PROGRESS SOFTWARE

Artix® ESB

Management Guide, Java 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 Java applications. It explains how to integrate and manage Artix Java applications with the following:

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

This guide applies to Artix applications written using JAX-WS (Java XML-Based APIs for Web Services) and JavaScript only.

For information on Artix applications written in C++ or JAX-RPC (Java XML-Based APIs for Remote Procedure Call), see the Artix Management Guide, C++ 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.

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 Java runtime's management architecture and features.

Part II

- Chapter 2 introduces the JMX features supported by the Artix Java runtime, and describes the Artix components that can be managed using JMX.
- Chapter 3 explains how to instrument your Artix Java services using custom MBeans.
- Chapter 4 explains how to configure an Artix Java runtime for JMX.
- Chapter 5 explains how to manage and monitor Artix Java services using JMX consoles.

Part III

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

Part IV

- Chapter 9 describes the architecture of the Artix Java 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 Java 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 Java Management

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

In this chapter

This chapter includes the following section:

Introduction to Artix Java Management

Overview

This section introduces the Artix Java management architecture and explains its various components.

Management architecture

The Artix Java management architecture provides:

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

Figure 1 shows a basic overview of the Artix Java management architecture. The Artix Java runtime uses 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

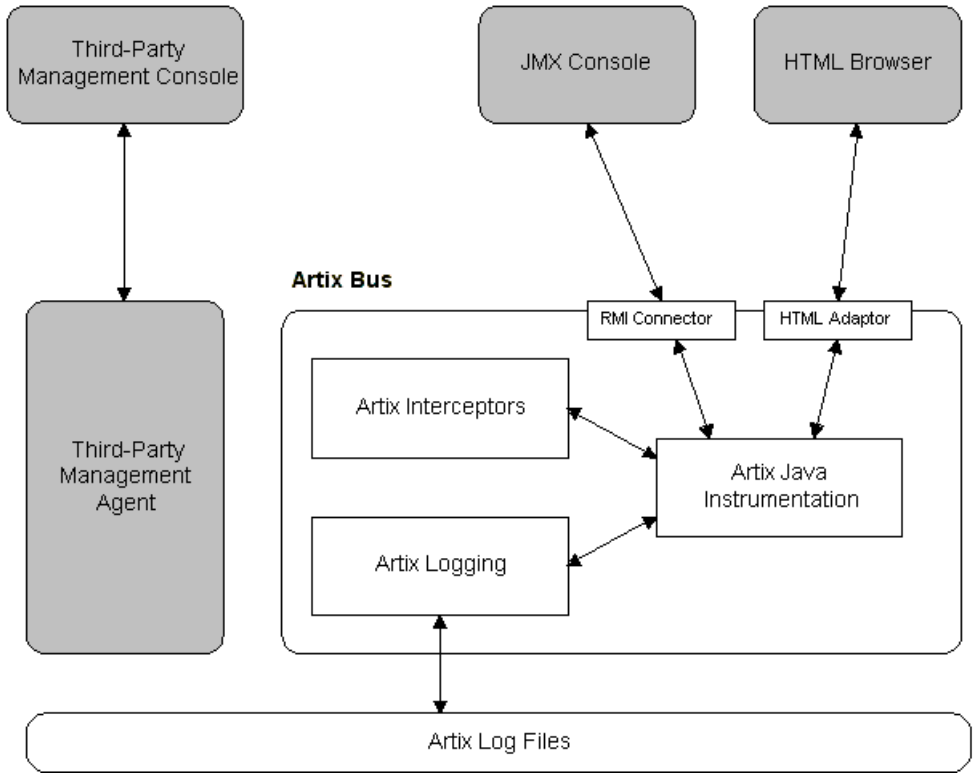
Integrations with third-party enterprise management and SOA management systems are critical to large corporations. Artix provides integration with the Actional and AmberPoint SOA management systems, and the BMC Patrol Enterprise Management System (EMS).

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 Java Management Integrations" on page 19.

Figure 1: *Artix Java 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 2 on page 20).

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 Java service endpoints and Artix Java bus). In addition, the Artix Java 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.

Artix Java logging

The Artix Java runtime uses the standard Java logging utility, `java.util.logging`. For example, this can be used to display logging in a console, or to write to a log file or e-mail. Artix Java logging is configured in a logging configuration file, using the standard `java.util.Properties` format. You can also specify logging programmatically, or define command-line properties that point to the logging configuration file when you start an application.

The Artix Java runtime provides a default `logging.properties` file in the *`ArtixInstallDir`*`/java/etc` directory. This specifies the location for log output messages and the message level published. The default `logging.properties` file configures the Artix Java loggers to print log messages of level `WARNING` to the console. You can use this file without changing any configuration, or you can change the configuration to suit your application.

For more details on Artix Java logging, see Configuring and Deploying Artix Solutions, Java Runtime.

For more details on the `java.util.logging` utility, see: http://java.sun.com/j2se/1.5.0/docs/guide/logging/overview.html

---

Flexible configuration

The Artix Java runtime supports a number of configuration mechanisms to enable you to change the default behavior, enable specific functionality, or fine-tune behavior. The supported configuration mechanisms include XML configuration files, WS-Policy, and WSDL extensions. XML configuration files are the most flexible way to configure the Artix Java runtime and are the recommended approach to use.

The following example shows a simplified Artix Java configuration file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/
   beans http://www.springframework.org/schema/beans/
   spring-beans-2.0.xsd">
     <!-- your configuration goes here! -->
</beans>
```

Artix Java runtime configuration is based on the Spring Framework. This means that an Artix Java configuration file is really a Spring XML file. You must include an opening Spring `beans` element that declares the namespaces and schema files for the child elements that are encapsulated by the `beans` element.

The contents of your configuration depends on the behavior you want the Artix Java runtime to exhibit. You can also use the either of the following XML syntax:

- Plain Spring XML—the child elements of the Spring `beans` element are Spring `beans` elements.
- A simplified beans syntax—the child elements of the Spring `beans` element can be any one of a number of custom namespaces. For example, you can use `jaxws:endpoint` elements.

For a specific example of configuring Artix Java management integration, see "Configuring Artix–Actional Integration" on page 69.

For more details on Artix Java configuration, see Configuring and Deploying Artix Solutions, Java Runtime.

For more details on Spring, see www.springframework.org.

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. This is the JMX-based management console provided with JDK 1.5 to monitor and manage Java applications. For more details, see "JMX" on page 19.

Artix Java 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).

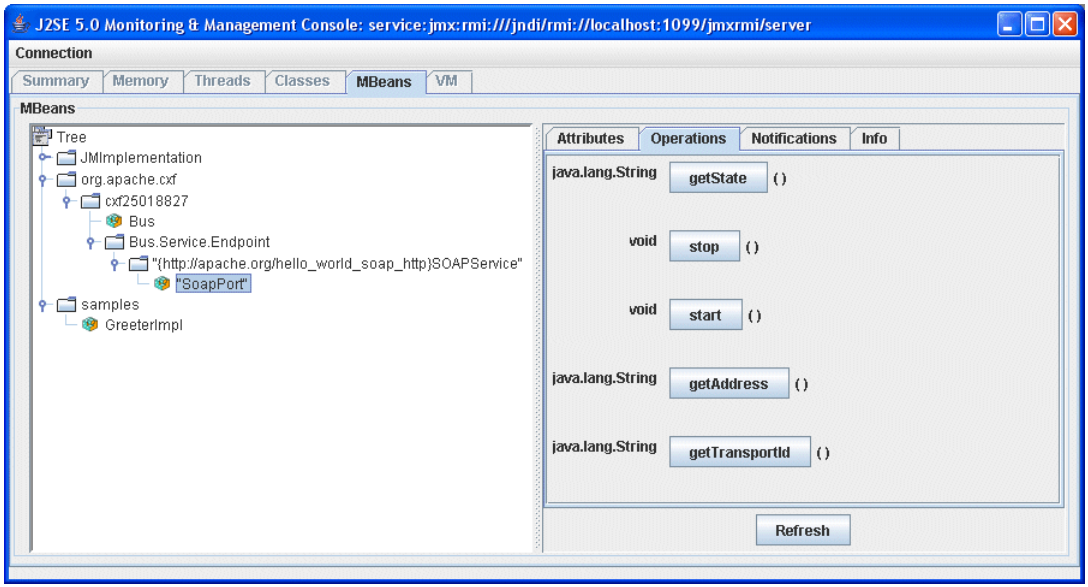
You can use JMX consoles to monitor and manage key Artix Java runtime components both locally and remotely. For example, using any JMX-compliant client, you can perform tasks such as:

- 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 Artix Java service endpoint being monitored in JConsole. For more details on Artix integration with JMX, see Part II.

Figure 2: *Artix Java Service Endpoint in JConsole*



Progress Actional

Integration between Artix and Actional enables Artix services to be monitored by Actional SOA management systems. For example, you can use Actional SOA management tools to perform 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 Artix services and report data back to the Actional SOA management tools:

Actional agents

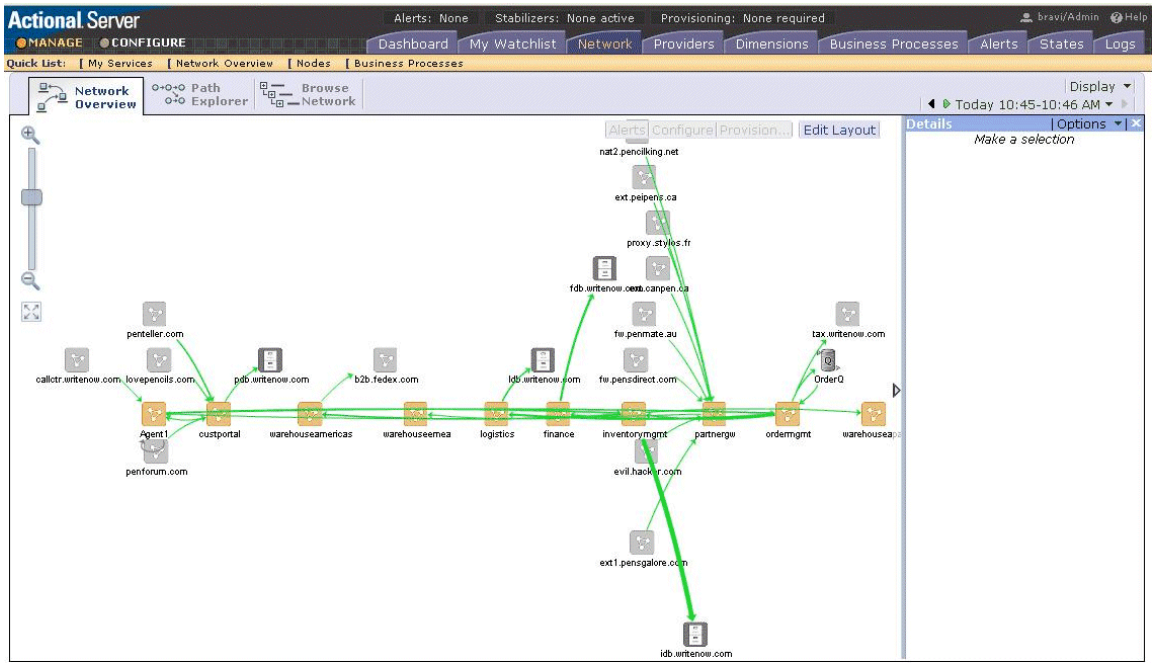
An Actional agent is run on each Artix node that you wish to manage. Actional agents are used to provide instrumentation data back to the Actional server. Actional agents are provisioned from the Actional server to establish initial contact and send configuration to the Actional agent.

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 Artix services to be monitored by the AmberPoint SOA management system. 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 Artix services to be monitored by the BMC Patrol Enterprise Management System (EMS). You can use the Artix integration with BMC Patrol to track key server metrics (for example, 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 Java runtime integration with BMC Patrol obtains server metrics using JMX-based Artix interceptors. 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:

CHAPTER 2

Monitoring and Managing with JMX

This chapter explains how to monitor and manage Artix Java runtime components using Java Management Extensions (JMX).

In this chapter

This chapter discusses the following topics:

Introduction

Overview

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

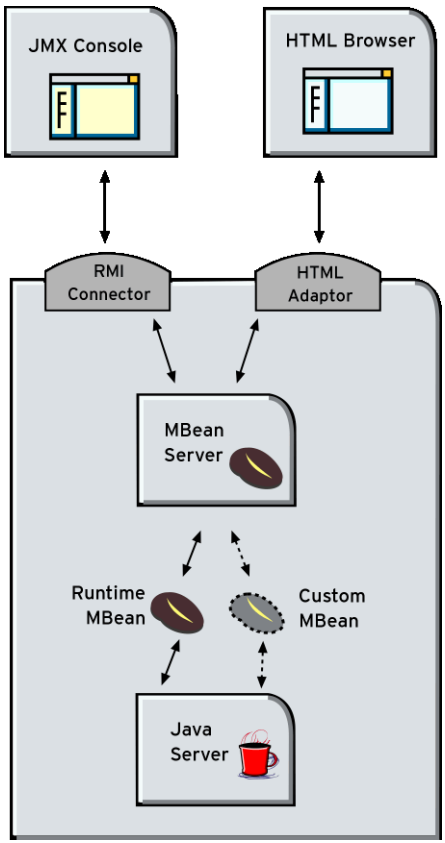
How it works

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.

Artix Java runtime components can be exposed as JMX MBeans out-of-the-box (for example, Artix Java service endpoints and the Artix Java bus). In addition, the Artix Java 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 (see "Relationship between runtime and custom MBeans" on page 28).

Artix Java services can be monitored and managed by any JMX-compliant client application (for example JConsole). Figure 4 shows an overview of how the various components interact.

Figure 4: *Artix Java Runtime JMX Architecture*



The custom MBeans shown in Figure 4 are optional components that can be implemented as required (for details, see Chapter 3).

Enabling JMX

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

For details on how to configure JMX support in Artix applications, see Chapter 4.

What can be managed

Artix JAX-WS servers can have the following runtime components exposed as JMX MBeans:

- Artix Java bus
- Service endpoint

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

Naming conventions

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 `org.apache.cfx` as their domain name when creating managed components.

Relationship between runtime and custom MBeans

The Artix Java runtime instrumentation provides an out-of-the-box JMX view of JAX-WS services. Java developers can also create custom JMX MBeans to manage Artix Java components such as service endpoint attributes and operations.

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, 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 more details, see Chapter 3.

Further information

For further information, see the following:

JMX

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

JMX Remote

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

Dynamic MBeans

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

MBeanServer

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

Managed Runtime Components

Overview

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

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

Artix bus operations

The following Artix Java bus method can be accessed using any JMX console or client:

Table 1: *Managed Bus Methods*

Name	Description	Parameters
`shutdown()`	Shuts down the current Artix Java bus.	`boolean`

Endpoint attributes

The following Artix service endpoint attributes can be managed by any JMX console or client:

Table 2: *Managed Endpoint Attributes*

Name	Description	Type
`Address`	Endpoint address (for example, `http://localhost:9000/SoapContext/SoapPort`).	`String`
`State`	Current service state manipulated by stop and start methods. Possible values are `STARTED` or `STOPPED`.	`String`
`TransportID`	Endpoint transport ID (for example, `http://schemas.xmlsoap.org/soap/http` for the HTTP transport).	`String`

Endpoint operations

The following Artix service endpoint operations can be managed by any JMX console or client:

Table 3: *Managed Endpoint Operations*

Name	Description	Parameters	Return Type
`getAddress()`	Get the service address (for example, `http://localhost:9000/SoapContext/SoapPort`).	None	`String`
`getState()`	Get the current service state. Possible values are `STARTED` or `STOPPED`.	None	`String`
`getTransportID()`	Get the service transport ID (for example, `http://localhost:9000/SoapContext/SoapPort`).	None	`String`
`start()`	Activate a service.	None	`Void`
`stop()`	Deactivate a service.	None	`Void`

For examples of operations and attributes displayed in a JMX console, see Chapter 5

MBeanInfo

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

CHAPTER 3

Instrumenting Artix Java Services

This chapter explains how to instrument your Artix Java services using custom MBeans. There are two different approaches. You can use either the JMX MBean interfaces or the Artix ManagedComponent interface. This applies to applications written using the Java API for XML-Based Web Services (JAX-WS).

In this chapter

This chapter discusses the following topics:

Using the JMX MBean Interfaces

Overview

This section shows how to implement a JMX MBean interface and register it with the Artix MBean server.

The Artix MBean server can be accessed through the Artix Java bus and enables the registration of custom MBeans. You can instrument your service implementation by developing a custom MBean using one of the JMX MBean interfaces and registering it with the Artix MBean server. Your custom instrumentation can be accessed using the same JMX connection as the Artix internal components used by your service.

Creating your custom MBean

When using the JMX APIs to instrument your service implementation, follow the design methodology laid out by the JMX specification. This involves the following steps:

1. Decide what type of MBean you wish to use.
   - Standard MBeans expose a management interface defined at development time.
   - Dynamic MBeans expose their management interface at runtime.
2. Create the MBean interface to expose the properties and operations used to manager your service implementation.
   - Standard MBeans use the `MBean` interface.
   - Dynamic MBeans use the `DynamicMBean` interface.
3. Implement the MBean class.

Example 1 shows the interface for a standard MBean.

Example 1: *Standard MBean Interface*

