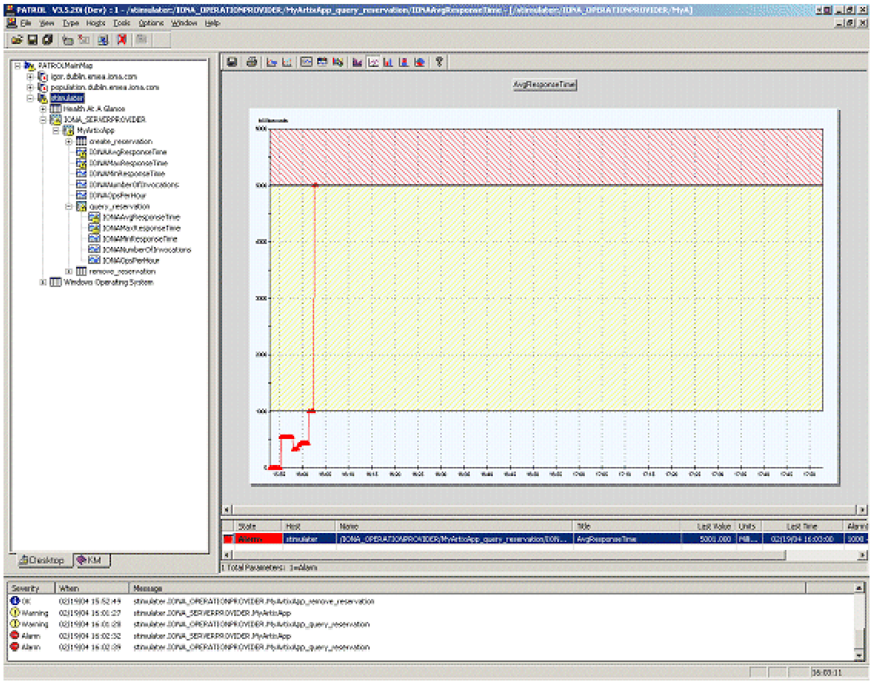
1 Total Parameters: 1=OK

Severity	When	Message
Offline	02/19/04 15:49:49	stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation
OK	02/19/04 15:52:49	stimulator.IONA_SERVERPROVIDER.MyArtixApp
OK	02/19/04 15:52:49	stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation
OK	02/19/04 15:52:49	stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_create_reservation
OK	02/19/04 15:52:49	stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_remove_reservation

16:00:41

Figure 49 shows warnings and alarms issued for the `IONAAvgResponseTime` parameter.

Figure 49: *Alarms for IONAAvgResponseTime*



	State	Host	Name	Title	Last Value	Units	Last Time	Alarm
🟥	Alarm	stimulator	/IONA_OPERATIONPROVIDER/MyArtixApp_query_reservation/ION...	AvgResponseTime	5001.000	Mill...	02/19/04 16:03:00	5000 -

1 Total Parameters: 1=Alarm

Severity	When	Message
OK	02/19/04 15:52:49	stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_remove_reservation
Warning	02/19/04 16:01:27	stimulator.IONA_SERVERPROVIDER.MyArtixApp
Warning	02/19/04 16:01:28	stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation
Alarm	02/19/04 16:02:32	stimulator.IONA_SERVERPROVIDER.MyArtixApp
Alarm	02/19/04 16:02:39	stimulator.IONA_OPERATIONPROVIDER.MyArtixApp_query_reservation

Starting, stopping and restarting servers

The `server_commands.txt` file contains the details about the commands for services that you are deploying on your host (see Chapter 12). To execute commands in this file, perform the following steps:

1. Right click on an instance in the BMC Patrol Console **Main Map**.
2. Select **Knowledge Module Commands|IONA|Commands**.
3. Select one of the following commands:

Start	Starts a server.
Stop	Stops a server.
Restart	Executes a stop followed by a start.

Troubleshooting

If you have difficulty getting the Artix BMC Patrol integration working, you can use the menu commands to cause debug output to be sent to the system output window.

To view the system output window for a particular host, right click on the icon for your selected host in the BMC Patrol **Main Map**, and choose **System Output Window**.

