Java Dynamic Management Kit 5.1 Tutorial

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–7609–10
June, 2004

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

LA DOCUMENTATION EST FOURNIE “EN L’ÉTAT” ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L’APTITUDE A UNE UTILISATION PARTICULIERE OU A L’ABSENCE DE CONTREFAÇON.

Adobe PostScript

040526@9061

Contents

Examples

Figures

Preface

The Java™ Dynamic Management Kit (Java DMK) 5.1 provides a set of Java classes and tools for developing dynamic management solutions. This product conforms to the Java Management Extensions (JMX), v1.2 Maintenance Release, and the JMX Remote API, v1.0. These specifications define a three-level architecture:

- Instrumentation of resources
- Dynamic agents
- Remote management applications

The JMX architecture is applicable to network management, remote system maintenance, application provisioning, and the management needs of the service-based network.

Once you are familiar with management concepts, the *Java Dynamic Management Kit 5.1 Tutorial* is intended to demonstrate each of the management levels and how they interact. The parts of this tutorial will show you:

- The different ways of making your resources manageable
- How to write an agent and add management services dynamically
- How to access your resources from a remote management application
- The mechanism used to forward events and exceptions from agent to manager

Taken as a whole, these topics will demonstrate the complete development process for implementing a management solution in the Java programming language.

This book also features a part devoted to the details of programming simple network management protocol (SNMP) managers and agents (peers) using the Java DMK.

Changes Between Versions 5.0 and 5.1 of Java DMK

The following are the main changes and additions to Java DMK since the 5.0 release:

- Instrumentation and Agent services compatible with the latest JMX 1.2 Maintenance Release.
- Secure and interoperable remote access compatible with the new JMX Remote API 1.0 Specification, including support for both the RMI-based and JMXMP-based standard connectors (see Chapter 9).
- Flexible authentication and privacy based on the Simple Authentication and Security Layer (SASL) 1.1 Specification and TLS (see Chapter 11).
- SASL mechanisms providing authentication: SASL-PLAIN, DIGEST-MD5, CRAM-MD5, and GSSAPI/Kerberos (see "11.4.2 SASL Provider" on page 198).
- SASL mechanisms providing connection privacy: DIGEST-MD5 and GSSAPI/Kerberos (see "11.4.1 SASL Privacy" on page 195).
- Fine-grained access control based on an authenticated client (see "11.3 Fine-Grained Security" on page 189).
- Wrapping of existing Java DMK 5.0 RMI and HTTP(S) connectors such that applications based on the standard JMX Remote API can interoperate with existing Java DMK-based applications (see "9.5 Wrapping Legacy Connectors" on page 133).
- Enhanced Cascading service, supporting both the JMX Remote API connectors and the legacy Java DMK connectors (see Chapter 14).
- Enhanced Discovery service, allowing the discovery of Java DMK based applications using legacy connectors as well as those using the new connectors (see Chapter 15).

Who Should Use This Book

This tutorial is for developers who would like to learn how to instrument new or existing resources for management, write dynamic agents, or write management applications. You should be familiar with Java programming. Some tutorials also rely on system and network management concepts: knowledge of these is helpful, though not required.

This book is not intended to be an exhaustive reference. Management concepts and product features are covered in the *Java Dynamic Management Kit 5.1 Getting Started Guide*, and the complete API documentation generated by the Javadoc™ tool are provided in the online documentation package.

Before You Read This Book

To build and run the sample programs in this tutorial or use the tool commands provided in the Java DMK, you must have a complete installation of the product on your system. Before programming with the Java DMK you should be familiar with the concepts and tools used throughout this tutorial.

Related Documentation

The Java DMK documentation set includes the following documents:

Book Title	Part Number
Java Dynamic Management Kit 5.1 Installation README	N/A
Java Dynamic Management Kit 5.1 Getting Started Guide	816–7607
Java Dynamic Management Kit 5.1 Tutorial	816–7609
Java Dynamic Management Kit 5.1 Tools Reference Guide	816–7608
Java Dynamic Management Kit 5.1 Release Notes	N/A

These books are available online after you have installed the Java DMK documentation package. The online documentation also includes the API documentation generated by the Javadoc tool for the Java packages and classes. To access the online documentation, using any web browser, open the home page corresponding to your platform.

Operating Environment	Homepage Location
Solaris / Linux / Windows 2000	*installDir*`/SUNWjdmk/5.1/doc/index.html`

In these file names, *installDir* refers to the base directory or folder of your Java DMK installation. In a default installation procedure, *installDir* is as follows.

- `/opt` on the Solaris or Linux platforms
- `C:\Program Files` on the Windows 2000 platform

These conventions are used throughout this book whenever referring to files or directories that are part of the installation.

The Java Dynamic Management Kit relies on the management architecture of two Java Specification Requests (JSRs): the JMX specification (JSR 3) and the JMX Remote API specification (JSR 160). The specification documents and reference implementations of these JSRs are available at:

`http://java.sun.com/products/JavaManagement/download.html`

Further Documentation

The structure of this book is based on the *The Java Tutorial*:

- Online version:
  `http://java.sun.com/docs/books/tutorial/`
- Paperback reference:
  The Java Tutorial Third Edition by Mary Campione, Kathy Walrath and Alison Huml; book and CD-ROM edition (December 2000); Addison-Wesley Pub. Co.; ISBN: 0201703939

Some chapters in the part on SNMP refer to RFC standards for further information. The complete text of RFC papers can be found on the Internet site of the Internet Engineering Task Force:

`http://www.ietf.org/`

Disclaimer – This site is in no way affiliated with Sun Microsystems, Inc. and Sun makes no claim as to the accuracy or relevance of the data it contains.

Directories and Classpath

These tutorials are based on the example programs shipped with the Java DMK. Each example is a set of Java source code files in a separate subdirectory. The following table gives the location of the main examples directory:

Operating Environment	Examples Directory
Solaris / Linux / Windows 2000	*installDir*/SUNWjdmk/5.1/examples

Within the examples directory, the examples themselves are separated into two sub-directories: legacy and current. This is to make a clear distinction between the examples that demonstrate the older implementations of features that have been superseded by the implementation of JMX Remote API in Java DMK 5.1. The implementations of features that have been deprecated in Java DMK 5.1 are now held in the legacy directory. The examples of the current features are held in current.

Except where noted, the source code in this book is taken from these example programs. However, some code fragments might be rearranged and comments might be changed. Program listings in the tutorials usually simplify comments and omit output statements for space considerations.

On the Solaris platform, you must have root access to write in the installed examples directory. For this reason, it might be necessary to copy all examples to a different location before compiling them. Throughout the rest of this book, we will use the term *examplesDir* to refer to the directory in which you compile and run the examples.

When either compiling or running the example programs, make sure that your PATH and CLASSPATH environment variables are correctly set to use the Java 2 Platform, Standard Edition (J2SE). In addition, your classpath must also contain the Java archive (JAR) files for the Java DMK runtime libraries, as well as the current directory (.). If you are using J2SE 1.4.x, then you must also include the runtime libraries for JMX and JMX Remote API, as well as other JAR files relating to the Simple Authentication Security layer (SASL) and Secure Sockets Layer (SSL) security mechanisms, should you require them.

- The Java DMK runtime library:
  - SUNWjdmkrt/5.1/lib/jdmkrt.jar
- The JMX, JMX Remote API, SASL and SSL runtime libraries:
  - SUNWjdmk/5.1/lib/jmx.jar
  - SUNWjdmk/5.1/lib/jmxremote.jar
  - SUNWjdmk/5.1/lib/jmxremote_optional.jar
  - SUNWjdmk/5.1/lib/sunsasl.jar
  - SUNWjdmk/5.1/lib/sasl.jar
  - SUNWjdmk/5.1/lib/rmissl.jar
  -
- The runtime library for the Java DMK tool kit:
  - SUNWjdmk/5.1/lib/jdmktk.jar

At the time of writing, a Beta release of version 1.5.0 of the J2SE platform is available. The JMX API and the mandatory part of JMX Remote API have been included in J2SE 1.5, so if you are using version 1.5.0 of the J2SE platform, you do not need to add the jmx.jar, jmxremote.jar and SASL libraries to your class path.

For a Solaris or RedHat Linux platform, use the classpath given in Table P–1. For a Windows 2000 platform, use the classpath given in Table P–2.

TABLE P–1 Classpath for Compiling or Running the Examples on a Solaris or Linux Platform

J2SE Platform Version	CLASSPATH
1.4.x	*installDir*/SUNWjdmk/5.1/lib/jdmkrt.jar: *installDir*/SUNWjdmk/5.1/lib/jmx.jar: *installDir*/SUNWjdmk/5.1/lib/jmxremote.jar: *installDir*/SUNWjdmk/5.1/lib/jmxremote_optional.jar: *installDir*/SUNWjdmk/5.1/lib/sasl.jar: *installDir*/SUNWjdmk/5.1/lib/sunsasl.jar: *installDir*/SUNWjdmk/5.1/lib/rmissl.jar: *installDir*/SUNWjdmk/5.1/lib/jdmktk.jar:.
1.5.0 (Beta)	*installDir*/SUNWjdmk/5.1/lib/jdmkrt.jar: *installDir*/SUNWjdmk/5.1/lib/jmxremote_optional.jar:.

If you installed the software in the default directory on a Solaris or Linux platform, *installDir* will be /opt.

TABLE P–2 Classpath for Compiling or Running the Examples on a Windows 2000 Platform

J2SE Platform Version	Classpath

1.4.x	*installDir*\SUNWjdmk\5.1\lib\jdmkrt.jar;
	installDir\SUNWjdmk\5.1\lib\jmx.jar;
	installDir\SUNWjdmk\5.1\lib\jmxremote.jar;
	installDir\SUNWjdmk\5.1\lib\mxremote_optional.jar;
	installDir\SUNWjdmk\5.1\lib\sasl.jar;
	installDir\SUNWjdmk\5.1\lib\sunsasl.jar;
	installDir\SUNWjdmk\5.1\lib\rmissl.jar;.

1.5.0 (Beta)	*installDir*\SUNWjdmk-runtime\lib\jdmkrt.jar: *installDir*\SUNWjdmk-runtime-jmx\lib\jmxremote_optional.jar:.

If you installed the software in the default directory on a Windows 2000 platform, *installDir* will be C:\Program Files

These classpaths assume that you are in the subdirectory of a particular example when compiling or running it. Specify the classpath on the command line of the javac and java tools with the -classpath option. The J2SE platform version must match the version of the javac or java command that you are using.

Throughout the rest of this book, we will use the term *classpath* in command-line examples to indicate that you must use the classpath indicated in this section. You can also define this classpath in an environment variable according to your platform and omit its definition on the command line.

To use the `mibgen` tool and the deprecated `proxygen` tool provided with Java DMK, add the installation binary directory to your environment's path. The following table gives the location of this directory:

Operating Environment	Binary Directories
Solaris / Linux / Windows 2000	*installDir*`/SUNWjdmk/5.1/lib/jdmktk.jar`

How This Book Is Organized

This book follows the organization of the *The Java Tutorial*. Each major part covers a subject and each chapter covers a topic within that subject.

- Part I: "Instrumentation Using MBeans" shows various ways of making a resource manageable, using standard MBeans, dynamic MBeans, model MBeans and open MBeans.
- Part II: "Agent Applications" demonstrates the functionality of the MBean server at the heart of an agent.
- Part III: "Remote Management Applications" shows how a distant manager can access the resources in an agent.
- Part IV: "Agent Services" demonstrates the various kinds of management intelligence that can be added dynamically to an agent.
- Part V: "SNMP Interoperability" shows how Java agents can also implement SNMP agents, how to write managers, how to implement SNMP security mechanisms, including SNMPv3 security, and how to implement an SNMP master agent.
- Part VI: "Legacy Features" describes all the features from previous versions of Java DMK that have been superseded by the inclusion of JMX Remote API, but that have been retained for reasons for backwards compatibility.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Ordering Sun Documentation

Sun Microsystems offers select product documentation in print. For a list of documents and how to order them, see "Buy printed documentation" at `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic conventions used in this book.