```
public interface ServerNameMBean
{
   String getServiceName();
   String getAddress();
}
```

Example 2 shows a class that implements the MBean defined in Example 2.

Example 2: *Standard MBean Implementation Class*

```
public class ServerName{

   String getServiceName()
   {
      return "mySOAPservice";
   }

   String getAddress()
   {
      return "myServiceAddress";
   }
}
```

Registering the MBean

To expose your MBean in a JMX management console, it must be registered with the Artix MBean server. The Artix MBean server can be accessed through the Artix Java bus. Typically, this happens when your service is initialized.

To register a custom MBean, perform the following steps:

1. Instantiate your custom MBean.
2. Get an instance of the bus.
3. Get the Artix MBean server from the bus.
4. Create an `ObjectName` for your MBean.

Note: It is recommended that your MBeans follow the "Naming conventions" on page 28. However, you can choose any naming scheme.

5. Register your MBean server using the server's `registerMBean()` method.

Example 3 shows the steps for registering a custom MBean with the Artix MBean server.

Example 3: *Registering a Custom MBean*

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import org.apache.cfx.Bus;

...

//Instantiate the MBean
ServerName sName = new ServerName();

//Get the MBean server from the bus
InstrumentationManager im =
   BusFactory.getDefaultBus().getExtension(org.apache.cxf.manage
   ment.InstrumentationManager.class);
MBeanServer server = im.getMBeanServer();

//Create ObjectName for the MBean
ObjectName name = new
   ObjectName(my..instrumentation:type=CustomMBean,Bus="+
   bus.getBusID() + name="ServerNameMBean");

//Register the MBean
server.registerMBean(sName, name);
```

Alternatively, you do not have to register the MBean directly with the `MBeanServer`. You can also use the `register(Object, ObjectName)` utility method on the `InstrumentationManager` to register a `StandardMBean`.

Further information

For further information, see the following:

ObjectName

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

MBeanServer

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

JMX specifications

http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/docs.jsp.

Using the Artix ManagedComponent Interface

Overview

If you do not wish to use the JMX APIs to add instrumentation to your service, you can use the Artix `ManagedComponent` interface. This interface wraps the JMX subsystem in an Artix-specific API. You do not need to access the Artix MBean server to register your managed components because the Artix wrappers take care of this for you. This approach uses JMX annotations to specify which methods and attributes are exposed.

Adding custom instrumentation using the Artix `ManagedComponent` interface involves the following steps:

1. Write an instrumentation class that implements the `org.apache.cxf.management.ManagedComponent` interface.
2. When your service is starting up, activate your `ManagedComponent` object by instantiating it and registering it with the `InstrumentationManager`.
3. When your service is shutting down, deactivate your `ManagedComponent` by unregistering it and cleaning it up.

This section shows how to implement the Artix `ManagedComponent` interface using JMX annotations. It uses example code from the Artix `jmx` sample application:

```
ArtixInstallDir\java\samples\management\jmx
```

Writing the instrumentation class

Like an MBean, an Artix instrumentation class is responsible for providing access to the attributes that you wish to track, and implementing any management operations that you want to expose. Unlike an MBean, an Artix instrumentation class does not implement a user-defined interface. Instead, it implements the Artix-defined `ManagedComponent` interface, and defines the operations required to expose your managed attributes and operations.

JMX annotations

The Artix management API uses JDK 5.0 annotations to create an object that implements the `ModelMBeanInfo` interface. This reads the Artix annotations to identify the attributes and operations that are exposed. It then uses this information to create a `ModelMBean` and registers it with the MBean server.

Table 4 lists the JDK 5.0 annotations that can be used when implementing your instrumentation class.

Table 4: *JDK 5.0 JMX Annotations*

Annotation	Type	Description
`@ManagedResource`	Class	Marks all instances of a class as a JMX managed resource.
`@ManagedOperation`	Method	Marks a method as a JMX operation.
`@ManagedAttribute`	Method	Marks a getter or a setter as one half of a JMX attribute.
`@ManagedNotification`	Method	Marks a JMX notification issued by an MBean.
`@ManagedOperationParameter` `@ManagedOperationParameters`	Method	Describes the parameters of a managed operation.

Annotation parameters

Table 5 lists parameters that can be supplied to the JMX annotations.

Table 5: *JMX Annotation Metadata*

Parameter	Annotation	Description
`componentName`	`@ManagedResource`	Specifies the name of the managed resource.

`description`	`@ManagedResource`
	`@ManagedAttribute`
	`@ManagedOperation`
	`ManagedNotification`
	`@ManagedOperationParameter`
		Specifies a user-friendly description of the resource, attribute, or operation.

Table 5: *JMX Annotation Metadata*

Parameter	Annotation	Description
currencyTimeLimit	@ManagedResource @ManagedAttribute	Specifies how long in seconds a cached value is valid (0 for never, =0 for always, or >0).
defaultValue	@ManagedAttribute	Specifies a default cached value.
log	@ManagedResource @ManagedNotification	Enables logging. Specify true to log all notifications or false to log no notifications.
logFile	@ManagedResource @ManagedNotification	Specifies the a fully qualified filename to log events to.
persistPolicy	@ManagedResource	Specifies the persistence policy. Values are: OnUpdate OnTimer NoMoreOftenThan Always Never
persistPeriod	@ManagedResource	Specifies the frequency of the persist cycle in seconds for the OnTime and NoMoreOftenThan policies.
persistLocation	@ManagedResource	Specifies the filename in which the MBean should be persisted.
persistName	@ManagedResource	Specifies the name that is persisted.
name	@ManagedOperationParameter	Specifies the display name of an operation parameter.
index	@ManagedOperationParameter	Specifies the index of an operation parameter.

Adding annotations

When implementing your custom MBean's instrumentation class using the `ManagedComponent` interface, you should annotate the class with the `@ManagedResource` attribute. Any management operation that you wish to expose in your class should be annotated with the `@ManagedOperation` attribute. Any attributes that you wish to expose should have their getter and setter methods annotated with the `@ManagedAttribute` attribute. If you want to make an attribute read-only or write-only, you can omit the annotation from either its setter method or its getter method.

Example 4 shows an custom MBean class taken from the Artix `jmx` sample application.

Example 4: *Example Artix Instrumentation Class*

```
// GreaterImpl.java

package demo.hw.server;

import java.util.logging.Logger;

import javax.management.JMException;
import javax.management.ObjectName;

import org.apache.cxf.management.ManagedComponent;
import org.apache.cxf.management.annotation.ManagedAttribute;
import org.apache.cxf.management.annotation.ManagedOperation;
import org.apache.cxf.management.annotation.ManagedResource;

import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.types.FaultDetail;

@ManagedResource(componentName = "GreeterImpl",
                 description = "A typical Greeter implementation.")

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")
```

```
public class GreeterImpl implements Greeter, ManagedComponent {

    private static final Logger LOG = Logger.getLogger(GreeterImpl.class.getPackage().getName());
    private int count;

    /* (non-Javadoc)
     * @see org.apache.hello_world_soap_http.Greeter#greetMe(java.lang.String)
     */
    public String greetMe(String me) {
        LOG.info("Executing operation greetMe");
        count++;
        System.out.println("Executing operation greetMe");
        System.out.println("Message received: " + me + "\n");
        return "Hello " + me;
    }

/* (non-Javadoc)
     * @see org.apache.hello_world_soap_http.Greeter#greetMeOneWay(java.lang.String)
     */
    public void greetMeOneWay(String me) {
        LOG.info("Executing operation greetMeOneWay");
        count++;
        System.out.println("Executing operation greetMeOneWay\n");
        System.out.println("Hello there " + me);
    }

 /* (non-Javadoc)
     * @see org.apache.hello_world_soap_http.Greeter#sayHi()
     */
    public String sayHi() {
        LOG.info("Executing operation sayHi");
        count++;
        System.out.println("Executing operation sayHi\n");
        return "Bonjour";
    }

    public void pingMe() throws PingMeFault {
        count++;
        FaultDetail faultDetail = new FaultDetail();
        faultDetail.setMajor((short)2);
        faultDetail.setMinor((short)1);
        LOG.info("Executing operation pingMe, throwing PingMeFault exception");
        System.out.println("Executing operation pingMe, throwing PingMeFault exception\n");
        throw new PingMeFault("PingMeFault raised by server", faultDetail);
    }
```

```
    public ObjectName getObjectName() throws JMException {
        return new ObjectName("samples:name=GreeterImpl");
    }

    @ManagedAttribute(description = "The Count Attribute", currencyTimeLimit = 15)
    public int getCount() {
        return count;
    }

    @ManagedOperation(description = "Add Two Numbers Together")
    public void resetCount() {
        count = 0;
    }
}
```

Registering your custom MBean

To make your custom instrumentation available to JMX management consoles, you must create an instance of your custom class and register it with the bus. This handles the creation of the `ModelMBean` to represent your custom MBean. It also handles the registration of the MBean with the MBean server.