You can change the level of diagnostics for a particular instance by right clicking on that instance and choosing:

Knowledge Module Commands|IONA|Log Levels

You can choose from the following levels:

- **Set to Error**
- **Set to Info**
- **Set to Debug**

Set to Debug provides the highest level of feedback and **Set to Error** provides the lowest.

CHAPTER 14

Extending to a BMC Production Environment

This section describes how to extend an Artix BMC Patrol integration from a test environment to a production environment.

In this chapter

This chapter contains the following sections:

Configuring an Artix Production Environment

Overview

This section describes the steps that you need to take when extending the BMC Patrol integration from an Artix test environment to a production environment. It includes the following sections:

- "Monitoring your Artix applications"
- "Monitoring Artix applications on multiple hosts"
- "Monitoring multiple Artix applications on the same host"

Monitoring your Artix applications

You must add configuration settings to your Artix server configuration files.

For C++ and JAX-RPC applications, add the following example configuration settings to your Artix application's `.cfg` file:

```
// my_app.cfg

my_application {

# Ensure that it_response_time_collector is in your orb_plugins list.
orb_plugins = [ ...,"it_response_time_collector"];

# Enable performance logging.
use_performance_logging = true;

# Collector period (in seconds). How often performance information is logged.
plugins:it_response_time_collector:period = "60";

# Set the name of the file which holds the performance log
plugins:it_response_time_collector:filename =
   "/opt/myapplication/log/myapplication_perf.log"

};
```

Note: The specified `plugins:it_response_time_collector:period` should divide evenly into your cycle time (for example, a `period` of `20` and a cycle time of `60`).

Monitoring Artix applications on multiple hosts

To monitor your Artix applications on multiple hosts, you must distribute the Artix KM to your hosts. The best approach to distributing the Artix Knowledge Module to a large number of machines is to use the Knowledge Module Distribution Service (KMDS).

Using the KMDS to distribute the IONA KM

To create a deployment set for machines that run Patrol Agents (but not the Patrol Console), perform the following steps:

1. Choose a machine with the Patrol Developer Console installed. Follow the procedure for installing the Artix KM on this machine (see “Setting up your BMC Patrol Environment” on page 50).
2. Start the Patrol Developer Console and choose **Edit Package** from the list of menu Items.
3. Open the following file:

```
$PATROL_HOME/archives/IONA_Server_KM_Agent_Resources.pkg file
```

   You will see a list of all the files that need to be installed on machines that run the Patrol Agent.
4. Now select **Check In Package** from the **File** menu to check the package into the KMDS.
5. You can now use the KMDS Manager to create a deployment set based on this KM package, and distribute it to all the machines that Artix installed and that also have a Patrol Agent.
6. You repeat this process for the `IONA_Server_KM_Console_Resources.pkg` file.

This creates a deployment set for all machines that have both the Patrol Agent and Patrol Console installed, and which will be used to monitor Artix applications.

For further details about using the KMDS, see your BMC Patrol documentation.

Monitoring multiple Artix applications on the same host

Sometimes you may need to deploy multiple Artix applications on the same host. The solution is simply to merge the `servers.conf` and `server_commands.txt` files from each of the applications into single `servers.conf` and `server_commands.txt` files.

For example, if the `servers.conf` file from the `UnderwriterCalc` application looks as follows:

```
UnderwriterCalc,1,/opt/myAppUnderwritierCalc/log/UnderwriterCalc_perf.log
```

And the `servers.conf` file for the `ManagePolicy` application looks as follows:

```
ManagePolicy, 1, /opt/ManagePolicyApp/log/ManagePolicy_perf.log
```

The merged `servers.conf` file will then include the following two lines:

```
UnderwriterCalc,1,/opt/myAppUnderwritierCalc/log/UnderwriterCalc_perf.log
ManagePolicy, 1, /opt/ManagePolicyApp/log/ManagePolicy_perf.log
```

You can now copy this merged file to your `$PATROL_HOME/lib/iona/conf` directory and BMC Patrol will monitor both applications.

Exactly the same procedure applies to the `server_commands.txt` file.

Further information

For more detailed information on the BMC Patrol consoles, see your BMC Patrol documentation.

Index