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine-name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Shell	Prompt
C shell prompt	`machine-name%`
C shell superuser prompt	`machine-name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

PART I

Instrumentation Using MBeans

Given a resource in the Java programming language—either an application, a service or an object representing a device—its *instrumentation* is the way that you expose its management interface. The *management interface* is the set of attributes and operations that are visible to managers that need to interact with that resource. Therefore, *instrumenting a resource* makes it manageable.

This part covers the four ways to instrument a resource:

- Chapter 1 shows how to write a *standard MBean* by following the design patterns defined by the JMX specification. The example shows how an agent then accesses the attributes and operations.
- Chapter 2 shows how to implement a *dynamic MBean* in order to expose a coherent management interface. The example highlights the similarities and differences between dynamic and standard MBeans, with an analysis of performance issues.
- Chapter 3 shows how to dynamically instantiate a configurable *model MBean*. This chapter includes an example of how to create a model MBean, configure its behavior, set its target object, and then manage it in the same way as any other MBean.
- Chapter 4 shows how to write an *open MBean* that enables management applications to understand and use new managed resources as soon as they are discovered at runtime.

When you write a standard MBean, you follow certain design patterns so that the method names in your object expose the attributes and operations to static introspection. Dynamic MBeans implement a generic interface and can expose a rich description of their management interface. Model MBeans are MBean templates whose management interface and resource target are defined at runtime. Open MBeans are managed objects that refer only to a limited and predefined set of data types, that can be combined into compound types such as tables.

CHAPTER 1

Standard MBeans

A standard MBean is the simplest and fastest way to instrument a resource from scratch: Attributes and operations are simply methods which follow certain design patterns. A standard MBean is composed of the MBean interface which lists the methods for all exposed attributes and operations, and the class which implements this interface and provides the functionality of the resource.

The code samples in this chapter are from the files in the `StandardMBean` example directory located in the main *examplesDir*/`current` (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "1.1 Exposing the MBean Interface" on page 35 demonstrates the design patterns for attributes and operations and gives some general rules for writing the MBean interface.
- "1.2 Implementing an MBean" on page 38 shows how the MBean interface is related to the code for the manageable resource.
- "1.3 Running the Standard MBean Example" on page 40 demonstrates the runtime behavior of a standard MBean.

1.1 Exposing the MBean Interface

To expose the MBean interface, first determine the management interface of your resource, that is the information needed to manage it. This information is expressed as attributes and operations. An *attribute* is a value of any type that a manager can get or set remotely. An *operation* is a method with any signature and any return type that the manager can invoke remotely.

Note – Attributes and operations are conceptually equivalent to properties and actions on JavaBeans objects. However, their translation into Java code is entirely different to accommodate the management functionality.

As specified by the Java Management Extensions (JMX) instrumentation, all attributes and operations are explicitly listed in an MBean interface. This is a Java interface that defines the full management interface of an MBean. This interface must have the same name as the class that implements it, followed by the MBean suffix. Because the interface and its implementation are usually in different files, two files make up a standard MBean.

For example, the class `SimpleStandard` (in the file `SimpleStandard.java`) has its management interface defined in the interface `SimpleStandardMBean` (in the file `SimpleStandardMBean.java`).

EXAMPLE 1–1 `SimpleStandardMBean` Interface

```
public interface SimpleStandardMBean {

    public String getState() ;

    public void setState(String s) ;

    public Integer getNbChanges() ;

    public void reset() ;
}
```

Only `public` methods in the MBean interface are taken into consideration for the management interface. When present, non-public methods should be grouped separately, to make the code clearer for human readers.

1.1.1 MBean Attributes

Attributes are conceptual variables that are exposed for management through getter and setter methods in the MBean interface:

- A *getter* is any public method whose name begins with `get` and which does not return `void`; it enables a manager to read the value of the attribute, whose type is that of the returned object.
- A public method whose name begins with `is` and which returns a boolean or `Boolean` is also a getter, though a boolean attribute can have only one getter (it must be one form or the other).
- A *setter* is any public method whose name begins with `set` and which takes a single parameter; it enables a manager to write a new value in the attribute, whose type is that of the parameter.

Attribute types can be arrays of objects, but individual array elements cannot be accessed individually through the getters and setters. Use operations to access the array elements, as described in the next section. The following code example demonstrates an attribute with an array type:

```
public String[] getMessages();
public void setMessages(String[] msgArray);
```

The name of the attribute is the literal part of the method name following `get`, `is`, or `set`. This name is case sensitive in all Java Dynamic Management Kit (Java DMK) objects that manipulate attribute names. Using these patterns, we can determine the attributes exposed in Example 1–1:

- `State` is a readable and writable attribute of type `String`
- `NbChanges` is a read-only attribute of type `Integer`

The specification of the design patterns for attributes implies the following rules:

- Attributes can be read-only, write-only, or readable and writable.
- Attribute names cannot be overloaded. For any given attribute name there can be at most one setter and one getter, and if both are defined, they must use the same type.

1.1.2 MBean Operations

Operations are methods that management applications can call remotely on a resource. They can be defined with any number of parameters of any type and can return any type.

The design patterns for operations are simple. Any public method defined in the MBean interface that is not an attribute getter or setter is an operation. For this reason, getters and setters are usually declared first in the Java code, so that all operations are grouped afterwards. The name of an operation is the name of the corresponding method.

The `SimpleStandardMBean` in Example 1–1 defines one operation, `reset`, which takes no parameters and returns nothing.

While the following methods define valid operations (and not attributes), to avoid confusion with attributes, these types of names should not be used:

```
public void getFoo();
public Integer getBar(Float p);
public void setFoo(Integer one, Integer two);
public String isReady();
```

For performance reasons, you might want to define operations for accessing individual elements of an array type attribute. In this case, use unambiguous operation names:

```
public String singleGetMessage(int index);
public void singleSetMessage(int index, String msg);
```

Note – The Java DMK imposes no restrictions on attribute types, operation attribute types, and operation return types. However, the developer must ensure that the corresponding classes are available to all applications manipulating these objects, and that they are compatible with the type of communication used. For example, attribute and operation types must be serializable to be manipulated remotely using the remote method invocation (RMI) or JMX messaging protocol (JMXMP) protocols.

1.2 Implementing an MBean

The second part of an MBean is the class that implements the MBean interface. This class encodes the expected behavior of the manageable resource in its implementation of the attribute and operation methods. The resource does not need to reside entirely in this class. The MBean implementation can rely on other objects.

If the MBean must be instantiated remotely, it must be a concrete class and must expose at least one public constructor so that any other class can create an instance.

Otherwise, the developer is free to implement the management interface in any way, provided that the object has the expected behavior. Here is the sample code that implements our MBean interface.

EXAMPLE 1–2 `SimpleStandard` Class

```
public class SimpleStandard
    extends NotificationBroadcasterSupport
    implements SimpleStandardMBean {

     public String getState() {
        return state;
    }

    public void setState(String s) {
        state = s;
        nbChanges++;
    }

    public Integer getNbChanges() {
        return nbChanges;
    }

    public void reset() {
             AttributeChangeNotification acn =
```

EXAMPLE 1–2 `SimpleStandard` Class *(Continued)*

```
                new AttributeChangeNotification(this,
                                                    0,
                                                    0,
                                                    "NbChanges reset",
                                                    "NbChanges",
                                                    "Integer",
                                                    new Integer(nbChanges),
                                                    new Integer(0));
              state = "initial state";
           nbChanges = 0;
              nbResets++;
              sendNotification(acn);
    }

    // This method is not a getter in the management sense because
    // it is not exposed in the "SimpleStandardMBean" interface.
    public int getNbResets() {
        return nbResets;
    }

    public MBeanNotificationInfo[] getNotificationInfo() {
        return new MBeanNotificationInfo[] {
             new MBeanNotificationInfo(
             new String[] { AttributeChangeNotification.ATTRIBUTE_CHANGE },
             AttributeChangeNotification.class.getName(),
             "This notification is emitted when the reset()
           method is called.")
           };
}
```

As in this example, attributes are usually implemented as internal variables whose value is returned or modified by the getter and setter methods. However, an MBean can implement any access and storage scheme to fit particular management needs, provided getters and setters retain their read and write semantics. Methods in the MBean implementation can have side-effects, but it is up to you to ensure that these are safe and coherent within the full management solution.

We also see here that by calling the `MBeanNotificationInfo` class, the MBean can send notifications of different types. In this case, the MBean sends notifications about changes in its attributes when the `reset()` method is called.

As we will see later, management applications never have a direct handle on an MBean. They only have an identification of an instance and the knowledge of the management interface. In this case, the mechanism for exposing attributes through methods in the MBean interface makes it impossible for an application to access the MBean directly. Internal variables and methods, and even public ones, are totally encapsulated and their access is controlled by the programmer through the implementation of the MBean interface.

1.3 Running the Standard MBean Example

The *examplesDir*/`StandardMBean` directory contains the `SimpleStandard.java` and `SimpleStandardMBean.java` files which make up the MBean. This directory also contains a simple agent application which instantiates this MBean, introspects its management interface, and manipulates its attributes and operations.

▼ To Run the Standard MBean Example

1. **Compile all files in this directory with the `javac` command.**

   For example, on the Solaris platform with the Korn shell, type:

   ```
   $ cd examplesDir/current/StandardMBean/
   $ javac -classpath classpath *.java
   ```

2. **To run the example, start the agent class which will interact with the `SimpleStandard` MBean:**

   ```
   $ java -classpath classpath StandardAgent
   ```

3. **Press `Enter` when the application pauses, to step through the example.**

   The agent application handles all input and output in this example and gives a view of the MBean at runtime.

We will examine how agents work in Chapter 2, but this example demonstrates how the MBean interface limits the view of what the MBean exposes for management. Roughly, the agent introspects the MBean interface at runtime to determine what attributes and operations are available. You then see the result of calling the getters, setters, and operations.

Part II also covers the topics of object names and exceptions which you see when running this example.

CHAPTER 2

Dynamic MBeans

A dynamic MBean implements its management interface programmatically, instead of through static method names. To do this, it relies on metadata classes that represent the attributes and operations exposed for management. Management applications then call generic getters and setters whose implementation must resolve the attribute or operation name to its intended behavior.

One advantage of this instrumentation is that you can use it to make an existing resource manageable quickly. The implementation of the `DynamicMBean` interface provides an instrumentation wrapper for an existing resource.

Another advantage is that the metadata classes for the management interface can provide human-readable descriptions of the attributes, operations, and the MBean itself. This information could be displayed to a user on a management console to describe how to interact with this particular resource.

The code samples in this chapter are from the files in the `DynamicMBean` example directory located in the main *examplesDir*`/current` directory (see “Directories and Classpath” in the Preface).

This chapter covers the following topics:

- “2.1 Exposing the Management Interface” on page 42 explains the `DynamicMBean` interface and its generic methods common to all dynamic MBeans.
- “2.2 Implementing a Dynamic MBean” on page 43 shows how to implement this interface to expose specific attributes and operations.
- “2.3 Running the Dynamic MBean Example” on page 51 demonstrates the runtime behavior of a dynamic MBean.

2.1 Exposing the Management Interface

In the standard MBean, attributes and operations are exposed statically in the names of methods in the MBean interface. Dynamic MBeans all share the same interface that defines generic methods to access attributes and operations. Because the management interface is no longer visible through introspection, dynamic MBeans must also provide a description of their attributes and operations explicitly.