To activate your custom MBean, do the following:

1. Get the current Artix Java bus instance.
2. Get the `InstrumentationManager` from the bus using `bus.getExtension()`.
3. Create an instance of your instrumentation class.
4. Register your custom MBean instance with the `InstrumentationManager`.

Example 5 shows these steps in the sample server code for activating a custom MBean.

Example 5: *Example Server Code*

```
// Server.java

package demo.hw.server;

import javax.xml.ws.Endpoint;
import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.cxf.management.InstrumentationManager;
import org.apache.cxf.management.ManagedComponent;

public class Server {

    protected Server() throws Exception {
        System.out.println("Starting Server");

        SpringBusFactory factory = new SpringBusFactory();
        Bus bus = factory.createBus();
        InstrumentationManager im = bus.getExtension(InstrumentationManager.class);
        ManagedComponent component = new GreeterImpl();
        im.register(component);
        String address = "http://localhost:9000/SoapContext/SoapPort";
        Endpoint.publish(address, component);
    }

    public static void main(String args[]) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(10 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

Deactivating your custom instrumentation

You can explicitly tell the bus to remove the `ModelMBean` created for your instrumentation using the `InstrumentationManager.unregister()` method. This method removes the MBean from the Artix MBean server, destroys the associated `ModelMBean`, and frees up any resources used by it.

In the Artix `jmx` sample application MBean is not explicitly unregistered, but is destroyed when the server process is destroyed.

Further information

For further information, see the following:

ModelMBeanInfo

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

CHAPTER 4

Configuring JMX in Artix Java

This chapter explains how to configure an Artix Java runtime for JMX using the XML-based Spring Framework.

In this chapter

This chapter discusses the following topic:

Artix JMX Configuration

Overview

JMX support in an Artix Java runtime is enabled using configuration settings only. You do not need to write any Artix code to enable JMX support in the Artix runtime.

JMX is not configured by default. When configured, you can use any third party console that supports JMX Remote to monitor and manage Artix services. This section shows the Artix configuration settings required to enable JMX monitoring of the Artix runtime, and access for remote JMX clients.

Configuring Artix JMX features

The Artix Java configuration mechanism uses the XML-based Spring Framework. In the Artix `jmx` sample application, the JMX support is configured using the `org.apache.cxf.management.jmx.InstrumentationManagerImpl` class. This class includes the following properties:

`bus`	Specifies the name of the Artix bus. The name of the Artix Java bus is `cfx`.
`enabled`	Specifies whether JMX monitoring and management is enabled. Possible values are `true` or `false`. Specifying `true` enables remote JMX clients to access runtime and custom MBeans.
`JMXServiceURL`	Specifies the connection URL for the JMX server. The default URL is: `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server`

Example 6 shows the JMX configuration settings taken from the `managed_server.xml` file in the Artix `jmx` sample application:

```
ArtixInstallDir\java\samples\management\jmx
```

Example 6: *Contents of managed_server.xml*

```
?xml version="1.0" encoding="UTF-8"?>
<!-- -->
<!-- Copyright (c) 1993-2007 IONA Technologies PLC. -->
<!--            All Rights Reserved. -->
<!-- -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:im="http://cxf.apache.org/management"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- InstrumetationManager's setting -->
    <bean id="InstrumentationManager"
   class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
        <property name="bus" ref="cxf" />
        <property name="enabled" value="true" />
        <property name="JMXServiceURL"
   value="service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server"/>
    </bean>
</beans>
```

The `InstrumentationManagerImpl` class extends `JMXConnectorPolicyType`. For more details, see "Artix JMX schema" on page 48.

Accessing Artix Java configuration

You can make your configuration available to the Artix Java runtime in one of the following ways:

- Use one of the following command-line flags to point to your XML configuration file:
  - `-Dcxf.config.file=<myCfgResource>`
  - `-Dcxf.config.file.url=<myCfgURL>`

  This enables you to save your XML configuration file anywhere on your system and avoid adding it to your `CLASSPATH`. This is the approach used in most of the Artix Java samples (for example, `managed_server.xml`).

- Specify the XML configuration file on your `CLASSPATH`.
- Programmatically, by creating a bus and passing the configuration file location as either a URL or string, as follows:

```
(new SpringBusFactory()).createBus(URL myCfgURL)
(new SpringBusFactory()).createBus(String myCfgResource)
```

Artix JMX schema

The complete schema for configuring JMX in an Artix Java runtime is contained in the `instrumentation.xsd` file shown in Example 7:

Example 7: *JMX Configuration Schema*

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema targetNamespace="http://cxf.apache.org/management"
    xmlns:tns="http://cxf.apache.org/management"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    jaxb:version="2.0">

    <xs:complexType name="JMXConnectorPolicyType">
        <xs:attribute name="Enabled" type="xs:boolean" use="required" />
        <xs:attribute name="Threaded" type="xs:boolean" use="required" />
        <xs:attribute name="Daemon" type="xs:boolean" use="required" />
        <xs:attribute name="JMXServiceURL" type="xs:string"
   default="service:jmx:rmi:///jndi/rmi://localhost:9913/jmxrmi"/>
    </xs:complexType>

    <xs:element name="JMXConnectorPolicy"
   type="tns:JMXConnectorPolicyType"/>

</xs:schema>
```

These attributes are explained as follows:

`Enabled`	Specifies whether the JMX infrastructure is available to the Artix Java runtime. JMX is disabled by default. The MBean server is an unnecessary overhead if you do not require JMX.
`Threaded`	Specifies whether the JMX server starts in a separate thread.

`Daemon`	If the JMX server is running in a separate thread, specifies whether it is run as daemon thread
`JMXServiceURL`	Specifies the `JMXServiceURL` to connect to remotely. Remote access is performed using JMX Remote, using an RMI Connector on a default port of `1099`. Use the following JNDI-based `JMXServiceURL` to connect remotely: `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server`

Further information

For more information, see the following:

Artix Java configuration

- Configuring and Deploying Artix Solutions, Java Runtime
- Artix Configuration Reference, Java Runtime

Spring Framework

www.springframework.org

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

CHAPTER 5

Managing Java Services with JMX Consoles

You can use any third-party management console that supports JMX Remote to monitor and manage Artix services (for example, JConsole or MC4J).

In this chapter

This chapter discusses the following topics:

Managing Artix Services with JConsole

Overview

The JConsole management console is provided with JDK 1.5 to monitor and manage Artix Java applications. For convenience, Artix installs JConsole, which can be run out-of-the-box with the Artix `jmx` sample application:

```
ArtixInstallDir\java\samples\management\jmx
```

Using JConsole with Artix

Artix runtime MBeans can be accessed remotely using JMXRemote. This means that any management console that supports JMXRemote can be used to monitor and manage Artix-enabled applications.

To view the management information for a deployed Artix-enabled application using JConsole, perform the following steps:

1. Launch the JConsole application using the following command:

   ```
   ArtixInstallDir/java/bin/jmx_console_start
   ```

   Alternatively, you can use:

   ```
   JDK_HOME/bin/jconsole
   ```

2. Select the **Advanced** tab.
3. Enter the URL of your Artix MBean server in the `JMXServiceURL` field. This will either be the default Artix `JMXServiceURL` (`service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server`), or the value specified by the `JMXServiceURL` property in your application's Spring configuration file.

Note: When running the `jmx` sample application, steps 2 and 3 are not necessary.

Managing runtime components

JConsole displays managed Artix runtime components in a hierarchical tree, as shown in Figure 5. This shows the MBean information displayed for the managed bus component (for example, the MBean name and Java class).

Figure 5: *Managed Bus Info in JConsole*

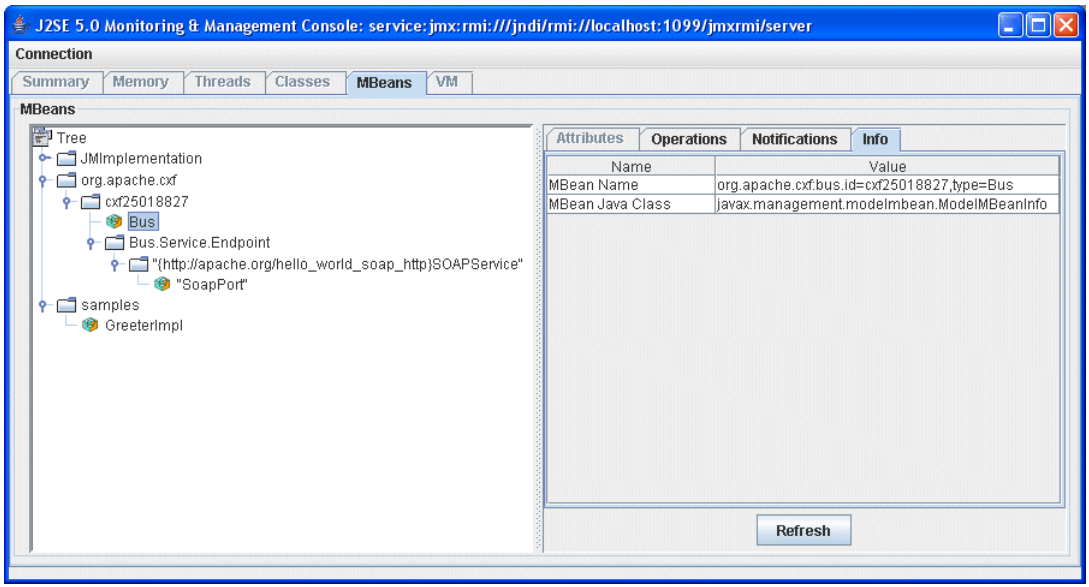


Figure 6 shows the `shutdown()` operation for the managed bus displayed in JConsole.

Figure 6: *Managed Bus Operation in JConsole*

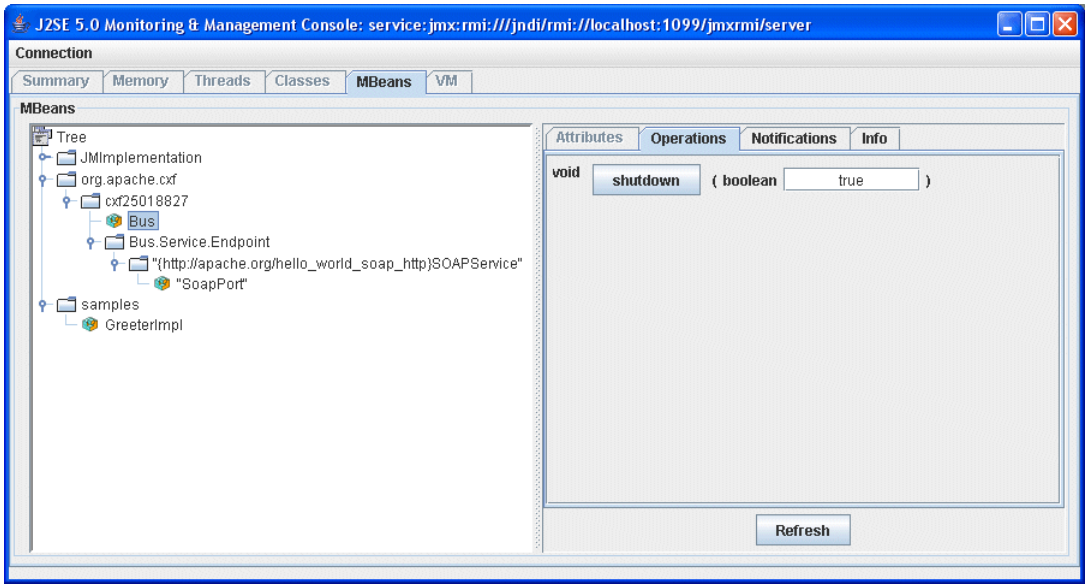


Figure 7 shows the attributes displayed for a managed service endpoint displayed in JConsole.

Figure 7: *Managed Endpoint Attributes in JConsole*

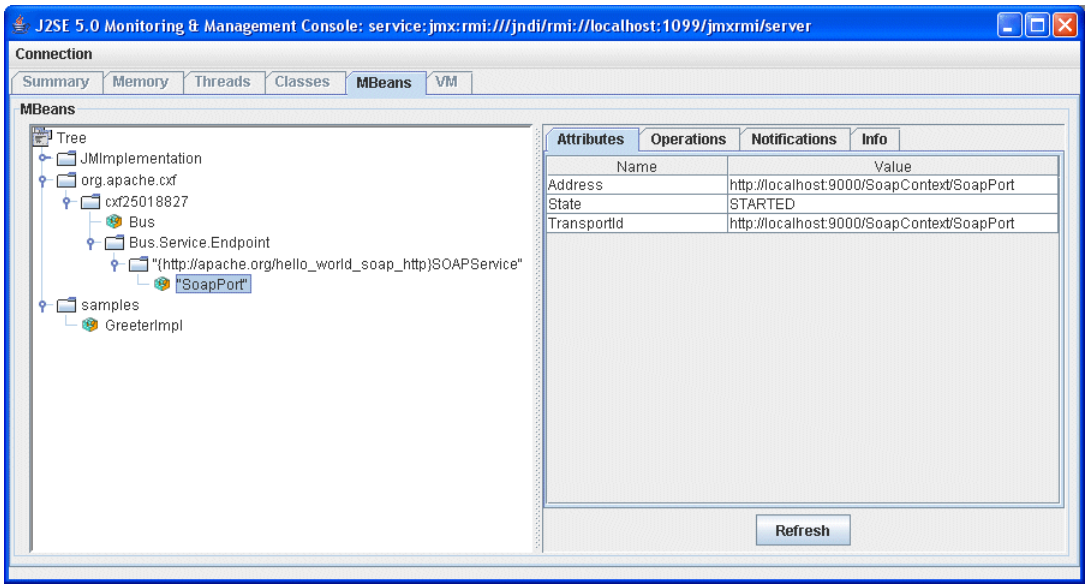
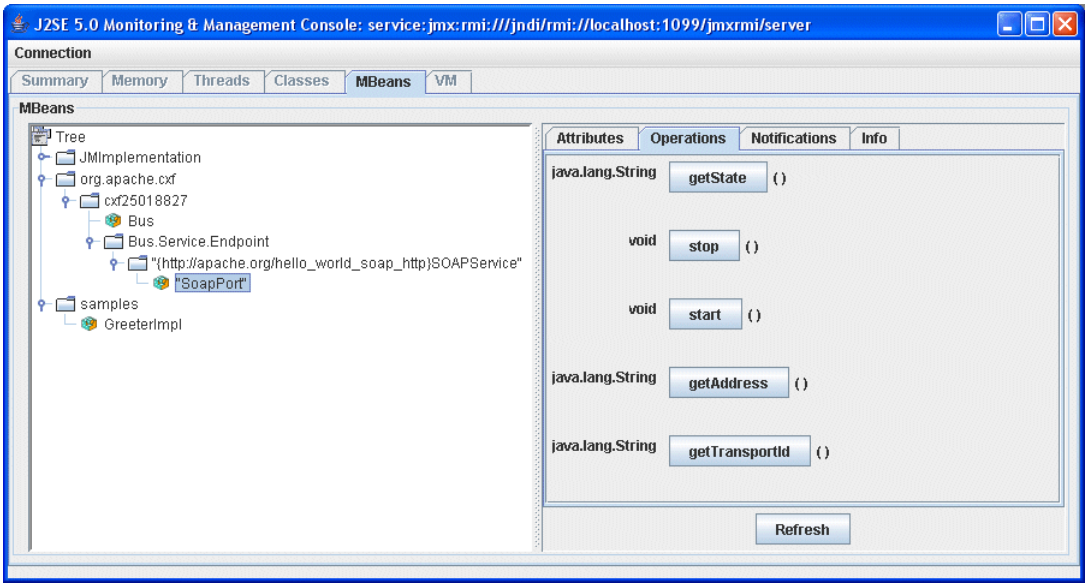


Figure 8 shows the operations displayed for managed a service endpoint displayed in JConsole.

Figure 8: *Managed Endpoint Operations in JConsole*



Managing custom MBeans

Figure 9 shows the attributes displayed for the sample custom MBean displayed in JConsole.

Figure 9: *Custom MBean Attributes in JConsole*

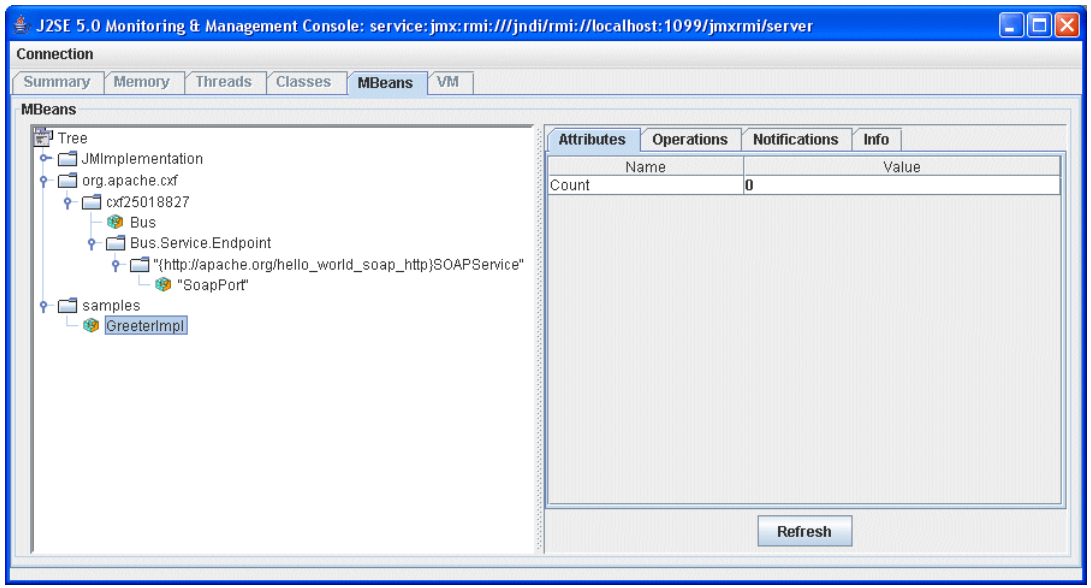
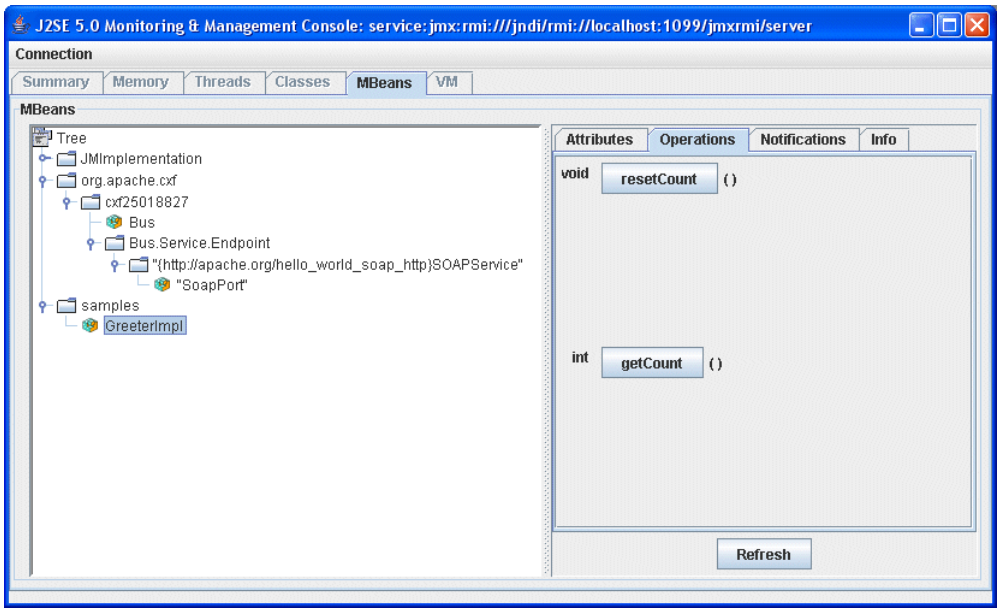


Figure 8 shows the operations displayed for the sample custom MBean displayed in JConsole.

Figure 10: *Custom MBean Operations in JConsole*



Further information

For detailed information on Artix runtime attributes and operations see "Managed Runtime Components" on page 30.

For more information on using JConsole, see the following:

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

Part III

Progress Actional

In this part

This part contains the following chapters:

CHAPTER 6

Integrating with Progress Actional™

Artix provides support for integration with Progress Actional SOA management products. This chapter provides an overview of the integration architecture

In this chapter

This chapter includes the following section:

Artix–Actional Integration 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 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 systems to enable reporting of management data back to the Actional server. The data reported back to Actional includes the following system administration metrics:

- response time
- fault location
- auditing
- alerts based on policies and rules

The Artix–Actional integration can be used with Artix Web service applications implemented in JAX-WS, JavaScript, and JAX-RPC. This guide explains how to integrate JAX-WS and JavaScript applications. For details on integrating JAX-RPC applications, see the Artix Management Guide, C++ Runtime.

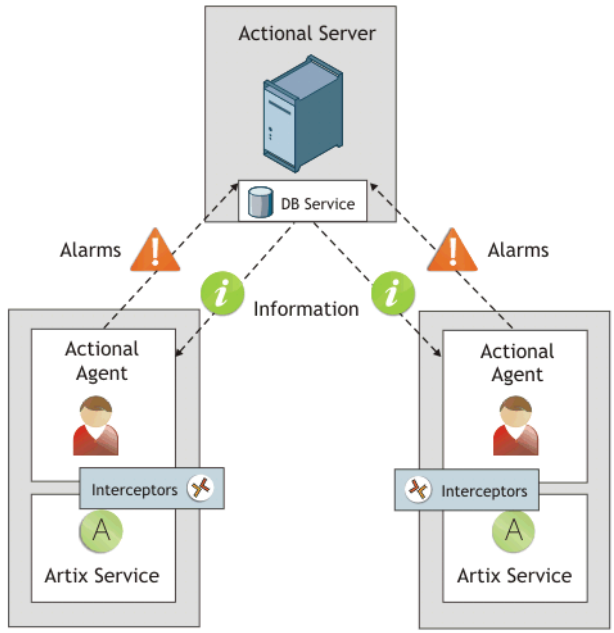
Integration architecture

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

Figure 11: *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 console is a Web application deployed on Apache Tomcat. It has a runtime management mode and agent configuration mode (for example, for setting up policies). 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

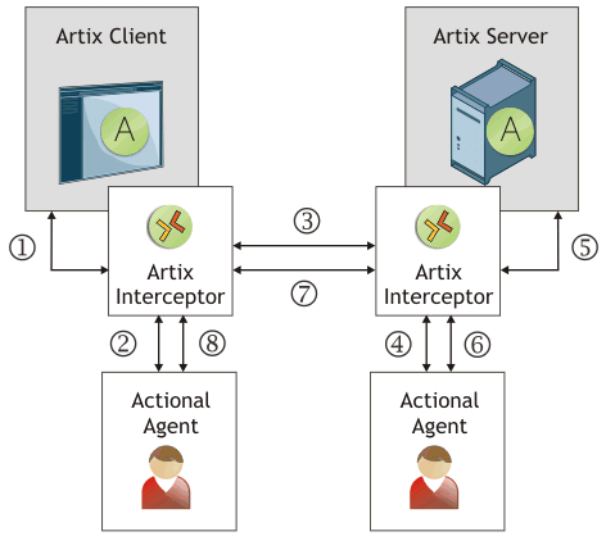
At the level of a managed node, Artix interceptors send the instrumentation data to the Actional agent using an Actional-specific API. These interceptors essentially push events to the Actional agent.

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.

Figure 12 shows the flow of information at the Artix interceptor points:

1. The outbound client interceptor is invoked, which starts a client interaction, and records the outgoing message. All other management data, such as the service, port, and operation names are stored. The correlation ID used to track the message is assigned in the transport, and the client interaction is marked as analyzed.
2. All the management data is sent to the local client-side Actional agent.
3. The outbound client interceptor sends the management data to the server interceptor.

Figure 12: *Artix–Actional Interception Points*



4. The inbound server interceptor receives the request, starts a server interaction, and the correlation ID is fetched from the transport. All management data is set on the server side, and the interaction is marked as analyzed. For one-way calls, the interaction is marked as ended.
5. This shows the interaction with the Artix server.
6. All the management data is transmitted to the local server-side Actional agent.
7. The outbound server interceptor point mimics the outbound client interceptor, and sends the management data to the client interceptor.
8. The inbound client interceptor mimics the inbound server interceptor.

Actional agent interceptor API

The Actional Agent Interceptor SDK is an Actional-specific API used to send the management instrumentation data from the 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 using JAX-WS, or a scripting language such as JavaScript. 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.