2.1.1 DynamicMBean Interface

The DynamicMBean class is a Java interface defined by the Java Management Extensions (JMX) specification. It specifies the methods that a resource implemented as a dynamic MBean must provide to expose its management interface. Example 2–1 shows the uncommented code for the DynamicMBean interface.

EXAMPLE 2–1 The DynamicMBean Interface

```
public class SimpleDynamic
    extends NotificationBroadcasterSupport
    implements DynamicMBean {

[...]

    public MBeanInfo getMBeanInfo() {

        return dMBeanInfo;
    }
```

The getMBeanInfo method provides a description of the MBean's management interface. This method returns an dMBeanInfo object that contains the metadata information about attributes and operations.

The attribute getters and setters are generic, since they take the name of the attribute that needs to be read or written. For convenience, dynamic MBeans must also define bulk getters and setters to operate on any number of attributes at once. These methods use the Attribute and AttributeList classes to represent attribute name-value pairs and lists of name-value pairs, respectively.

Because the names of the attributes are not revealed until runtime, the getters and setters are necessarily generic. In the same way, the invoke method takes the name of an operation and its signature, in order to invoke any method that might be exposed.

As a consequence of implementing generic getters, setters, and invokers, the code for a dynamic MBean is more complex than for a standard MBean. For example, instead of calling a specific getter by name, the generic getter must verify the attribute name and then encode the functionality to read each of the possible attributes.

2.1.2 MBean Metadata Classes

A dynamic MBean has the burden of building the description of its own management interface. The JMX specification defines the Java objects used to completely describe the management interface of an MBean. Dynamic MBeans use these objects to provide a complete self description as returned by the `getMBeanInfo` method. Agents also use these classes to describe a standard MBean after it has been introspected.

As a group, these classes are referred to as the *MBean metadata classes* because they provide information about the MBean. This information includes the attributes and operations of the management interface, also the list of constructors for the MBean class, and the notifications that the MBean might send. Notifications are event messages that are defined by the JMX architecture. See Chapter 8.

Each element is described by its metadata object containing its name, a description string, and its characteristics. For example, an attribute has a type and is readable and/or writable. Table 2–1 lists all MBean metadata classes.

TABLE 2–1 MBean Metadata Classes

Class Name	Purpose
`MBeanInfo`	Top-level object containing arrays of metadata objects for all MBean elements; also includes the name of the MBean's Java class and a description string
`MBeanFeatureInfo`	Parent class from which all other metadata objects inherit a name and a description string
`MBeanOperationInfo`	Describes an operation: the return type, the signature as an array of parameters, and the impact (whether the operation just returns information or modifies the resource)
`MBeanConstructorInfo`	Describes a constructor by its signature
`MBeanParameterInfo`	Gives the type of a parameter in an operation or constructor signature
`MBeanAttributeInfo`	Describes an attribute: its type, whether it is readable, and whether it is writable
`MBeanNotificationInfo`	Contains an array of notification type strings

2.2 Implementing a Dynamic MBean

A dynamic MBean consists of a class that implements the `DynamicMBean` interface coherently. By this, we mean a class that exposes a management interface whose description matches the attributes and operations that are accessible through the generic getters, setters and invokers.

A dynamic MBean class can declare any number of public or private methods and variables. None of these are visible to management applications. Only the methods implementing the DynamicMBean interface are exposed for management. A dynamic MBean can also rely on other classes that might be a part of the manageable resource.

2.2.1 Dynamic Programming Issues

An MBean is a manageable resource that exposes a specific management interface. The name *dynamic MBean* refers to the fact that the interface is revealed at runtime, instead of through the introspection of static class names. The term *dynamic* is not meant to imply that the MBean can dynamically change its management interface.

2.2.2 getMBeanInfo Method

Because the MBean description should never change, it is usually created one time only at instantiation, and the getMBeanInfo method simply returns its reference at every call. The MBean constructor should therefore build the MBeanInfo object from the MBean metadata classes such that it accurately describes the management interface. And since most dynamic MBeans will always be instantiated with the same management interface, building the MBeanInfo object is fairly straightforward.

Example 2–2 shows how the SimpleDynamic MBean defines its management interface, as built at instantiation and returned by its getMBeanInfo method.

EXAMPLE 2–2 Implementation of the getMBeanInfo Method

```
private void buildDynamicMBeanInfo() {

    dAttributes[0] =
        new MBeanAttributeInfo("State",
                               "java.lang.String",
                               "State string.",
                               true,
                               true,
                               false);
    dAttributes[1] =
        new MBeanAttributeInfo("NbChanges",
                               "java.lang.Integer",
                               "Number of times the " +
                               "State string has been changed.",
                               true,
                               false,
                               false);

    Constructor[] constructors = this.getClass().getConstructors();
    dConstructors[0] =
        new MBeanConstructorInfo("Constructs a " +
                                 "SimpleDynamic object",
```

EXAMPLE 2–2 Implementation of the `getMBeanInfo` Method *(Continued)*

```
                                   constructors[0]);

        MBeanParameterInfo[] params = null;
        dOperations[0] =
            new MBeanOperationInfo("reset",
                                   "reset State and NbChanges " +
                                   "attributes to their initial values",
                                   params ,
                                   "void",
                                   MBeanOperationInfo.ACTION);

        dNotifications[0] =
            new MBeanNotificationInfo(
            new String[] { AttributeChangeNotification.ATTRIBUTE_CHANGE },
            AttributeChangeNotification.class.getName(),
            "This notification is emitted when the reset() method
            is called.");

        dMBeanInfo = new MBeanInfo(dClassName,
                                   dDescription,
                                   dAttributes,
                                   dConstructors,
                                   dOperations,
                                   dNotifications);
    }
// PRIVATE VARIABLES

    private MBeanAttributeInfo[] dAttributes =
        new MBeanAttributeInfo[2];
    private MBeanConstructorInfo[] dConstructors =
        new MBeanConstructorInfo[1];
    private MBeanNotificationInfo[] dNotifications =
        new MBeanNotificationInfo[1];
    private MBeanOperationInfo[] dOperations =
        new MBeanOperationInfo[1];
    private MBeanInfo dMBeanInfo = null;
}
```

2.2.3 Generic Attribute Getters and Setters

Generic getters and setters take a parameter that indicates the name of the attribute to read or write. There are two issues to keep in mind when implementing these methods:

- Attribute names must be correctly mapped to their corresponding internal representation.
- Invalid attribute names and types should raise an exception, including when writing to a read-only attribute and reading a write-only attribute.

The getAttribute method is the simplest, since only the attribute name must be verified.

EXAMPLE 2–3 Implementation of the getAttribute Method

```
public Object getAttribute(String attribute_name)
    throws AttributeNotFoundException,
           MBeanException,
           ReflectionException {

    // Check attribute_name to avoid NullPointerException later on
    if (attribute_name == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException("Attribute name cannot be null"),
            "Cannot invoke a getter of " + dClassName +
                " with null attribute name");
    }

    // Call the corresponding getter for a recognized attribute_name
    if (attribute_name.equals("State")) {
        return getState();
    }
    if (attribute_name.equals("NbChanges")) {
        return getNbChanges();
    }

    // If attribute_name has not been recognized
    throw(new AttributeNotFoundException(
        "Cannot find " + attribute_name + " attribute in " + dClassName));
}

// internal methods for getting attributes
public String getState() {
    return state;
}

public Integer getNbChanges() {
    return new Integer(nbChanges);
}

// internal variables representing attributes
private String      state = "initial state";
private int         nbChanges = 0;
```

The setAttribute method is more complicated, since you must also ensure that the given type can be assigned to the attribute and handle the special case for a null value.

EXAMPLE 2–4 Implementation of the setAttribute Method

```
public void setAttribute(Attribute attribute)
    throws AttributeNotFoundException,
           InvalidAttributeValueException,
           MBeanException,
           ReflectionException {
```

EXAMPLE 2–4 Implementation of the `setAttribute` Method *(Continued)*

```
    // Check attribute to avoid NullPointerException later on
    if (attribute == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException("Attribute cannot be null"),
            "Cannot invoke a setter of " + dClassName +
                " with null attribute");
    }
    String name = attribute.getName();
    Object value = attribute.getValue();

    if (name.equals("State")) {
        // if null value, try and see if the setter returns any exception
        if (value == null) {
            try {
                setState( null );
            } catch (Exception e) {
                throw(new InvalidAttributeValueException(
                    "Cannot set attribute "+ name +" to null"));
            }
        }
        // if non null value, make sure it is assignable to the attribute
        else if (String.class.isAssignableFrom(value.getClass())) {
                    setState((String) value);
        } else {
                    throw new InvalidAttributeValueException(
                        "Cannot set attribute "+ name +
                            " to a " + value.getClass().getName() +
                            " object, String expected");
                }
            }
    // recognize an attempt to set a read-only attribute
    else if (name.equals("NbChanges")) {
        throw new AttributeNotFoundException(
            "Cannot set attribute "+ name +
                " because it is read-only");
    }

    // unrecognized attribute name
    else {
        throw new AttributeNotFoundException(
            "Attribute " + name + " not found in " +
                this.getClass().getName());
    }
}

// internal method for setting attribute
public void setState(String s) {
    state = s;
    nbChanges++;
}
```

Notice that if a change in your management solution requires you to change your management interface, it will be harder to do with a dynamic MBean. In a standard MBean, each attribute and operation is a separate method, so unchanged attributes are unaffected. In a dynamic MBean, you must modify the generic methods that encode all attributes.

2.2.4 Bulk Getters and Setters

The `DynamicMBean` interface includes bulk getter and setter methods for reading or writing more than one attribute at once. These methods rely on the classes shown in Table 2–2.

TABLE 2–2 Bulk Getter and Setter Classes

Class Name	Description
`Attribute`	A simple object that contains the name string and value object of any attribute
`AttributeList`	A dynamically extendable list of `Attribute` objects (extends `java.util.ArrayList)`

The `AttributeList` class extends the `java.util.ArrayList` class.

The bulk getter and setter methods usually rely on the generic getter and setter, respectively. This makes them independent of the management interface, which can simplify certain modifications. In this case, their implementation consists mostly of error checking on the list of attributes. However, all bulk getters and setters must be implemented so that an error on any one attribute does not interrupt or invalidate the bulk operation on the other attributes.

If an attribute cannot be read, then its name-value pair is not included in the list of results. If an attribute cannot be written, it will not be copied to the returned list of successful set operations. As a result, if there are any errors, the lists returned by bulk operators will not have the same length as the array or list passed to them. In any case, the bulk operators *do not* guarantee that their returned lists have the same ordering of attributes as the input array or list.

The `SimpleDynamic` MBean shows one way of implementing the bulk getter and setter methods.

EXAMPLE 2–5 Implementation of the Bulk Getter and Setter Methods

```
public AttributeList getAttributes(String[] attributeNames) {

    // Check attributeNames to avoid NullPointerException later on
    if (attributeNames == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException(
```

EXAMPLE 2–5 Implementation of the Bulk Getter and Setter Methods *(Continued)*

```
                "attributeNames[] cannot be null"),
            "Cannot invoke a getter of " + dClassName);
    }
    AttributeList resultList = new AttributeList();

    // if attributeNames is empty, return an empty result list
    if (attributeNames.length == 0)
            return resultList;

    // build the result attribute list
    for (int i=0 ; i<attributeNames.length ; i++){
        try {
            Object value = getAttribute((String) attributeNames[i]);
            resultList.add(new Attribute(attributeNames[i],value));
        } catch (Exception e) {
            // print debug info but continue processing list
            e.printStackTrace();
        }
    }
    return(resultList);
}

public AttributeList setAttributes(AttributeList attributes) {

    // Check attributes to avoid NullPointerException later on
    if (attributes == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException(
                "AttributeList attributes cannot be null"),
            "Cannot invoke a setter of " + dClassName);
    }
    AttributeList resultList = new AttributeList();

    // if attributeNames is empty, nothing more to do
    if (attributes.isEmpty())
        return resultList;

    // try to set each attribute and add to result list if successful
    for (Iterator i = attributes.iterator(); i.hasNext();) {
        Attribute attr = (Attribute) i.next();
        try {
            setAttribute(attr);
            String name = attr.getName();
            Object value = getAttribute(name);
            resultList.add(new Attribute(name,value));
        } catch(Exception e) {
            // print debug info but keep processing list
            e.printStackTrace();
        }
    }
    return(resultList);
}
```

2.2.5 Generic Operation Invoker

A dynamic MBean must implement the `invoke` method so that operations in the management interface can be called. This method requires the same considerations as the generic getter and setter:

- Operations and their parameters must be correctly mapped to their internal representation and the result must be returned.
- Operation names and parameter types must be verified.
- These verifications are usually hard-coded, again making modifications to the management interface more difficult than in a standard MBean.