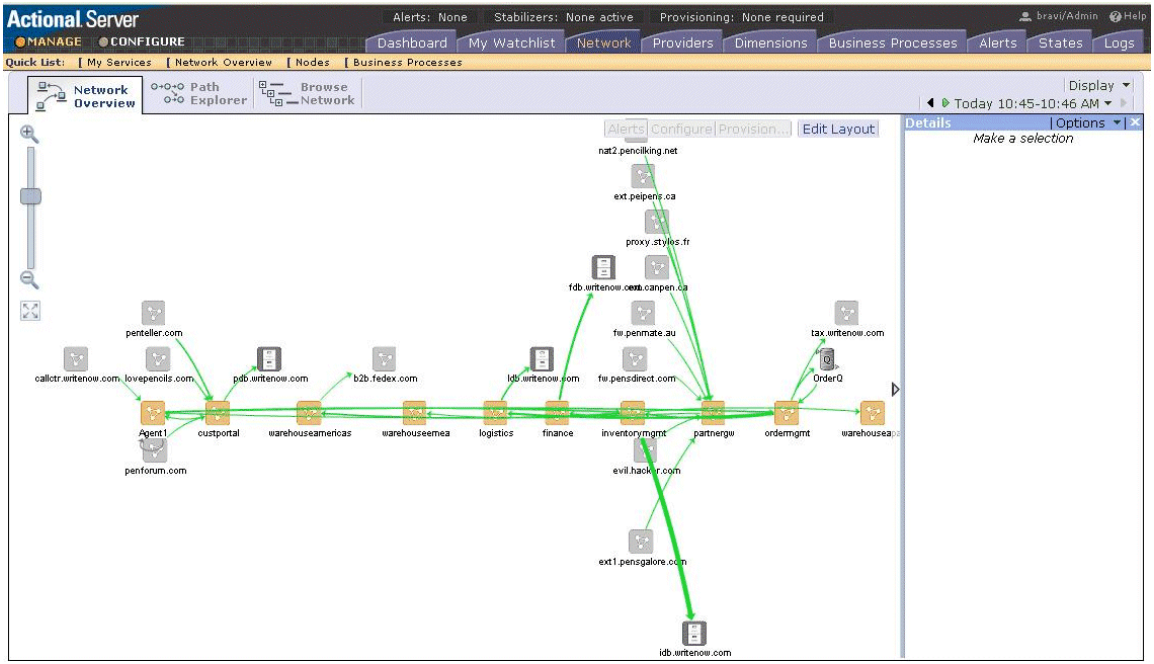
Figure 13 shows an example of the **Actional Server Administration Console**. Managed nodes are displayed as orange boxes, and unmanaged nodes are displayed as grey boxes. The green arrow indicates the message flow through various nodes.

Clicking on each of the nodes shows more in-depth information regarding the response time, alarms 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*.

Further information

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

Figure 13: *Actional Server Administration Console*



CHAPTER 7

Configuring Artix–Actional Integration

This chapter explains how to configure integration between Artix and Actional SOA management products. It shows examples from an Artix–Actional integration demo.

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
- XML over HTTP
- XML over JMS
- CORBA

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

Further information

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

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 samples. For information on how to run these samples, see the `readme.txt` files in the following directories:

```
ArtixInstallDir/java/samples/management/actional/corba
ArtixInstallDir/java/samples/management/actional/jms_queues
ArtixInstallDir/java/samples/management/actional/router
```

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 samples, the Actional agent can be started with the default configuration settings.

Actional server configuration

To set up the Actional server to run an Artix-Actional sample:

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 sample application, 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 Artix 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 services written using JAX-WS for integration with Actional. It shows examples of Artix XML configuration from the Artix–Actional `jms_queues` integration sample. For information on how to run the sample, see the `readme.txt` file in the following directory:

```
ArtixInstallDir/java/samples/management/actional/jms_queues
```

This demo is based on `.../java/samples/transports/jms_queues/`, with some modifications to illustrate Artix and Actional integration, and with two JMS queues instead of one.

Service endpoint configuration

The Artix Java configuration mechanism is based on the XML-based Spring Framework. The following example from the `server1.xml` file in the Artix `jms_queues` sample shows the XML configuration used by the Artix service endpoint:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:soap="http://cxf.apache.org/bindings/soap"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://cxf.apache.org/bindings/soap
                           http://cxf.apache.org/schema/bindings/soap.xsd
                          http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd"
       xmlns:mgmt="http://www.iona.com/management/cxf/">
```

```
  <bean name="{http://apache.org/hello_world_soap_http}SOAPService" abstract="true">
    <property name="properties">
      <map>
        <entry key="schema-validation-enabled" value="true" />
      </map>
    </property>
  </bean>

  <jaxws:endpoint name="{http://cxf.apache.org/jms_greeter}GreeterPort1"
   createdFromAPI="true">
      <jaxws:properties>
         <entry key="schema-validation-enabled" value="true" />
      </jaxws:properties>
      <jaxws:features>
          <mgmt:management capturePayload="true" />
      </jaxws:features>
  </jaxws:endpoint>

</beans>
```

This example shows how an Artix service endpoint named `GreeterPort1` is configured for Artix and Actional integration using the `jaxws:features` element. The `mgmt:management capturePayload` attribute must be set to `true` to enable Artix and Actional integration.

Service consumer configuration

The following example from the `client.xml` file in the Artix `jms_queues` sample shows the client-side configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http="http://cxf.apache.org/transports/http/configuration"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:soap="http://cxf.apache.org/bindings/soap"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                           http://cxf.apache.org/schemas/configuration/http-conf.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://cxf.apache.org/bindings/soap
                           http://cxf.apache.org/schema/bindings/soap.xsd
                           http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd"
       xmlns:mgmt="http://www.iona.com/management/cxf/">

    <http:conduit name="{http://apache.org/hello_world_soap_http}SoapPort9001.http-conduit">
         <http:client DecoupledEndpoint="http://localhost:9990/decoupled_endpoint"/>
    </http:conduit>

    <bean name="{http://apache.org/hello_world_soap_http}SOAPService" abstract="true">
        <property name="properties">
           <map>
             <entry key="schema-validation-enabled" value="true" />
           </map>
        </property>
    </bean>

    <jaxws:client name="{http://cxf.apache.org/jms_greeter}GreeterPort1" createdFromAPI="true">
        <jaxws:features>
            <mgmt:management capturePayload="true" />
        </jaxws:features>
   </jaxws:client>

    <jaxws:client name="{http://cxf.apache.org/jms_greeter}GreeterPort2" createdFromAPI="true">
        <jaxws:features>
            <mgmt:management capturePayload="true" />
        </jaxws:features>
   </jaxws:client>
</beans>
```

Like on the server-side, the `mgmt:management capturePayload` attribute must be set to `true` to enable Artix and Actional integration. This server and client side configuration enables the appropriate interceptors to be loaded into the Artix Java runtime to transmit the monitoring information to the Actional agent.

Accessing Artix Java configuration

You can make your Artix Java configuration available to the Artix Java runtime in one of the following ways:

- Specify the XML configuration file on your `CLASSPATH`.
- Programmatically, by creating a bus and passing the configuration file location as either a URL or string, as follows:

```
(new SpringBusFactory()).createBus(URL myCfgURL)
(new SpringBusFactory()).createBus(String myCfgResource)
```

- Use one of the following command-line arguments to point to your XML configuration file:

```
-Dcxf.config.file.url=<myCfgURL>
-Dcxf.config.file=<myCfgResource>
```

  This enables you to save your XML configuration file anywhere on your system and avoid adding it to your `CLASSPATH`.

Example commands

The following example command is used to start a server:

```
start java
   -Djava.util.logging.config.file=%CXF_HOME%\etc\logging.properties
   -Dcxf.config.file=server.xml demo.hw.server.Server
```

The following example command is used to start a client:

```
start java
   -Djava.util.logging.config.file=%CXF_HOME%\etc\logging.properties
   -Dcxf.config.file=client.xml demo.hw.client.Client
   .\wsdl\hello_world.wsdl
```

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 Java configuration

- Configuring and Deploying Artix Solutions, Java Runtime
- Artix Configuration Reference, Java Runtime

Spring Framework

www.springframework.org

CHAPTER 8

Monitoring Artix Services with Actional

This chapter shows examples of monitoring Artix service endpoints and consumers in Actional SOA management tools.

In this chapter

This chapter includes the following sections:

Monitoring Artix Endpoints

Overview

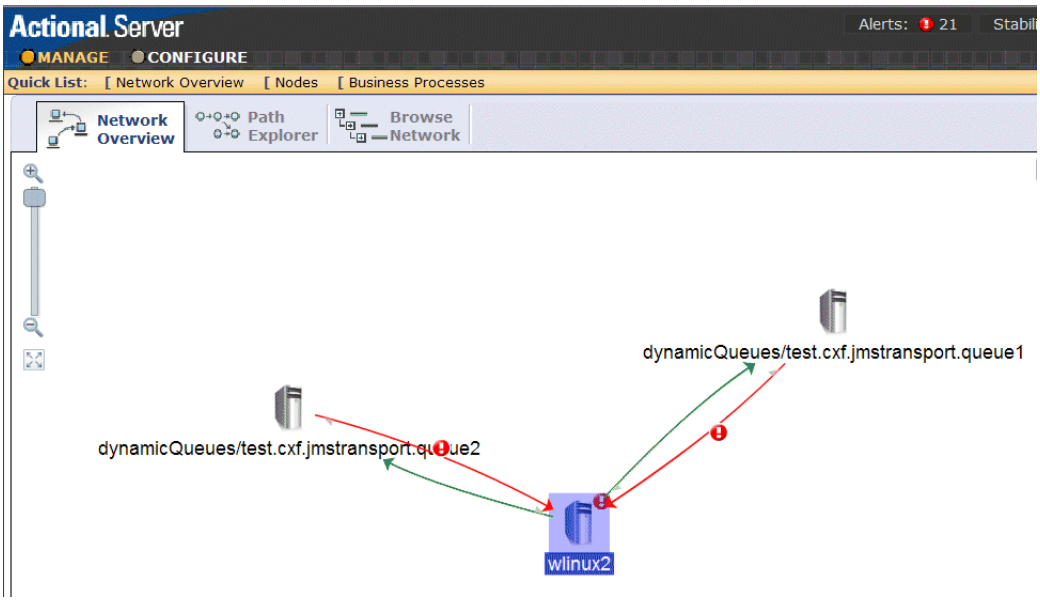
When your Artix service endpoints and consumers have been configured for integration with Actional, they can be viewed in Actional SOA management tools.

For example, when you run the Artix–Actional `jms_queues` sample, the **Actional Server Administration Console** displays the server queues and managed agent nodes. Monitoring information such as response times are displayed as green arrows, while alarms are displayed as red arrows, flowing to and from the queues. The implementation for `server2` includes a delayed response time, which can be viewed in the console.

Network overview

Figure 14 shows a running `jms_queues` sample displayed in the **Network Overview** screen of the **Actional Server Administration Console**.

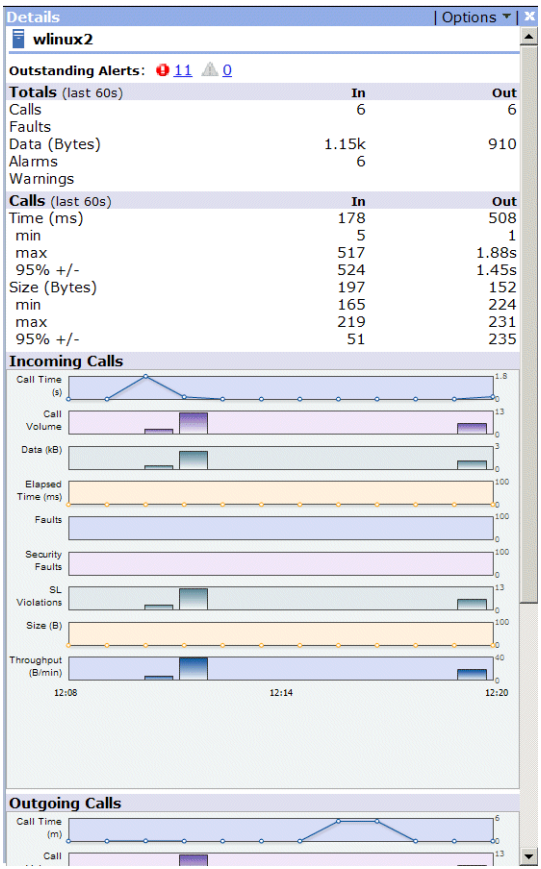
Figure 14: *Actional Server Network Overview*



The **Network Overview** screen provides a high-level overview of the nodes in your network, and displays details such as the flow of calls and alarms between them.

Figure 15 shows the **Details** displayed for the `wlinux2` managed node, which was selected in Figure 14. The **Details** panel displays information such as the total number of calls and alarms, and detailed analysis of incoming and outgoing calls.

Figure 15: *Actional Node Details*



Path Explorer

Figure 16 shows the example `queue1` in the **Path Explorer** view. This view displays the relationships between different components in detail. For example, you can view the call chain between services and consumers. Summary statistics are displayed for selected components. These include the number of calls and alarms, and average response time.

Figure 16: *Actional Server Path Explorer*

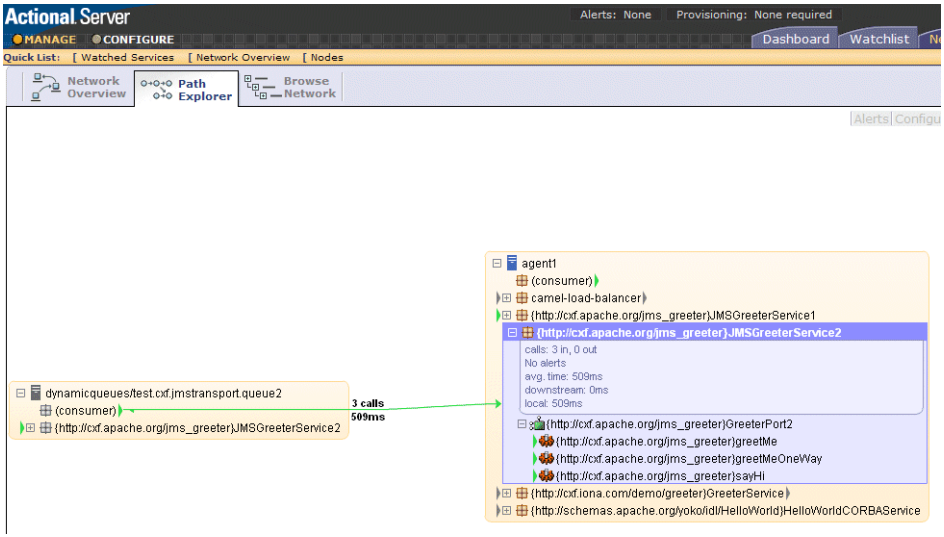
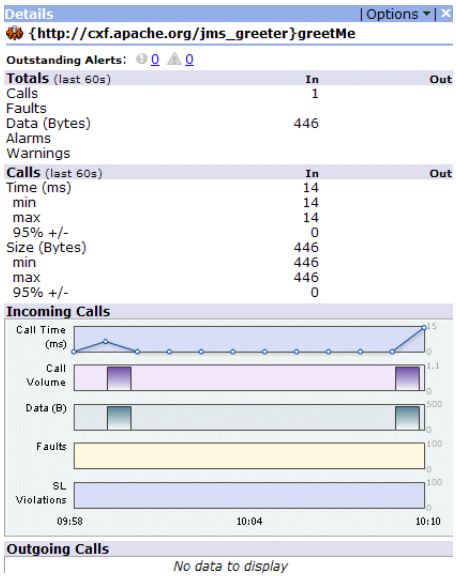


Figure 17 shows the **Details** displayed for the `GreetMe` operation for `JMSGreeterService1`, which is selected in Figure 16.

Figure 17: *Service Details in Actional*



Further information

For detailed information on how to use the **Actional Server Administration Console**, see the Actional product documentation.

Monitoring Routing Patterns

Overview

Artix also supports Actional monitoring of Artix Java service endpoints and Artix Java routes (based on Apache Camel). This enables end-to-end call correlation between an Artix Java consumer, an Artix Java router, and an Artix Java server.

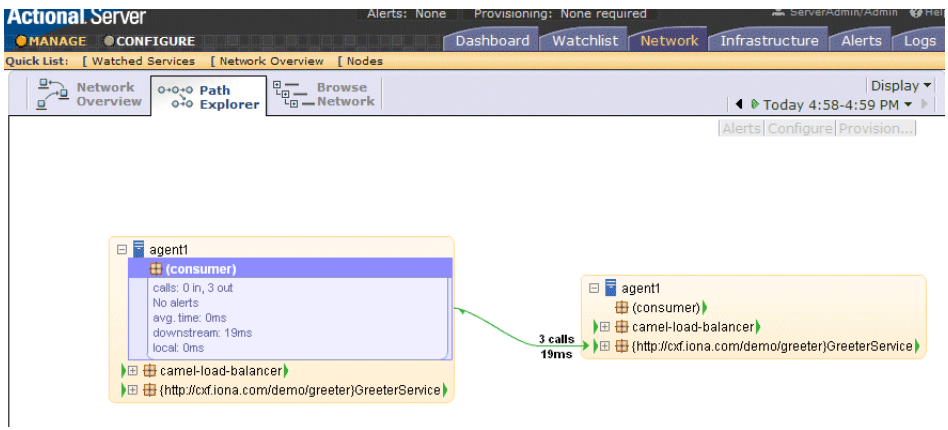
You can use the **Actional Server Administration Console** to monitor a call as it is sent from an Artix Java consumer. This console then shows which Artix Java router processors have processed the call. Finally, it shows the call being forwarded on to an Artix server.

This section shows examples from the **Path Explorer** screen when running the Artix–Actional `router` sample application.

Viewing the Artix Java consumer

Figure 18 shows the call chain starting in the Artix Java consumer displayed in the **Path Explorer**. This also shows the summary statistics displayed for the consumer.

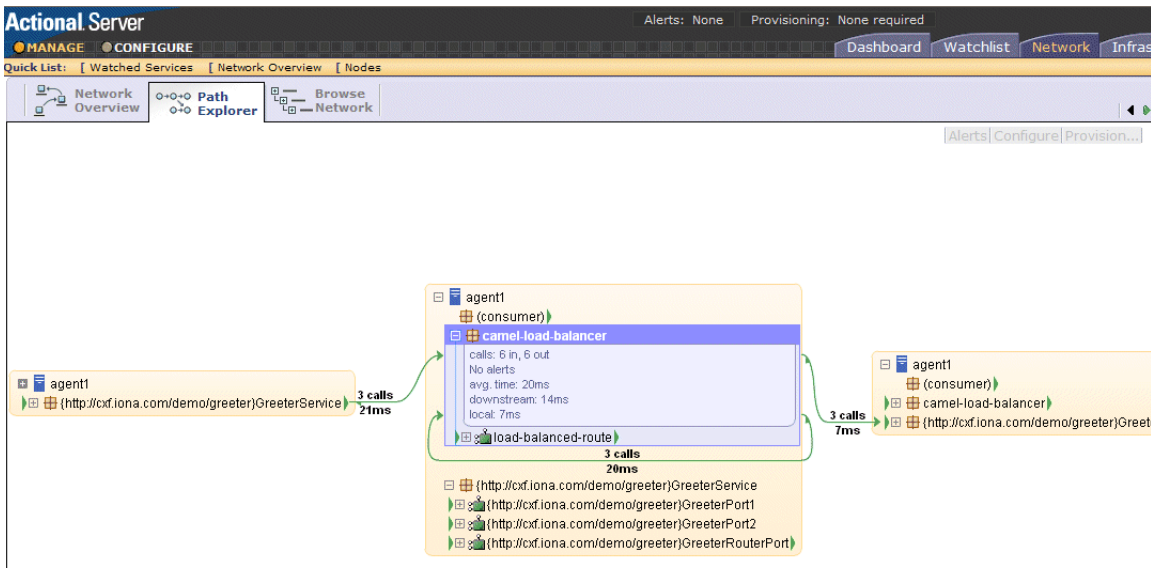
Figure 18: *Artix Java Consumer Call in Actional*



Viewing the Artix Java router

Figure 19 shows the Artix Java router (based on Apache Camel) processing calls between the consumer and service in the **Path Explorer**.

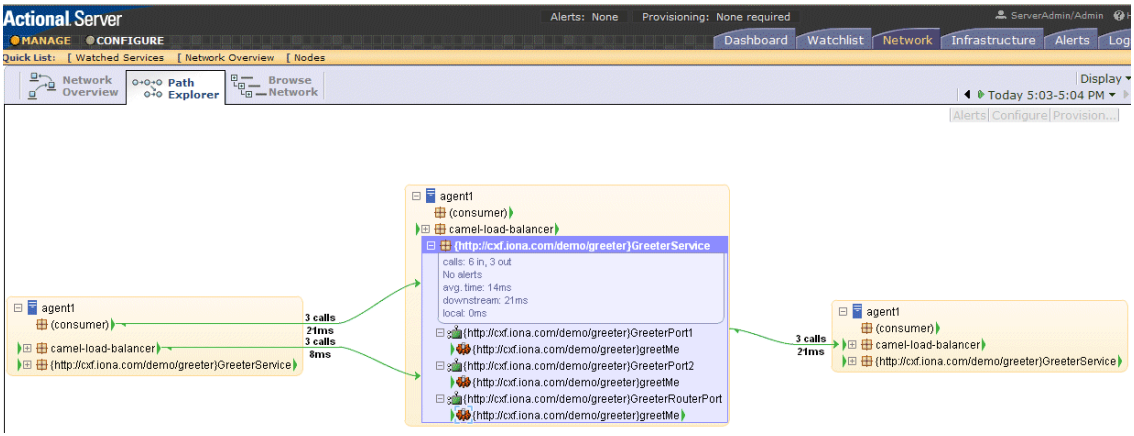
Figure 19: *Artix Java Router Processing in Actional*



Viewing the Artix Java service endpoint

Figure 20 shows the calls forwarded on to the Artix Java service from the router displayed in the **Path Explorer**.

Figure 20: *Artix Java Service Endpoint in Actional*



Note: Actional monitoring of Apache Camel-based routes is supported in the Artix `camel-cxf` component only.

Monitoring CORBA Endpoints

Overview

Artix also supports Actional monitoring of Artix Java CORBA service endpoints and consumers. This use case involves a Web services consumer talking to a Web services endpoint over the IIOP protocol.

You can use the **Actional Server Administration Console** to monitor calls between the CORBA consumer and service endpoint.

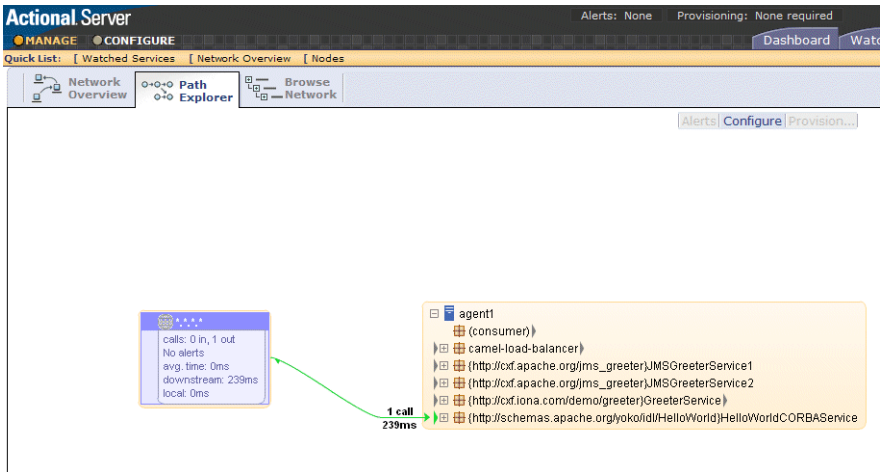
This section shows examples from the **Path Explorer** screen when running the Artix–Actional `corba` sample application.

Note: Actional monitoring is supported in the Artix CORBA binding only. Orbix CORBA clients and servers do not currently support Actional monitoring.

Viewing CORBA consumers and endpoints

Figure 21 shows the Artix CORBA consumer calling the Artix CORBA service endpoint displayed in the **Path Explorer**.

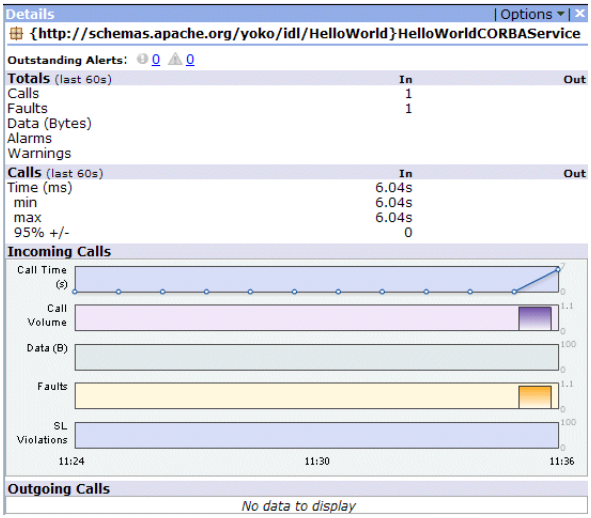
Figure 21: *Artix CORBA Call in Actional*



Viewing an Artix CORBA service endpoint

Figure 17 shows the **Details** displayed for the `HelloWorldCORBAService` shown in Figure 16.