The implementation in the `SimpleDynamic` MBean is relatively simple due to the one operation with no parameters.

EXAMPLE 2–6 Implementation of the `invoke` Method

```
public Object invoke(
        String operationName, Object params[], String signature[])
    throws MBeanException, ReflectionException {

    // Check operationName to avoid NullPointerException later on
    if (operationName == null) {
        throw new RuntimeOperationsException(
            new IllegalArgumentException(
                "Operation name cannot be null"),
            "Cannot invoke a null operation in " + dClassName);
    }

    // Call the corresponding operation for a recognized name
    if (operationName.equals("reset")){
        // this code is specific to the internal "reset" method:
        reset();     // no parameters to check
        return null; // and no return value
    } else {
        // unrecognized operation name:
        throw new ReflectionException(
            new NoSuchMethodException(operationName),
            "Cannot find the operation " + operationName +
                " in " + dClassName);
    }
}

// internal variable
private int         nbResets = 0;

// internal method for implementing the reset operation
public void reset() {
       AttributeChangeNotification acn =
            new AttributeChangeNotification(this,
                                            0,
                                            0,
                                            "NbChanges reset",
```

EXAMPLE 2–6 Implementation of the `invoke` Method *(Continued)*

```
                                           "NbChanges",
                                           "Integer",
                                           new Integer(nbChanges),
                                           new Integer(0));
        state = "initial state";
        nbChanges = 0;
        nbResets++;
        sendNotification(acn);
}
```

As it is written, the `SimpleDynamic` MBean correctly provides a description of its management interface and implements its attributes and operations. However, this example demonstrates the need for a strict coherence between what is exposed by the `getMBeanInfo` method and what can be accessed through the generic getters, setters, and invoker.

A dynamic MBean whose `getMBeanInfo` method describes an attribute or operation that cannot be accessed is not compliant with the JMX specification and is technically not a manageable resource. Similarly, a class could make attributes or operations accessible without describing them in the returned `MBeanInfo` object. Since MBeans should raise an exception when an undefined attribute or operation is accessed, this would, again, technically not be a compliant resource.

2.3 Running the Dynamic MBean Example

The *examplesDir*`/current/DynamicMBean` directory contains the `SimpleDynamic.java` file that makes up the MBean. The `DynamicMBean` interface is defined in the `javax.management` package provided in the runtime JAR file (`jmx.jar`) of the Java Dynamic Management Kit (Java DMK). This directory also contains a simple agent application that instantiates this MBean, calls its `getMBeanInfo` method to get its management interface, and manipulates its attributes and operations.

▼ To Run the Dynamic MBean Example

1. **Compile all files in this directory with the `javac` command.**

   For example, on the Solaris platform, type:

   ```
   $ cd examplesDir/current/DynamicMBean/
   $ javac -classpath classpath *.java
   ```

2. **To run the example, start the agent class that will interact with the `SimpleDynamic` MBean:**

   ```
   $ java -classpath classpath DynamicAgent
   ```

3. **Press `Enter` when the application pauses, to step through the example.**

   The agent application handles all input and output in this example and gives a view of the MBean at runtime.

This example demonstrates how the management interface encoded in the `getMBeanInfo` method is made visible in the agent application. We can then see the result of calling the generic getters and setters and the `invoke` method. Finally, the code for filtering attribute and operation errors is exercised, and we see the exceptions from the code samples as they are raised at runtime.

2.3.1 Comparison With the `SimpleStandard` Example

Now that we have implemented both the standard and dynamic types of MBeans, we can compare how they are managed. We intentionally created a dynamic MBean and a standard MBean with the same management interface so that we can do exactly the same operations on them. On the Solaris platform, we can compare the relevant code of the two agent applications with the `diff` utility (your output might vary):

```
$ cd examplesDir/current
$ diff ./StandardMBean/StandardAgent.java ./DynamicMBean/DynamicAgent.java
[...]
41c40
< public class StandardAgent {
---
> public class DynamicAgent {
49c48
<     public StandardAgent() {
---
>     public DynamicAgent() {
77c76
<       StandardAgent agent = new StandardAgent();
---
>       DynamicAgent agent = new DynamicAgent();
88c87
<       echo("\n>>> END of the SimpleStandard example:\n");
---
>       echo("\n>>> END of the SimpleDynamic example:\n");
113c112
<       String mbeanName = "SimpleStandard";
---
>       String mbeanName = "SimpleDynamic";
```

If the two agent classes had the same name, the only programmatic difference would be the following:

```
113c112
<       String mbeanName = "SimpleStandard";
---
>       String mbeanName = "SimpleDynamic";
```

We can see that the only difference between the two example agents handling different types of MBeans is the name of the MBean class that is instantiated. In other words, standard and dynamic MBeans are indistinguishable from the agent's point of view. The JMX architecture enables managers to interact with the attributes and operations of a manageable resource, and the specification of the agent hides any implementation differences between MBeans.

Because we know that the two MBeans are being managed identically, we can also compare their runtime behavior. In doing so, we can draw two conclusions:

- The dynamic MBean was programmed to have the same behavior as the standard MBean; the example output shows that this is indeed the case. Despite the different implementations, the functionality of the resource is the same.
- The only functional difference between the two is that the agent can obtain the self-description strings encoded in the dynamic MBean. Attributes and operations are associated with the explanation that the programmer provides for them.

Note – There is no mechanism to enable a standard MBean to provide a self-description. The MBean server provides a default description string for each feature in a standard MBean, and these descriptions are identical for all standard MBeans. See also `javax.management.StandardMBean`.

2.3.2 Dynamic MBean Execution Time

In the introduction to this chapter we presented two structural advantages of dynamic MBeans, namely the ability to wrap existing code to make it manageable and the ability to provide a self-description of the MBean and its features. Another advantage is that using dynamic MBeans can lead to faster overall execution time.

The performance gain depends on the nature of the MBean and how it is managed in the agent. For example, the `SimpleDynamic` MBean, as it is used, is probably not measurably faster than the `SimpleStandard` example in Chapter 1. When seeking improved performance, there are two situations that must be considered:

- MBean introspection
- Management operations

Because the dynamic MBean provides its own description, the agent does not need to introspect it as it would a standard MBean. Since introspection is done only once by the agent, this is a one-time performance gain during the lifetime of the MBean. In an environment where there are many MBean creations and where MBeans have a short lifetime, a slight performance increase can be measured.

However, the largest performance gain is in the management operations, when calling the getters, setters and invoker. As we shall see in Part II, the agent makes MBeans manageable through generic getters, setters, and invokers. In the case of standard MBeans, the agent must do the computations for resolving attribute and operation names according to the design patterns. Because dynamic MBeans necessarily expose the same generic methods, these are called directly by the agent. When a dynamic MBean has a simple management interface requiring simple programming logic in its generic methods, its implementation can show a better performance than the same functionality in a standard MBean.

CHAPTER 3

Model MBeans

A model MBean is a generic, configurable MBean that applications can use to instrument any resource dynamically. It is a dynamic MBean that has been implemented so that its management interface and its actual resource can be set programmatically. This enables any manager connected to a Java dynamic management agent to instantiate and configure a model MBean dynamically.

Model MBeans enable management applications to make resources manageable at runtime. The managing application must provide a compliant management interface for the model MBean to expose. It must also specify the *target objects* that actually implement the resource. Once it is configured, the model MBean passes any management requests to the target objects and handles the result.

In addition, the model MBean provides a set of mechanisms for how management requests and their results are handled. For example, caching can be performed on attribute values. The management interface of a model MBean is augmented by *descriptors* that contain attributes for controlling these mechanisms.

The code samples in this chapter are taken from the files in the `ModelMBean` example directory in the main *examplesDir*/`current` directory (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "3.1 `RequiredModelMBean` Class" on page 56 gives an overview of model MBeans.
- "3.2 Model MBean Metadata" on page 56 explains how to describe a model MBean's management interface.
- "3.3 Target Object(s)" on page 58 describes how a model MBean is associated with its resource.
- "3.4 Creating a Model MBean" on page 61 shows how to instantiate and register a model MBean.
- "3.5 Running the Model MBean Example" on page 62 shows how an agent interacts with a model MBean.

3.1 RequiredModelMBean Class

The required model MBean is mandated by the Java Management Extensions (JMX) specification for all compliant implementations. It is a dynamic MBean that lacks any predefined management interface. It contains a generic implementation that transmits management requests on its management interface to the target objects that define its managed resource.

The name of the required model MBean class is the same for all JMX-compliant implementations. Its full package and class name is `javax.management.modelmbean.RequiredModelMBean`. By instantiating this class, any application can use model MBeans.

In order to be useful, the instance of the required model MBean must be given a management interface and the target object of the management resource. In addition, the model MBean metadata must contain descriptors for configuring how the model MBean will respond to management requests. We will cover these steps in subsequent sections.

The MBean server does not make any special distinction for model MBeans. Internally they are treated as the dynamic MBeans that they are, and all of the model MBean's internal mechanisms and configurations are completely transparent to a management application. Like all other managed resources in the MBean server, the resources available through the model MBean can only be accessed through the attributes and operations defined in the management interface.

3.2 Model MBean Metadata

The metadata of an MBean is the description of its management interface. The metadata of the model MBean is described by an instance of the `ModelMBeanInfo` class, which extends the `MBeanInfo` class.

Like all other MBeans, the metadata of a model MBean contains the list of attributes, operations, constructors, and notifications of the management interface. Model MBeans also describe their target object and their policies for accessing the target object. This information is contained in an object called a *descriptor*, defined by the `Descriptor` interface and implemented in the `DescriptorSupport` class.

There is one overall descriptor for a model MBean instance and one descriptor for each element of the management interface, that is for each attribute, operation, constructor, and notification. Descriptors are stored in the metadata object. As defined by the JMX specification, all classes for describing elements are extended so that they contain a descriptor. For example, the `ModelMBeanAttributeInfo` class extends the `MBeanAttributeInfo` class and defines the methods `getDescriptor` and `setDescriptor`.

A descriptor is a set of named field and value pairs. Each type of metadata element has a defined set of fields that are mandatory, and users are free to add others. The field names reflect the policies for accessing target objects, and their values determine the behavior. For example, the descriptor of an attribute contains the fields `currencyTimeLimit` and `lastUpdatedTimeStamp` that are used by the internal caching mechanism when performing a get or set operation.

In this way, model MBeans are manageable in the same way as any other MBean, but applications that are aware of model MBeans can interact with the additional features they provide. The JMX specification defines the names of all required descriptor fields for each of the metadata elements and for the overall descriptor. The field names are also described in the API documentation generated by the Javadoc tool for the `ModelMBean*Info` classes.

In Example 3–1 , the application defines a subroutine to build all descriptors and metadata objects needed to define the management interface of the model MBean.