Figure 22: *CORBA Service Details in Actional*



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

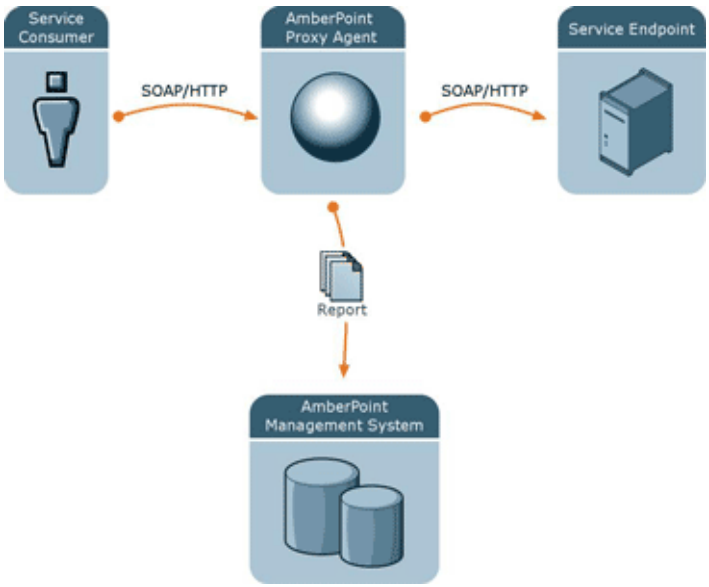
There are two possible approaches to integrating Artix with the AmberPoint SOA management system:

- Using the AmberPoint Proxy Agent.
- Using the 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 23 shows a simple AmberPoint Proxy Agent architecture:

Figure 23: *AmberPoint Proxy Agent Integration*



In this architecture, the following restrictions apply:

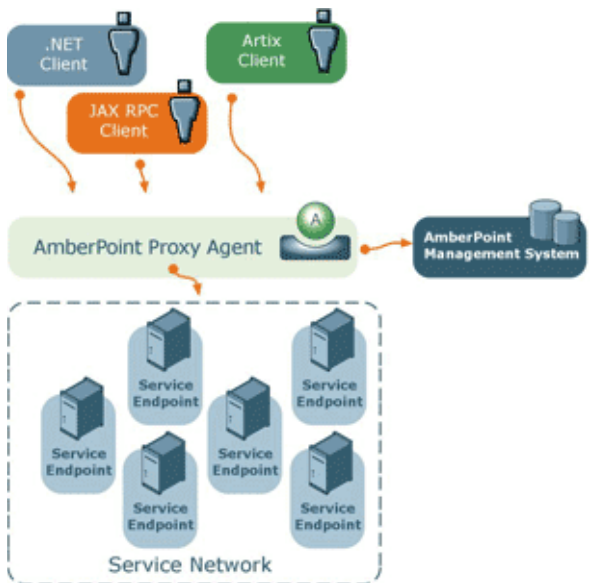
- All messages between the service consumer and service endpoint must be routed through the AmberPoint Proxy Agent.
- All messages must use SOAP over HTTP.
- The service consumer is unaware of the back-end service endpoint, and views its relationship as being with the proxy only.

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

Figure 24: *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 JAX-WS, JavaScript, C++ and JAX-RPC.

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

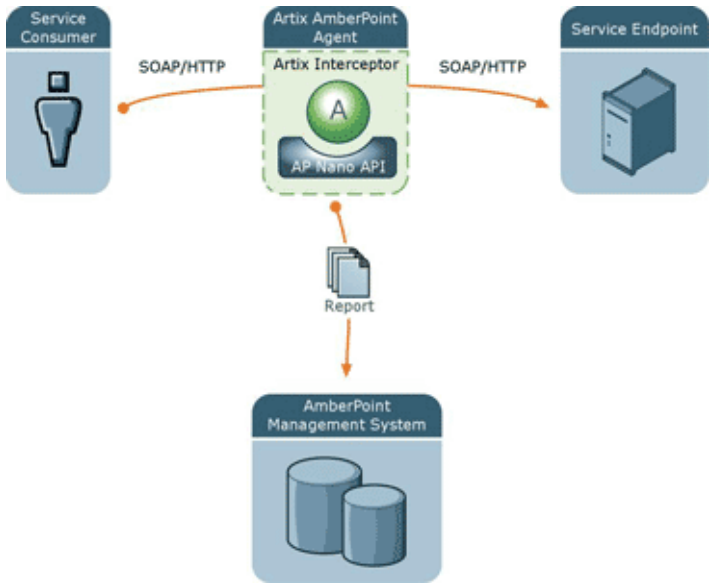
Figure 25 shows how Artix can be integrated with AmberPoint using the Artix AmberPoint Agent.

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.

Figure 25: *Artix AmberPoint Agent Integration*



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 26.
- *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 26: *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. This applies to the Artix Java and C++ core runtimes. The Artix interceptor then reports performance metrics using the AmberPoint nano agent API.

Artix service endpoints

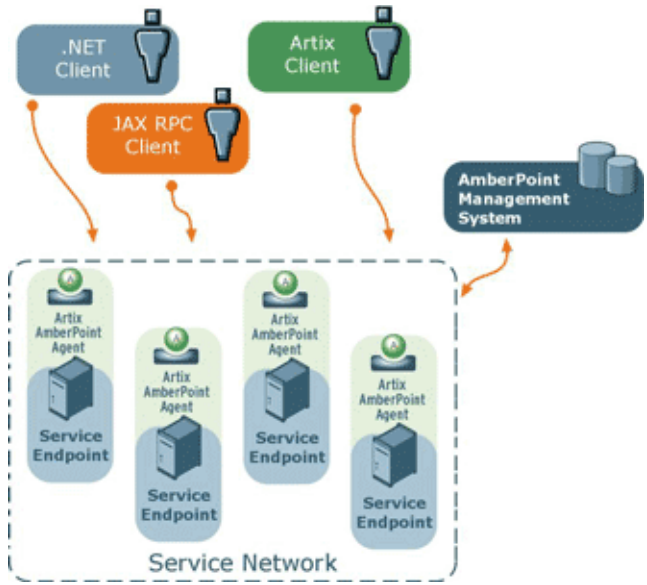
An Artix service endpoint is a service built using Artix, and described using WSDL. The endpoint can be implemented using C++, JAX-RPC, JAX-WS, or even a scripting language, such as JavaScript. However, its main characteristic is that it can be described in WSDL, and classified as a service, which can 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 27 shows the Artix AmberPoint Agent deployed in a service network with multiple service consumers and service endpoints.

Figure 27: *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

This guide explains how to integrate Artix services implemented in JAX-WS and JavaScript with AmberPoint. For details of integrating Artix services implemented in C++ and JAX-RPC, see the *Artix Management Guide, C++ Runtime*.

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

Artix 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 Java Services for AmberPoint Integration" on page 106).

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 Java Services for AmberPoint Integration

Overview

This section explains how to configure Artix JAX-WS services to support the Artix AmberPoint Agent. It describes Artix AmberPoint demo configuration settings. 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/java/samples/management/amberpoint
```

This `amberpoint` demo is based on `.../java/samples/basic/wsdl_first`, with some modifications to enable Artix and AmberPoint integration.

Server configuration

The Artix Java configuration mechanism uses the XML-based Spring Framework. The following code shows the server-side configuration taken from the `server.xml` file in the Artix `amberpoint` demo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- -->
<!-- Copyright (c) 1993-2006 IONA Technologies PLC. -->
<!--            All Rights Reserved. -->
<!-- -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <!-- wiring server life cycle listener for gathering the server's endpoint information -->
    <bean id="com.iona.cxf.management.amberpoint.ServerLifeCycleListenerImpl"
    class="com.iona.cxf.management.amberpoint.ServerLifeCycleListenerImpl">
        <property name="bus" ref="cxf" />
    </bean>
```

```
    <!-- wiring the Nano Agent Logger factory for writing logger to apsocketconverter -->
    <bean id="com.iona.cxf.management.amberpoint.nanoagent.NanoAgentLoggerFactory"
    class="com.iona.cxf.management.amberpoint.nanoagent.NanoAgentLoggerFactory">
        <property name="bus" ref="cxf" />
    </bean>

    <!-- wiring the Amberpoint integration feature-->
     <jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                  createdFromAPI="true">
        <jaxws:features>
         <bean class="com.iona.cxf.management.amberpoint.interceptor.InvocationMessageFeature"/>
        </jaxws:features>
     </jaxws:endpoint>
</beans>
```

This example shows the configuration setting for the server lifecycle listener, which gathers the server's endpoint information. It also shows how to log the server to AmberPoint. And finally, the `hello_world` service endpoint is configured for Artix AmberPoint integration, using the `jaxws:endpoint` attribute.

For details on how to make your configuration available to the Artix Java runtime, see "Configuring the AmberPoint hostname" on page 109.

Client configuration

The following code shows the client-side configuration taken from the `client.xml` file in the Artix `amberpoint` demo.

```
?xml version="1.0" encoding="UTF-8"?>
<!-- -->
<!-- Copyright (c) 1993-2007 IONA Technologies PLC. -->
<!--            All Rights Reserved. -->
<!-- -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <!-- wiring the Nano Agent Logger factory for writing logger to apsocketconverter -->
    <bean id="com.iona.cxf.management.amberpoint.nanoagent.NanoAgentLoggerFactory"
    class="com.iona.cxf.management.amberpoint.nanoagent.NanoAgentLoggerFactory">
        <property name="bus" ref="cxf" />
    </bean>

    <!-- wiring the Amberpoint integration feature-->
    <jaxws:client name="{http://apache.org/hello_world_soap_http}SoapPort"
                  createdFromAPI="true">
        <jaxws:features>
          <bean class="com.iona.cxf.management.amberpoint.interceptor.InvocationMessageFeature"/>
        </jaxws:features>
     </jaxws:client>

</beans>
```

This example shows how to log the client to AmberPoint. It also shows how the `hello_world` client is configured for Artix AmberPoint integration, using the `jaxws:client` attribute.

For details on how to make your configuration available to the Artix Java runtime, see "Configuring the AmberPoint hostname" on page 109.

Configuring the AmberPoint hostname

If you are running your Artix services and the AmberPoint Nano Agent Server on different machines, you must update the hostname in your AmberPoint Nano Agent Client configuration file. For example:

```
ArtixInstallDir/java/samples/management/amberpoint/apobserver.configuration
```

You should update the `messageLogWriter logName` attribute to point the hostname or IP address where the AmberPoint 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 105.

Accessing Artix Java configuration

You can make your Artix Java configuration available to the Artix Java runtime in one of the following ways:

- Use one of the following command-line arguments to point to your XML configuration file:

```
-Dcxf.config.file.url=<myCfgURL>
-Dcxf.config.file=<myCfgResource>
```

  This enables you to save your XML configuration file anywhere on your system and avoid adding it to your `CLASSPATH`.
- Specify the XML configuration file on your `CLASSPATH`.
- Programmatically, by creating a bus and passing the configuration file location as either a URL or string, as follows:

```
(new SpringBusFactory()).createBus(URL myCfgURL)
(new SpringBusFactory()).createBus(String myCfgResource)
```

Demo examples

The Artix Java sample applications uses the command-line approach. For example, in the Artix AmberPoint demo, the following command is used to start the server:

```
start java -Djava.util.logging.config.file=%CXF_HOME%\etc\logging.properties
   -Dcxf.config.file=server.xml demo.hw.server.Server
```

The following command is used to start the client:

```
start java -Djava.util.logging.config.file=%CXF_HOME%\etc\logging.properties
   -Dcxf.config.file=client.xml demo.hw.client.Client .\wsdl\hello_world.wsdl
```

Viewing the demo in AmberPoint

You can use the following AmberPoint tools to view the demo application.