EXAMPLE 3–1 Defining Descriptors and `MBeanInfo` Objects

```
private void buildDynamicMBeanInfo(
                 ObjectName inMbeanObjectName, String inMbeanName) {
  try {

    // Create the descriptor and ModelMBeanAttributeInfo
    // for the 1st attribute
    //
    mmbDesc = new DescriptorSupport( new String[]
                    { ("name="+inMbeanObjectName),
                      "descriptorType=mbean",
                      ("displayName="+inMbeanName),
                      "log=T",
                      "logfile=jmxmain.log",
                      "currencyTimeLimit=5"});
    Descriptor stateDesc = new DescriptorSupport();
    stateDesc.setField("name","State");
    stateDesc.setField("descriptorType","attribute");
    stateDesc.setField("displayName","MyState");
    stateDesc.setField("getMethod","getState");
    stateDesc.setField("setMethod","setState");
    stateDesc.setField("currencyTimeLimit","20");

    dAttributes[0] = new ModelMBeanAttributeInfo(
                             "State",
                             "java.lang.String",
                             "State: state string.",
                             true,
                             true,
                             false,
```

EXAMPLE 3–1 Defining Descriptors and `MBeanInfo` Objects *(Continued)*

```
                            stateDesc);

    [...] // create descriptors and ModelMBean*Info for
          // all attributes, operations, constructors
          // and notifications


    dMBeanInfo = new ModelMBeanInfoSupport(
                        dClassName,
                        dDescription,
                        dAttributes,
                        dConstructors,
                        dOperations,
                        dNotifications);

    dMBeanInfo.setMBeanDescriptor(mmbDesc);

  } catch (Exception e) {
    echo("\nException in buildModelMBeanInfo : " +
          e.getMessage());
    e.printStackTrace();
  }
    // Create the ModelMBeanInfo for the whole MBean
    //
    private String dClassName = "TestBean";
    private String dDescription =
    "Simple implementation of a test app Bean.";
}
```

3.3 Target Object(s)

The object instance that actually embodies the behavior of the managed resource is called the *target object*. The last step in creating a model MBean is to give the MBean skeleton and its defined management interface a reference to the target object. Thereafter, the model MBean can handle management requests, forward them to the target object, and handle the response.

Example 3–2 implements the `TestBean` class that is the simple managed resource in our example. Its methods provide the implementation for two attributes and one operation.

EXAMPLE 3–2 Implementing the Managed Resource

```
public class TestBean
   implements java.io.Serializable
{
```

EXAMPLE 3–2 Implementing the Managed Resource *(Continued)*

```
    // Constructor
    //
    public TestBean() {
        echo("\n\tTestBean Constructor Invoked: State " +
             state + " nbChanges: " + nbChanges +
             " nbResets: " + nbResets);

    }

    // Getter and setter for the "State" attribute
    //
    public String getState() {
        echo("\n\tTestBean: getState invoked: " + state);
        return state;
    }

    public void setState(String s) {
        state = s;
        nbChanges++;
        echo("\n\tTestBean: setState to " + state +
             " nbChanges: " + nbChanges);
    }


    // Getter for the read-only "NbChanges" attribute
    //
    public Integer getNbChanges() {
        echo("\n\tTestBean: getNbChanges invoked: " + nbChanges);
        return new Integer(nbChanges);
    }

    // Method of the "Reset" operation
    //
    public void reset() {
        echo("\n\tTestBean: reset invoked ");
        state = "reset initial state";
        nbChanges = 0;
        nbResets++;
    }

    // Other public method; looks like a getter,
    // but no NbResets attribute is defined in
    // the management interface of the model MBean
    //
    public Integer getNbResets() {
        echo("\n\tTestBean: getNbResets invoked: " + nbResets);
        return new Integer(nbResets);
    }

    // Internals
    //
    private void echo(String outstr) {
```

EXAMPLE 3–2 Implementing the Managed Resource *(Continued)*

```
        System.out.println(outstr);
    }

    private String  state = "initial state";
    private int     nbChanges = 0;
    private int     nbResets = 0;

}
```

By default, the model MBean handles a managed resource that is contained in one object instance. This target is specified through the `setManagedResource` method defined by the `ModelMBean` interface. The resource can encompass several programmatic objects because individual attributes or operations can be handled by different target objects. This behavior is configured through the optional `targetObject` and `targetType` descriptor fields of each attribute or operation.

In Example 3–3, one of the operations is handled by an instance of the `TestBeanFriend` class. In the definition of this operation's descriptor, we set this instance as the target object. We then create the operation's `ModelMBeanOperationInfo` with this descriptor and add it to the list of operations in the metadata for our model MBean.

EXAMPLE 3–3 Setting Other Target Objects

```
MBeanParameterInfo[] params = null;

[...]

Descriptor getNbResetsDesc = new DescriptorSupport(new String[]
                                    { "name=getNbResets",
                                      "class=TestBeanFriend",
                                      "descriptorType=operation",
                                      "role=operation"});

TestBeanFriend tbf = new TestBeanFriend();
getNbResetsDesc.setField("targetObject",tbf);
getNbResetsDesc.setField("targetType","objectReference");

dOperations[1] = new ModelMBeanOperationInfo(
                         "getNbResets",
                "getNbResets(): get number of resets performed",
                         params ,
                         "java.lang.Integer",
                         MBeanOperationInfo.INFO,
                         getNbResetsDesc);
```

3.4 Creating a Model MBean

To ensure coherence in an agent application, you should define the target object of an MBean before you expose it for management. This implies that you should call the `setManagedResource` method before registering the model MBean in the MBean server.

Example 3–4 shows how our application creates the model MBean. First it calls the subroutine given in Example 3–1 to build the descriptors and management interface of the model MBean. Then it instantiates the required model MBean class with this metadata. Finally, it creates and sets the managed resource object before registering the model MBean.

EXAMPLE 3–4 Setting the Default Target Object

```
ObjectName mbeanObjectName = null;
String domain = server.getDefaultDomain();
String mbeanName = "ModelSample";

try
{
     mbeanObjectName = new ObjectName(
                               domain + ":type=" + mbeanName);
} catch (MalformedObjectNameException e) {
     e.printStackTrace();
     System.exit(1);
}
[...]

// Create the descriptors and ModelMBean*Info objects
// of the management interface
//
buildDynamicMBeanInfo( mbeanObjectName, mbeanName );

try {
    RequiredModelMBean modelmbean =
        new RequiredModelMBean( dMBeanInfo );
    // Set the managed resource for the ModelMBean instance
    modelmbean.setManagedResource( new TestBean(), "objectReference");

    // register the model MBean in the MBean server
    server.registerMBean( modelmbean, mbeanObjectName );

} catch (Exception e) {
     echo("\t!!! ModelAgent: Could not create the " + mbeanName +
                 " MBean !!!");
     e.printStackTrace();
     System.exit(1);
}
```

The model MBean is now available for management operations and remote requests, just like any other registered MBean.

3.5 Running the Model MBean Example

The *examplesDir*/current/ModelMBean directory contains the TestBean.java file that is the target object of the sample model MBean. This directory also contains a simple notification listener class and the agent application, ModelAgent, which instantiates, configures, and manages a model MBean.

The model MBean itself is given by the RequiredModelMBean class defined in the javax.management.modelmbean package provided in the runtime JAR file (jmx.jar) of the Java Dynamic Management Kit (Java DMK).

▼ To Run the Model MBean Example

1. **Compile all files in this directory with the javac command.**

   For example, on the Solaris platform, type:

   ```
   $ cd examplesDir/current/ModelMBean/
   $ javac -classpath classpath *.java
   ```

2. **To run the example, start the agent class with the following command:**

   ```
   $ java -classpath classpath ModelAgent
   ```

3. **Press Enter when the application pauses to step through the example.**

   The agent application handles all input and output in this example and gives a view of the MBean at runtime.

We can then see the result of managing the resource through its exposed attributes and operations. The agent also instantiates and registers a listener object for attribute change notifications sent by the model MBean. You can see the output of this listener whenever it receives a notification, after the application has called one of the attribute setters.

CHAPTER 4

Open MBeans

Open MBeans are dynamic MBeans, with specific constraints on their data types, that allow management applications and their human administrators to understand and use new managed objects as they are discovered at runtime. Open MBeans provide a flexible means of instrumenting resources which need to be *open* to a wide range of applications compliant with the Java Management Extensions (JMX) specification.

To provide its own description to management applications, an open MBean must be a dynamic MBean, with the same behavior and functionality as a dynamic MBean. Thus it implements the `DynamicMBean` interface and no corresponding open MBean interface is required. The *open* functionality of open MBeans is obtained by providing descriptively rich metadata and by using exclusively certain predefined data types in the management interface.

Because open MBeans build their data types from a predefined set of Java classes, a management application can manage an agent that uses open MBeans, without having to load Java classes specific to that agent. Furthermore, a JMX connector or adaptor only needs to be able to transport classes from the predefined set, not arbitrary Java classes. This means, for example, that a connector could use eXtensible Markup Language (XML) as its transport by defining an XML schema that covers the MBean server operations and the predefined open MBean data types. It also means that a management application could run in a language other than Java.

The code samples in this chapter are taken from the files in the `OpenMBean` and `OpenMBean2` example directories located in *examplesDir*`/current` (see “Directories and Classpath” in the Preface).

This chapter covers the following topics:

- “4.1 Open MBean Data Types” on page 64 describes the set of predefined types that must be used by open MBeans
- “4.2 Open Data Instances” on page 65 explains the implementation of the open MBean data types
- “4.3 Open MBean Metadata Classes” on page 67 presents the set of metadata classes that are used specifically to describe open MBeans

- "4.4 Running the Open MBean Examples" on page 68 provides information on running the open MBean examples

4.1 Open MBean Data Types

Open MBeans refer exclusively to a limited, predefined set of data types, which can be combined into compound types.

4.1.1 Supported Data Types

All open MBean attributes, method return values, and method arguments must be limited to the set of open MBean data types listed in this section. This set is defined as: the wrapper objects that correspond to the Java primitive types (such as `Integer`, `Long`, `Boolean`), `String`, `CompositeData`, `TabularData`, and `ObjectName`.

In addition, any array of the open MBean data types may be used in open MBeans. A special class, `javax.management.openmbean.ArrayType` is used to represent the definition of single or multi-dimensional arrays in open MBeans.

The following list specifies all data types that are allowed as scalars or as any-dimensional arrays in open MBeans:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `javax.management.ObjectName`
- `javax.management.openmbean.CompositeData` (interface)
- `javax.management.openmbean.TabularData` (interface)

In addition the `java.lang.Void` type may be used, but only as the return type of an operation.

4.1.2 Type Descriptor Classes

Open types are descriptor classes that describe the open MBean data types. To be able to manipulate the open MBean data types, management applications must be able to identify them. While primitive types are given by their wrapper class names, and arrays can be represented in a standard way, the complex data types need more structure than a flat string to represent their contents. Therefore open MBeans rely on description classes for all the open MBean data types, including special structures for describing complex data.

The abstract `OpenType` class is the superclass for the following specialized open type classes:

- `SimpleType`, which describes the primitive types and the `ObjectName` class when they are not used in arrays
- `CompositeType`, which describes composite data types
- `TabularType`, which describes tabular data types
- `ArrayType`, which describes any-dimensional arrays of any of the preceding types

The `CompositeType`, `TabularType`, and `ArrayType` classes are recursive, that is, they can contain instances of other open types.

4.2 Open Data Instances

All of the wrapper classes for the primitive types are defined and implemented in all Java virtual machines, as is `java.lang.String`. The `ObjectName` class is provided by the implementation of the JMX specification. You can use the `CompositeData` and `TabularData` interfaces to define aggregates of the open MBean data types and provide a mechanism for expressing complex data objects in a consistent manner.

All open data instances corresponding to the open MBean types defined in “4.1.1 Supported Data Types” on page 64, except `CompositeData` and `TabularData` instances, are available through the classes of the Java 2 Platform, Standard Edition (J2SE) or JMX specification.

4.2.1 `CompositeData` and `TabularData` Instances

The `CompositeData` and `TabularData` interfaces represent complex data types. The JMX specification now includes a default implementation of these interfaces, using the `CompositeDataSupport` and `TabularDataSupport` classes respectively.

The CompositeData and TabularData interfaces and implementations provide some semantic structure to build aggregates from the open MBean data types. An implementation of the CompositeData interface is equivalent to a map with a predefined list of keys: values are retrieved by giving the name of the desired data item. Other terms commonly used for this sort of data type are *record* or *struct*.

An instance of a TabularData object contains a set of CompositeData instances. Each CompositeData instance in a TabularData instance is indexed by a unique key derived from its values, as described in “4.2.1.2 TabularDataSupport Class” on page 66.

A CompositeData object is immutable once instantiated: you cannot add an item to it and you cannot change the value of an existing item. TabularData instances are modifiable: rows can be added or removed from existing instances.

4.2.1.1 CompositeDataSupport Class

A CompositeData object associates string keys with the values of each data item. The CompositeDataSupport class defines an immutable map with an arbitrary number of entries, called data items, which can be of any type. To comply with the design patterns for open MBeans, all data items must have a type among the set of open MBean data types, including other CompositeData types.

When instantiating the CompositeDataSupport class, you must provide the description of the composite data object in a CompositeType object (see “4.1.2 Type Descriptor Classes” on page 65). All the items provided through the constructor must match this description. Because the composite object is immutable, all items must be provided at instantiation time, and therefore the constructor can verify that the items match the description. The getOpenType method returns this description so that other objects which interact with a CompositeData object can know its structure.

4.2.1.2 TabularDataSupport Class

The TabularDataSupport class defines a table structure with an arbitrary number of rows which can be indexed by any number of columns. Each row is a CompositeData object, and all rows must have the same composite data description. The columns of the table are headed by the names of the data items which make up the uniform CompositeData rows. The constructor and the methods for adding rows verify that all rows are described by equal CompositeData instances.

The index consists of a subset of the data items in the common composite data structure. This subset must be a key which uniquely identifies each row of the table. When the table is instantiated, or when a row is added, the methods of this class ensure that the index can uniquely identify all rows. Often the index will be a single column, for instance a column containing unique strings or integers.

Both the composite data description of all rows and the list of items which form the index are given by the table description returned by the getOpenType method. This method defined in the TabularData interface returns the TabularType object which describes the table (see "4.1.2 Type Descriptor Classes" on page 65).

The access methods of the TabularData class take an array of objects representing a key value which indexes one row and returns the CompositeData instance which makes up the designated row. A row of the table can also be removed by providing its key value. All rows of the table can also be retrieved in an enumeration.

4.3 Open MBean Metadata Classes

To distinguish open MBeans from other MBeans, the JMX specification provides a set of metadata classes which are used specifically to describe open MBeans.

The following interfaces in the javax.management.openmbean package define the management interface of an open MBean:

- OpenMBeanInfo lists the attributes, operations, constructors and notifications
- OpenMBeanOperationInfo describes the method of an operation
- OpenMBeanConstructorInfo describes a constructor
- OpenMBeanParameterInfo describes a method parameter
- OpenMBeanAttributeInfo describes an attribute

For each of these interfaces, a support class provides an implementation. Each of these classes describes a category of components in an open MBean. However, open MBeans do not have a specific metadata object for notifications; they use the MBeanNotificationInfo class.

Open MBeans provide a universal means of exchanging management functionality and consequently their description must be explicit enough for any user to understand. All of the OpenMBean*Info metadata classes inherit the getDescription method which should return a non-empty string. Each component of an open MBean must use this method to provide descriptions that are suitable for displaying in a graphical user interface.

Only the OpenMBeanOperationInfo specifies the getImpact method. Instances of OpenMBeanOperationInfo.getImpact must return one of the following constant values:

- ACTION
- INFO
- ACTION_INFO

The value UNKNOWN cannot be used.

4.4 Running the Open MBean Examples

The examples directory provides two examples that demonstrate how to implement and manage an open MBean.

4.4.1 Open MBean Example 1

In the first open MBean example, we implement an open MBean and manage it through a simple JMX agent application. We develop a sample open MBean that uses some of the open data types and correctly exposes its management interface at runtime through the `OpenMBean*Info` classes. We then develop a simple JMX agent for exercising the open MBean, which involves:

- Initializing the MBean server
- Creating and adding the sample open MBean to the MBean server
- Getting and displaying the open MBean management information
- Calling some operations on the MBean

The *examplesDir*`/current/OpenMBean` directory contains the `SampleOpenMBean.java` file, which is an open MBean, and the `OpenAgent.java` file which is a simple JMX agent used to interact with the open MBean.

▼ To Run Open MBean Example 1

1. **Compile all files in the *examplesDir*`/current/OpenMBean` directory with the `javac` command.**

   For example, on the Solaris platform, type:

   ```
   $ cd examplesDir/current/OpenMBean/
   $ javac -classpath classpath *.java
   ```

2. **Run the agent class that interacts with the open MBean:**

   ```
   $ java -classpath classpath OpenAgent
   ```

3. **Press `Enter` when the application pauses, to step through the example.**

   You interact with the agent through the standard input and output in the window where it was launched. The `OpenAgent` displays information about each management step and waits for your input before continuing.

4.4.2 Open MBean Example 2

In the second open MBean example, we implement an open MBean and manage it through a simple JMX manager application. Although it is helpful to run the first open MBean example before this example, it is not obligatory.

We develop a sample open MBean that uses some of the open data types and correctly exposes its management interface at runtime through the `OpenMBean*Info` classes. See the `CarOpenMBean.html` file in the *examplesDir*`/current/OpenMBean2/docs` directory for more detailed information on the data structure of this example.

We then develop a simple manager for exercising the open MBean, which involves:

- Creating and adding the sample open MBean to the MBean server
- Getting and displaying the open MBean management information
- Invoking some operations on the MBean

The *examplesDir*`/current/OpenMBean2` directory contains the following source files:

- `CarOpenMBean.java`, which is an open MBean.
- `Agent.java`, which is a minimal agent comprising an MBean server, an HTML adaptor and a JMXMP connector.
- `Manager.java`, which is a client application that runs the example scenario.

▼ To Run Open MBean Example 2

1. **Compile all files in the *examplesDir*`/current/OpenMBean2` directory with the `javac` command.**

   For example, on the Solaris platform, type:

   ```
   $ cd examplesDir/current/OpenMBean2/
   $ javac -classpath classpath *.java
   ```

2. **Run the agent class that interacts with the open MBean:**

   ```
   $ java -classpath classpath Agent
   ```

3. **Open a second terminal window and change to the `OpenMBean2` example directory:**

   For example, on the Solaris platform, type:

   ```
   $ cd examplesDir/current/OpenMBean2/
   ```

4. **Run the manager class:**

   ```
   $ java -classpath classpath Manager
   ```

   Interact with the manager through the standard input and output in the window where it was started. The `Manager` class displays information about each management step and waits for your input before continuing.

PART II

Agent Applications

The *agent* is the central component of the Java Management Extensions (JMX) management architecture. An agent contains MBeans and hides their implementation behind a standardized management interface, lets management applications connect and interact with all MBeans, and provides filtering for handling large numbers of MBeans. JMX agents are dynamic because resources can be added and removed, connections can be closed and reopened with a different protocol, and services can be added and removed as management needs evolve.

In Part I, we saw how to represent resources as MBeans. However, MBeans can represent any object whose functionality you need to manage. In particular, management services and remote connectivity are handled by objects that are also MBeans. This creates a homogeneous model where an agent is a framework containing different kinds of MBeans and enabling them to interact.

The main component of an agent is the MBean server. It registers all MBeans in the agent and exposes them for management. The role of the MBean server is to be the liaison between any object available to be managed and any object with a management request. Usually resource MBeans are managed either by remote applications through connectivity MBeans or by local management service MBeans. This model allows a management service itself to be managed. Connectors and services can also be created, modified, or removed dynamically.

This part focuses on the functionality of the MBean server and the Java objects which are needed to create a simple agent. Details about programming managers and about using connectors and services will be covered in Part III.

This part contains the following chapters:

- Chapter 5 describes the *base agent*. The base agent shows how to manipulate MBeans programmatically through the instance of the MBean server. This chapter covers the different ways of creating and interacting with MBeans in the MBean server. This chapter also describes how to process the metadata objects that represent MBean information.

- Chapter 6 introduces the *HTML protocol adaptor*, which gives us a management view of the MBeans in an agent through a web browser. It lets us create MBeans, update their attributes, invoke their operations, and remove them dynamically in a running agent.
- Chapter 7 shows how *MBean interceptors* can be used to modify the behavior of the MBean server.
- Chapter 8 demonstrates the fundamentals of *notification broadcasters and listeners* where both are within the same agent. Since the MBean server delegate is a broadcaster, the example shows how to register a listener to process its events. The example also shows how to listen for attribute change notifications, a subclass of regular notifications that is defined by the JMX specification.

CHAPTER 5

Base Agent

An agent application is a program written in the Java language that contains an MBean server and a means of accessing its functionality. The base agent demonstrates the programming details of writing an agent application. We will cover the MBean server classes, how to reference MBeans, how to access the MBean server, use it to create MBeans, and then interact with those MBeans.

Interacting programmatically with the MBean server gives you more flexibility in your agent application. This chapter presents the MBean server and covers the three main ways to use it to create MBeans, how to interact with these MBeans, and how to unregister them.

The program listings in this tutorial show only functional code: comments and output statements have been modified or removed to conserve space. However, all management functionality has been retained for the various demonstrations. The complete source code is available in the `BaseAgent` and `StandardMBean` example directories located in *examplesDir*/`current` (see *Directories and Classpath* in the Preface).

This chapter covers the following topics:

- "5.1 MBean Server Classes" on page 74 presents the functionality of the MBean server.
- "5.2 Referencing MBeans" on page 75 demonstrates how to reference MBeans.
- "5.3 Agent Application" on page 77 shows how to launch the base agent programmatically.
- "5.4 Creating an MBean Using the `registerMBean` Method" on page 78 relies on the MBean server's `registerMBean` method after instantiating the MBean class.
- "5.5 Creating an MBean Using the `createMBean` Method" on page 80 relies on the MBean server's `createMBean` method to instantiate and register an MBean in one step.
- "5.6 Creating an MBean Using the `instantiate` Method" on page 80 relies on the MBean server's `instantiate` and `register` methods.

- "5.7 Managing MBeans" on page 81 demonstrates the same management operations we performed using the HTML protocol adaptor.
- "5.8 Filtering MBeans With the Base Agent" on page 83 shows how to get various lists of MBeans from the MBean server.
- "5.9 Running the Base Agent Example" on page 84 demonstrates its runtime behavior.

5.1 MBean Server Classes

Before writing an agent application, it is important to understand the functionality of the MBean server. It is actually an interface and a *factory* object defined by the agent specification level of the JMX specification. The Java DMK provides an implementation of this interface and factory. The factory object finds or creates the MBean server instance, making it possible to substitute different implementations of the MBean server.

5.1.1 `MBeanServer` Interface

The `MBeanServer` interface is an extension of the `MBeanServerConnection` interface, and represents the agent-side interface for interacting with MBeans.