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. You can manually create dependencies between services using the AmberPoint GUI 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 103). 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/java/samples/management/amberpoint\README.txt
```

Artix Java configuration

- Configuring and Deploying Artix Solutions, Java Runtime
- Artix Configuration Reference, Java Runtime

Spring Framework

www.springframework.org

Part V

BMC Patrol

In this part

This part contains the following chapters:

CHAPTER 11

Integrating with BMC Patrol™

This chapter introduces the integration of Artix 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 supports integration with the BMC Patrol Enterprise Management System. This section includes the following topics:

- "The application life cycle"
- "Enterprise Management Systems"
- "Artix BMC Patrol integration"
- "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.

For example, if an application fails because the `/tmp` directory fills up on a particular host, 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 is designed for EMS integration using a common management instrumentation layer. This provides a base that can be used to integrate with any EMS.

In addition, Artix provides packaged integrations that provide out-of-the-box integration with major EMS products. This guide describes IONA's integration with BMC Patrol products.

Artix BMC Patrol integration

The Artix 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 metrics 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.

How it works

The Artix Java runtime integration with BMC Patrol obtains server metrics using JMX-based Artix interceptors. Figure 1 on page 15 shows this overall architecture.

Artix also 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 BMC Knowledge Modules execute parameter collection periodically on each host. The Knowledge Modules compare the response times and other values against the defined alarm ranges, and issue an alarm event if a threshold has been breached. These events can be analyzed and appropriate action taken automatically.

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:

- "Requirements"
- "Main components"
- "Example metrics"
- "Further information"

Requirements

To use the BMC Patrol integration, you must have both Artix 5.x and BMC Patrol 3.4 or higher installed. The Artix BMC Patrol integration is compatible with the BMC Patrol 7 Central Console.

Main components

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

- `IONA_SERVERPROVIDER`
- `IONA_OPERATIONPROVIDER`

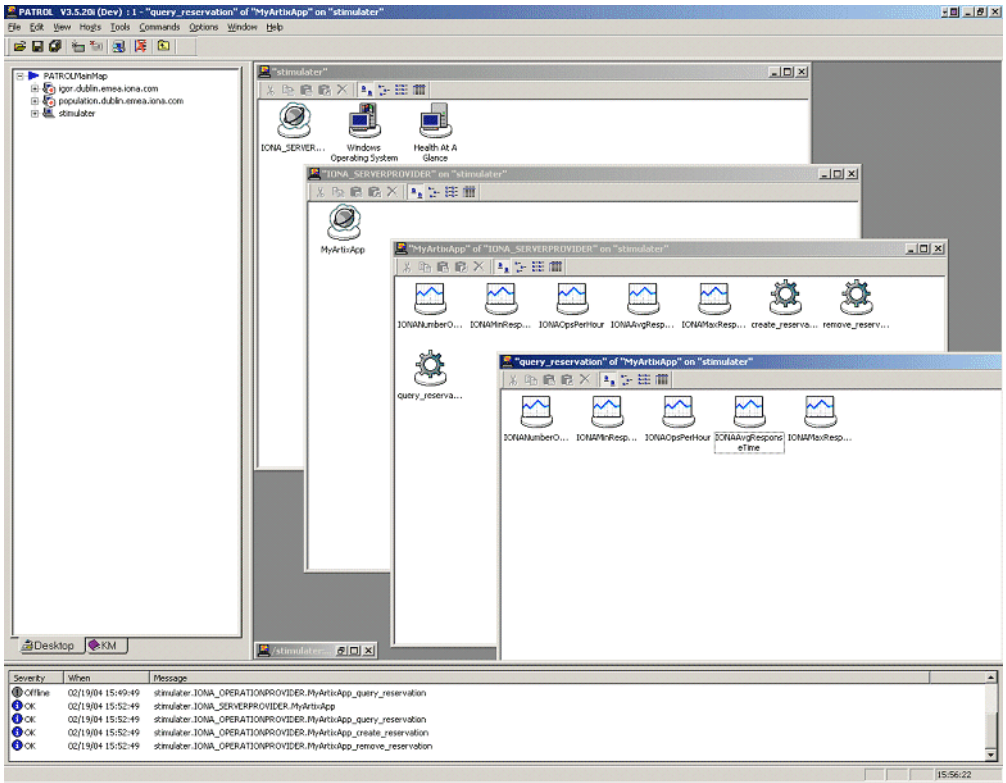
`IONA_SERVERPROVIDER.km` tracks key metrics associated with your Artox servers on a particular host. It also enables servers to be started, stopped, or restarted, if suitably configured.

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

Example metrics

Figure 28 shows an example of the `IONA_SERVERPROVIDER` Knowledge Module displayed in BMC Patrol. The window in focus shows the Artix performance metrics that are available for an operation named `query_reservation`, running on a machine named `stimulator`.

Figure 28: *Artix Server Running in BMC Patrol*



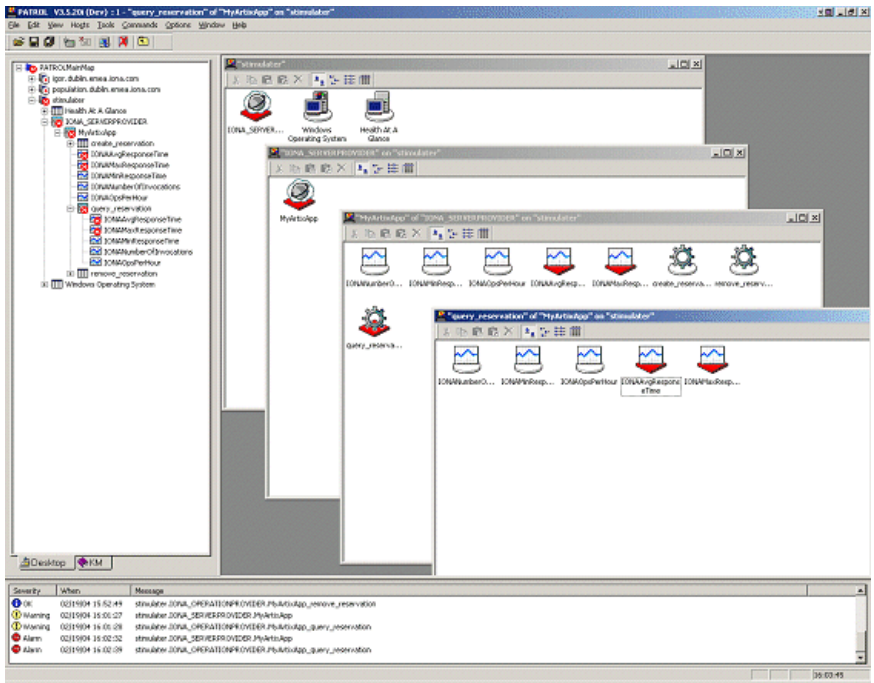
The Artix server performance metrics include the following:

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

For more details, see "Using the Artix Knowledge Module" on page 128.

Figure 29 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 29: *BMC Patrol Displaying Alarms*



Further information

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

CHAPTER 12

Configuring your Artix Environment for BMC

This chapter explains the steps that you need to perform in your Artix environment 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"
- "Further information"

EMS configuration files

You need to create a `servers.conf` text file to configure the BMC Patrol integration. This file is used to track your Artix applications in BMC Patrol. You will find a starting point file 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 this file, the starting point file is located in the following directory:

```
UnzipDir/lib/iona/conf
```

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\management
    \bmc-patrol\BMCCounterServer.log

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

Copy the EMS file to your BMC installation

When you have added content to your `servers.conf` file, copy this file 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 136.

For a complete explanation of configuring performance logging, see Configuring and Deploying Artix Solutions, Java 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 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 integration 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 Knowledge Module, do the following:

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 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 `servers.conf` EMS configuration file. Copy this generated file 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, perform the following steps:

1. Start your **BMC Console** and connect to the **BMC Patrol Agent** on the host where you have installed the 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

The `IONA_SERVERPROVIDER` class represents instances of Artix server or client applications. The parameters exposed in the Knowledge Module are shown in Table 6.

Table 6: *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 6 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 7 shows the Artix parameters that are tracked for each operation instance:

Table 7: *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 30 shows BMC Patrol graphing the value of the `IONAAvgResponseTime` parameter on a `query_reservation` operation call.

Figure 30: *BMC Graphing for IONAAvgResponseTime*

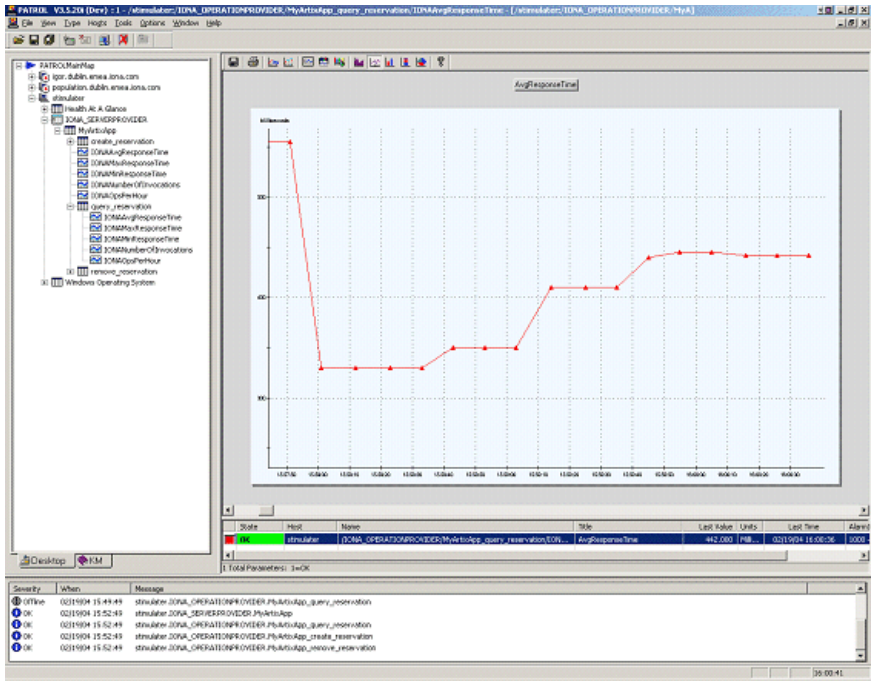
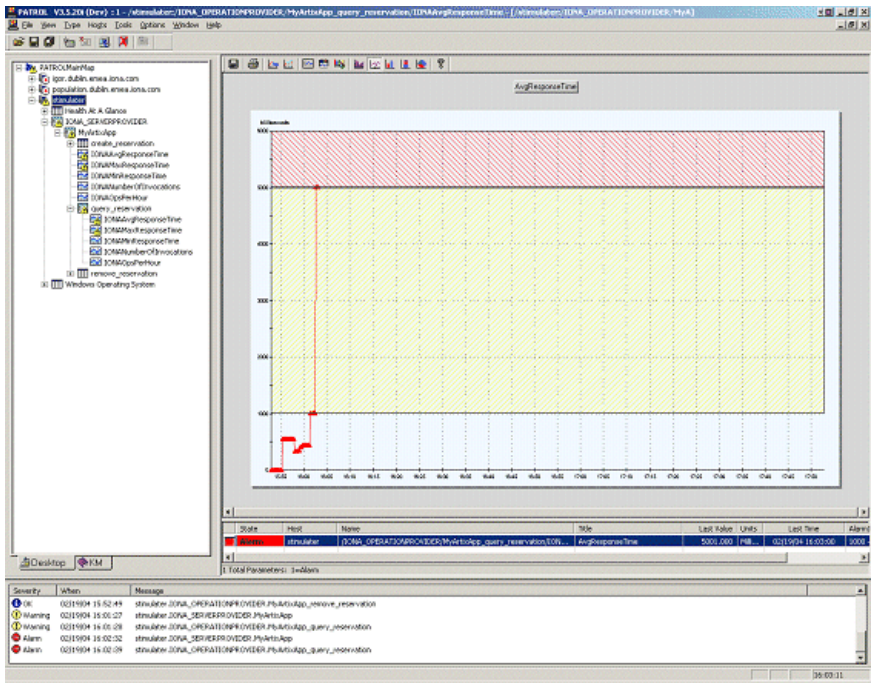


Figure 31 shows warnings and alarms issued for the `IONAAvgResponseTime` parameter.

Figure 31: *BMC Alarms for IONAAvgResponseTime*



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 Artix 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

For Artix JAX-WS applications, add the following example settings to your Artix application's Spring-based XML configuration file:

```
// managed_spring_server.xml

<?xml version="1.0" encoding="UTF-8"?>
<!-- -->
<!-- Copyright (c) 1993-2007 IONA Technologies PLC. -->
<!--            All Rights Reserved. -->
<!-- -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:im="http://cxf.apache.org/management"
       xsi:schemaLocation="http://www.springframework.org/schema/beans/
   http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- JMX InstrumetationManager settings -->
    <bean id="InstrumentationManager"
   class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
        <property name="bus" ref="cxf" />
        <property name="enabled" value="true" />
        <property name="JMXServiceURL"
   value="service:jmx:rmi:///jndi/rmi://localhost:9914/jmxrmi" />
    </bean>

    <!-- Wiring the counter repository -->
    <bean id="CounterRepository"
   class="org.apache.cxf.management.counters.CounterRepository">
        <property name="bus" ref="cxf" />
    </bean>
```

```
 <!-- BMC counter monitor setting for writing the performance log file-->
    <bean id="BMCCounterMonitor"
   class="com.iona.cxf.management.bmc.counters.BMCCounterMonitor">
    <property name="bus" ref="cxf" />
    <property name="serverID" value="management-bmc-patrol-demo-server" />
    <property name="fileName" value="BMCCounterServer.log" />
    <property name="granularityPeriod" value="30" />
    </bean>
</beans>
```

The performance log file location is specified in the `servers.conf` configuration file (see "Creating a servers.conf file" on page 123).

For a detailed JAX-WS sample application, see the following Artix demo:

```
ArtixInstallDir\java\samples\management\bmc-patrol
```

Monitoring Artix applications on multiple hosts

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

Distributing the 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 KM on this machine (see "Setting up your BMC Patrol Environment" on page 126).
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
   ```

   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 have 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.

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` files from each of the applications into single `servers.conf` 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.

Further information

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

Index