The specification of the interface defines all operations that can be applied to resources and other agent objects through the MBean server. Its methods can be divided into three main groups:

- Methods for controlling MBean instances:
  - `createMBean`, or `instantiate` and `registerMBean` add a new MBean to the agent
  - `unregisterMBean` removes an MBean from the agent
  - `isRegistered` and `getObjectInstance` associate the class name with the MBean's management name
  - `addNotificationListener` and `removeNotificationListener` control event listeners for a particular MBean
  - `getClassLoader` is used to download new MBean classes
- Methods for accessing MBean attributes and operations. These methods are identical to those presented in "2.1.1 `DynamicMBean` Interface" on page 42, except they all have an extra parameter for specifying the target MBean:
  - `getMBeanInfo`
  - `getAttribute` and `getAttributes`

  - `setAttribute` and `setAttributes`
  - `invoke`
- Methods for managing the agent as a whole:
  - `getDefaultDomain` (domains are a way of grouping MBeans in the agent)
  - `getMBeanCount` counts all MBeans in an agent
  - `queryMBeans` and `queryNames` to find MBeans by name or by value

5.1.2 MBean Server Implementation, Builder and Factory

The `MBeanServer` implementation class in Java DMK implements the `JdmkMBeanServer` and `MBServerInt` interfaces if the property `javax.management.builder.initial` has been set to use `com.sun.jdmk.JdmkMBeanServerBuilder`. This implementation of `MBeanServer` wraps the JMX MBean server.

The older `MBeanServerInt` interface and the `MBeanServerImpl` class are still implemented for reasons of backwards compatibility, but in Java DMK 5.1, `JdmkMBeanServer` is the prefered interface.

The `MBeanServerFactory` makes the agent application independent of the MBean server implementation, and thus allows applications to provide custom `MBeanServer` implementations. It resides in the Java virtual machine and centralizes all MBean server instantiation. The `MBeanServerFactory` class provides two static methods:

- `createMBeanServer` returns a new MBean server instance.
- `findMBeanServer` returns a list of MBean servers in the Java virtual machine.

You must use the `MBeanServerFactory` classes to create an MBean server so that other objects can obtain its reference by calling the `findMBeanServer` method. This method enables dynamically loaded objects to find the MBean server in an agent that has already been started.

5.2 Referencing MBeans

Most agent applications interact with MBeans through the MBean server. It is possible for an object to instantiate an MBean class itself, which it can then register in the MBean server. In this case, it keeps a programmatic reference to the MBean instance. All other objects can only interact with the MBean through its management interface exposed by the MBean server.

In particular, service MBeans and connectivity MBeans rely solely on the MBean server to access resources. The MBean server centralizes the access to all MBeans. It unburdens all other objects from having to keep numerous object references. To ensure this function, the MBean server relies on object names to identify MBean instances uniquely.

5.2.1 Object Names

Each MBean object registered in the MBean server is identified by an object name. The same MBean class can have multiple instances, but each must have a unique name. The `ObjectName` class encapsulates an object name composed of a domain name and a set of key properties. The object name can be represented as a string in the following format:

DomainName`:`*property=value*`[,`*property2=value2*`]*`

The *DomainName*, the *property* and *value* can be any alphanumeric string, as long as it does not contain any of the following characters: `: , = * ?`. All elements of the object name are treated as literal strings, meaning that they are case sensitive.

5.2.1.1 MBean Domains

A *domain* is an abstract category that can be used to group MBeans arbitrarily. The MBean server lets you search easily for all MBeans with the same domain. For example, all connectivity MBeans in the minimal server could have been registered into a domain called `Communications`.

Since all object names must have a domain, the MBeans in an MBean server necessarily define at least one domain. When the domain name is not important, the MBean server provides a default domain name that you can use. By default, it is called the `DefaultDomain`, but you can specify a different default domain name when creating the MBean server from its factory.

5.2.1.2 Key Properties

A *key* is a property-value pair that can also have any meaning that you assign to it. An object name must have at least one key. Keys and their values are independent of the MBean's attributes. The object name is a static identifier that identifies the MBean, whereas attributes are the exposed, runtime values of the corresponding resource. Keys are not positional and can be given in any order to identify an MBean.

Keys provide the specificity for identifying a unique MBean instance. For example, an object name for the HTML protocol adaptor MBean might be: `Communications:protocol=html,port=8082`, assuming the port will not change.

5.2.1.3 Usage of Object Names

All MBeans must be given an object name that is unique. It can be assigned by the MBean's preregistration method, if the MBean supports preregistration (see the Javadoc API of the `MBeanRegistration` interface). Or it can be assigned by the object that creates or registers the MBean, which overrides the one given during preregistration. However, if neither of these assign an object name, the MBean server will not create the MBean and will raise an exception. Once an MBean is instantiated and registered, its assigned object name cannot be modified.

You can encode any meaning into the domain and key strings. The MBean server handles them as literal strings. The contents of the object name should be determined by your management needs. Keys can be meaningless serial numbers if MBeans are always handled programmatically. On the other hand, the keys can be human-readable to simplify their translation to the graphical user interface of a management application. With the HTML protocol adaptor, object names are displayed directly to the user.

5.2.2 `ObjectInstance` of an MBean

An *object instance* represents the complete reference of an MBean in the MBean server. It contains the MBean's object name and its Java class name. Object instances are returned by the MBean server when an MBean is created or in response to queries about MBeans. Since the object name and class name cannot change over the life of a given MBean, its returned object instance will always have the same value.

You cannot modify the class or object name in an object instance. This information is read-only. The object name is used to refer to the MBean instance in any management operation through the MBean server. The class name can be used to instantiate similar MBeans or introspect characteristics of the class.

5.3 Agent Application

The base agent is a standalone application with a `main` method, but it also has a constructor so that it can be instantiated dynamically by another class.

EXAMPLE 5–1 Constructor for the Base Agent

```
public BaseAgent() {

    echo("\n\tInstantiating the MBean server of this agent...");
    mbs = MBeanServerFactory.createMBeanServer();
```

EXAMPLE 5–1 Constructor for the Base Agent *(Continued)*

```
    // Retrieves ID of the MBean server from
    // the associated MBean server delegate
    //
    echo("\n\tGetting the ID of the MBean server from the " +
             "associated MBean server delegate ...");
    try {
         echo("\tID = " +
             mbs.getAttribute(new ObjectName(ServiceName.DELEGATE),
                              "MBeanServerId"));
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
```

The MBean server is created through the static `MBeanServerFactory` object, and we store its object reference. Its true type is hidden by the factory object, that casts the returned object as an `MBeanServer` interface. The MBean server is the only functional class referenced directly in this application.

After the MBean server is initialized, we will create three communication MBeans.

5.4 Creating an MBean Using the `registerMBean` Method

The methods of the MBean server enable you to create an MBean in three different ways. The base agent demonstrates all three ways, and we will discuss the advantages of each.

One way of creating an MBean consists of first instantiating its class and then registering this instance in the MBean server. Registration is the internal process of the MBean server that takes a manageable resource's MBean instance and exposes it for management.

Bold text in Example 5–2 and the following code samples highlights the important statements that vary between the three methods.

EXAMPLE 5–2 Creating an MBean Using the `registerMBean` Method

```
// instantiate the HTML protocol adaptor object to use the default port
CommunicatorServer htmlAdaptor = new HtmlAdaptorServer();

try {
    // We let the HTML adaptor provide a default object name
```

EXAMPLE 5–2 Creating an MBean Using the `registerMBean` Method *(Continued)*

```
    ObjectInstance htmlAdaptorInstance =
        mbs.registerMBean(htmlAdaptor, null);
    echo("\tCLASS NAME  = " + htmlAdaptorInstance.getClassName());
    echo("\tOBJECT NAME = " +
    htmlAdaptorInstance.getObjectName().toString());
} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}
htmlAdaptor.start();

// Waiting to leave starting state...
while (htmlAdaptor.getState() == CommunicatorServer.STARTING) {
    sleep(1000);
}
echo("\tSTATE = " + htmlAdaptor.getStateString());
[...]
```

In this first case, we instantiate the `HtmlAdaptorServer` class and keep a reference to this object. We then pass it to the `registerMBean` method of the MBean server to make our instance manageable in the agent. During the registration, the instance can also obtain a reference to the MBean server, something it requires to function as a protocol adaptor.

The HTML adaptor gives itself a default name in the default domain. Its Javadoc API confirms this, so we can safely let it provide a default name. We print the object name in the object instance returned by the registration to confirm that the default was used.

Once the MBean is registered, we can perform management operations on it. Because we kept a reference to the instance, we do not need to go through the MBean server to manage this MBean. This enables us to call the `start` and `getStateString` methods directly. The fact that these methods are publicly exposed is particular to the implementation. The HTML adaptor is a dynamic MBean, so without any prior knowledge of the class, we would have to go through its `DynamicMBean` interface.

In a standard MBean you would call the implementation of its management interface directly. Because the HTML adaptor is a dynamic MBean, the `start` method is just a shortcut for the `start` operation. For example, we could start the adaptor and get its `StateString` attribute with the following calls:

```
htmlAdaptor.invoke("start", new Object[0], new String[0]);
echo("STATE = " + (String)htmlAdaptor.getAttribute("StateString"));
```

This type of shortcut is not specified by the Java Management Extensions (JMX) specification, nor is its functionality necessarily identical to that of the `start` operation exposed for management. In the case of the HTML adaptor, its Javadoc API confirms that it is identical, and in other cases, it is up to the MBean programmer to guarantee this functionality if it is offered.

5.5 Creating an MBean Using the `createMBean` Method

A second way to create an MBean is the single `createMBean` method of the MBean server. In this case, the MBean server instantiates the class and registers it all at once. As a result, the caller never has a direct reference to the new object. This is not demonstrated in the `BaseAgent` example, but the advantage of this method for creating MBeans is that the instantiation and registration are done in one call. In addition, if we have registered any class loaders in the MBean server, they will automatically be used if the class is not available locally. See Chapter 12 for more information on class loading.

The major difference is that we no longer have a reference to our MBean instance. The object instance that was only used for display purposes in the previous example now gives us the only reference we have on the MBean: its object name.

One disadvantage of this method is that all management of the new MBean must now be done through the MBean server. For the attributes of the MBean, we need to call the generic getter and setter of the MBean server, and for the operations we need to call the `invoke` method. When the agent needs to access the MBean, having to go through the MBean server adds some complexity to the code. However, it does not rely on any shortcuts provided by the MBean, making the code more portable and reusable.

The `createMBean` method is ideal for quickly starting new MBeans that the agent application does not need to manipulate. In just one call, the new objects are instantiated and exposed for management.

5.6 Creating an MBean Using the `instantiate` Method

The last way of creating an MBean relies on the `instantiate` method of the MBean server. In addition, we can use a nondefault constructor to instantiate the class with a different behavior. This method is not demonstrated in the `BaseAgent` example, but as in Example 5–2, we instantiate and register the MBean in separate steps.

First, you instantiate the class using the `instantiate` method of the MBean server. This method lets you specify the constructor you want use when instantiating. Note that we could also have specified a constructor to the `createMBean` method in the previous example.

To specify a constructor, you must give an array of objects for the parameters and an array of strings that defines the signature. If these arrays are empty or null, the MBean server will try to use the default no-parameter constructor. If the class does not have a public no-parameter constructor, you must specify the parameters and signature of a valid public constructor.

One advantage of this creation method is that the instantiate method of the MBean server also supports class loaders. If any are registered in the MBean server, they will automatically be used if the class is not available locally. See Chapter 12 for more information on class loading.

The main advantage of this method is that, like the first method of MBean creation, we retain a direct reference to the new object. The direct reference again enables us to use the MBean’s shortcut methods explicitly.

5.7 Managing MBeans

In “5.5 Creating an MBean Using the `createMBean` Method” on page 80, you rely totally on the MBean server to create and access an MBean. You get attributes and invoke operations through the MBean server. Here we will concentrate on the usage of MBean metadata classes when accessing MBeans that represent resources.

We will rely on the `StandardAgent` and `DynamicAgent` classes presented in Part I. As mentioned in “2.3.1 Comparison With the `SimpleStandard` Example” on page 52, the two are nearly identical. The same code works for any registered MBean, whether standard or dynamic. We examine the method for displaying MBean metadata that is common to both.

EXAMPLE 5–3 Processing MBean Information

```
private MBeanServer server = null; // assigned by MBeanServerFactory

private void printMBeanInfo(ObjectName mbeanObjectName, String mbeanName) {

    echo("    using the getMBeanInfo method of the MBeanServer");
    sleep(1000);
    MBeanInfo info = null;
    try {
         info = server.getMBeanInfo(mbeanObjectName);
    } catch (Exception e) {
          echo("\t!!! Could not get MBeanInfo object for " +
               mbeanName + " !!!");
          e.printStackTrace();
          return;
    }
    echo("\nCLASSNAME: \t"+ info.getClassName());
    echo("\nDESCRIPTION: \t"+ info.getDescription());
```

EXAMPLE 5–3 Processing MBean Information *(Continued)*

```
    echo("\nATTRIBUTES");
    MBeanAttributeInfo[] attrInfo = info.getAttributes();
    if (attrInfo.length>0) {
        for(int i=0; i<attrInfo.length; i++) {
            echo(" ** NAME: \t"+ attrInfo[i].getName());
            echo("    DESCR: \t"+ attrInfo[i].getDescription());
            echo("    TYPE: \t"+ attrInfo[i].getType() +
                 "\tREAD: "+ attrInfo[i].isReadable() +
                 "\tWRITE: "+ attrInfo[i].isWritable());
        }
    } else echo(" ** No attributes **");

    echo("\nCONSTRUCTORS");
    MBeanConstructorInfo[] constrInfo = info.getConstructors();
    for(int i=0; i<constrInfo.length; i++) {
        echo(" ** NAME: \t"+ constrInfo[i].getName());
        echo("    DESCR: \t"+ constrInfo[i].getDescription());
        echo("    PARAM: \t"+ constrInfo[i].getSignature().length +
             " parameter(s)");
    }

    echo("\nOPERATIONS");
    MBeanOperationInfo[] opInfo = info.getOperations();
    if (opInfo.length>0) {
        for(int i=0; i<constrInfo.length; i++) {
            echo(" ** NAME: \t"+ opInfo[i].getName());
            echo("    DESCR: \t"+ opInfo[i].getDescription());
            echo("    PARAM: \t"+ opInfo[i].getSignature().length +
                 " parameter(s)");
        }
    } else echo(" ** No operations ** ");

    echo("\nNOTIFICATIONS");
    MBeanNotificationInfo[] notifInfo = info.getNotifications();
    if (notifInfo.length>0) {
        for(int i=0; i<constrInfo.length; i++) {
            echo(" ** NAME: \t"+ notifInfo[i].getName());
            echo("    DESCR: \t"+ notifInfo[i].getDescription());
            String notifTypes[] = notifInfo[i].getNotifTypes();
            for (int j = 0; j < notifTypes.length; j++) {
                echo("    TYPE: \t" + notifTypes[j]);
            }
        }
    } else echo(" ** No notifications **");
}
```

The `getMBeanInfo` method of the MBean server gets the metadata of an MBean's management interface and hides the MBean's implementation. This method returns an `MBeanInfo` object that contains the MBean's description. We can then get the lists of attributes, operations, constructors, and notifications to display their descriptions. Recall that the dynamic MBean provides its own meaningful descriptions and that the standard MBean descriptions are default strings provided by the introspection mechanism of the MBean server.

5.8 Filtering MBeans With the Base Agent

The base agent does very little filtering because it does very little management. Usually, filters are applied programmatically to get a list of MBeans to which some operations apply. There are no management operations in the MBean server that apply to a list of MBeans. You must loop through your list and apply the desired operation to each MBean.

Before exiting the agent application, we do a query of all MBeans so that we can unregister them properly. MBeans should be unregistered before being destroyed because they might need to perform some actions before or after being unregistered. See the Javadoc API of the `MBeanRegistration` interface for more information.

EXAMPLE 5–4 Unregistering MBeans

```
public void removeMBeans() {

        try {
            echo("Unregistering all the registered MBeans except " +
                 "the MBean server delegate\n");
            echo("    Current MBean count = " + mbs.getMBeanCount()
                 + "\n");
            Set allMBeans = mbs.queryNames(null, null);
            for (Iterator i = allMBeans.iterator(); i.hasNext(); ) {
                ObjectName name = (ObjectName) i.next();
                if (!name.toString().equals(ServiceName.DELEGATE)) {
                    echo("\tUnregistering " + name.toString());
                    mbs.unregisterMBean(name);
                }
            }
            echo("\n    Current MBean count = " + mbs.getMBeanCount()
                +"\n");
            echo("done\n");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

We use the `queryNames` method because we only need the object names to operate on MBeans. The null object name as a filter gives us all MBeans in the MBean server. We then iterate through the resulting set and unregister each one, except for the MBean server delegate. As we shall see in "6.1.2 MBean Server Delegate" on page 89, the delegate is also an MBean and so it will be returned by the query. However, if we unregister it, the MBean server can no longer function and cannot remove the rest of our MBeans.

We recognized the delegate by its standard name which is given by the static field `ServiceName.DELEGATE`. The `ServiceName` class provides standard names and other default properties for communications and service MBeans. It also provides the version strings that are exposed by the delegate MBean. Note that, because the delegate is the only MBean created directly by the MBean server, it is the only one whose name cannot be overridden during its registration. The delegate object name is always the same, so we are always sure to detect it.

5.9 Running the Base Agent Example

The *examplesDir*`/current/BaseAgent/` directory contains the source file of the `BaseAgent` application.

▼ To Run the Base Agent Example

1. **Compile the `BaseAgent.java` file in this directory with the `javac` command.**

   For example, on the Solaris platform, type:

   ```
   $ cd examplesDir/current/BaseAgent/
   $ javac -classpath classpath *.java
   ```

   Again, we do not need the MBean classes at compile time, but they will be needed at runtime, because we do not use a dynamic class loader. You will need to have compiled the standard and dynamic MBean classes as described in "1.3 Running the Standard MBean Example" on page 40 and "2.3 Running the Dynamic MBean Example" on page 51. If you want to load any other class in the base agent, you must include its directory or JAR file in the classpath.

2. **To run the example, update your classpath to find the MBeans and start the agent class:**

   ```
   $ java -classpath classpath:../StandardMBean:../DynamicMBean BaseAgent
   ```

5.9.1 Agent Output

This agent displays output for the MBean created.

When the connection MBeans have been created, it is possible to connect to the agent through the HTML adaptor.

When you have finished, press `Enter` to remove all MBeans from the agent and exit the agent application.

CHAPTER 6

HTML Protocol Adaptor

The HTML protocol adaptor provides a view of the agent and its registered MBeans through a basic interface on any web browser. It is the easiest way to access an agent since no further coding is necessary. For this reason, it can be useful for testing and debugging your MBeans.

In this chapter, we will use your browser to “manage” the base agent and its MBeans. The HTML protocol adaptor outputs HTML pages that represent the agent and its MBeans. The adaptor also interprets the commands sent back by the buttons and fields appearing in your browser. It then interacts with the agent’s MBean server to get information about the MBeans that it has registered and to operate on them.

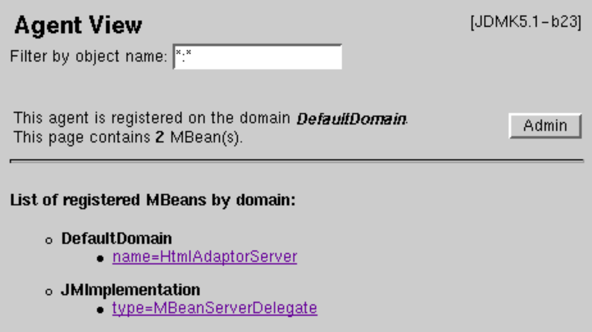
The HTML adaptor relies mostly on plain HTML. The only JavaScript™ that the generated pages contain are pop-up windows for displaying information. Browsers that are not enabled for JavaScript might give an incompatibility message and will not be able to display the information. Otherwise, the generated pages contain no further scripting (JavaScript, Visual Basic or other), no frames and no images that might slow down loading.

This chapter relies on the base agent that you will need to start first, as explained in Chapter 5. Once you can connect to the HTML protocol adaptor in the base agent, you are ready to begin these topics:

- “6.1 Agent View” on page 88 is the main page for managing an agent through the HTML protocol adaptor.
- “6.2 MBean View” on page 89 exposes an MBean’s management interface.
- The “6.3 Agent Administration” on page 94 enables you to instantiate new MBeans, modify attributes and invoke operations, and select the MBeans displayed in the agent view.

6.1 Agent View

The first page displayed by the HTML adaptor is always the agent view. It initially contains a list of all registered MBeans. The following figure shows the agent view for the base agent. It contains four MBeans: three communication MBeans, one of which is the HTML adaptor, and the MBean server delegate. The delegate is a special MBean explained in "6.1.2 MBean Server Delegate" on page 89.



Filter by object name: *:*

This agent is registered on the domain *DefaultDomain*.
This page contains **2** MBean(s).

List of registered MBeans by domain:

- **DefaultDomain**
  - name=HtmlAdaptorServer
- **JMImplementation**
  - type=MBeanServerDelegate

FIGURE 6–1 Initial Agent View of the Agent

The text field for filtering by object name enables you to modify the list of displayed MBeans. The filter string is initially `*:*`, which displays all registered MBeans. "6.3.2 Filtering MBeans" on page 97 describes how to use the filter. The agent's registered domain indicates the name of the default domain in this agent. The number of MBeans on this page is the count of those listed beneath the separator line.

The `Admin` button links to the agent administration page (see "6.3 Agent Administration" on page 94).

6.1.1 MBean List

The MBean list contains all MBeans whose object name matches the filter string. Object names can be filtered by their domain name and list of key-value pairs. In this list, MBeans are sorted and grouped by domain name. Each MBean name listed is an active link to the page of the corresponding MBean view.

After its initialization, the contents of an agent are dynamic. New MBeans can be created and registered into new or existing domains and old MBeans can be removed. These changes can also affect the functionality of the agent. New agent services can be registered (or removed) as well. We will demonstrate examples of dynamic management in "6.3.1 Instantiating and Managing MBeans" on page 95.

6.1.2 MBean Server Delegate

The MBean server delegate is an MBean that is automatically instantiated and registered by the MBean server when it is created. It provides information about the version of the Java Dynamic Management Kit (Java DMK) that is running, and it represents the MBean server when sending notifications.

Notifications are events sent by MBeans. They are covered in detail in Chapter 8. Because the MBean server instance is not an MBean object, it relies on its delegate MBean to send notifications. The MBean server delegate sends notifications to inform interested listeners about such events as MBean registrations and unregistrations.

The exposed attributes of the delegate MBean provide vendor and version information about the MBean server. This can let a remote management application know which agent version is running and which version of the Java runtime environment it is using. The delegate MBean also provides a unique identification string for its MBean server.

▼ To View the MBean Server Delegate Information

1. **Click the name of the delegate MBean to see its attributes.**

   Version, vendor and identification information is listed in the table of attributes.

2. **Click the `Back to Agent View` link or use your browser's "Previous page" function to return to the MBean list in the agent view.**

6.2 MBean View

The MBean view has two functions: it presents the management interface of the MBean and it enables you to interact with its instance. The management interface of an MBean is given through the name of the attributes, the operation signatures, and a self-description. You can interact with the MBean by reloading its attribute values, setting new values, or invoking an operation.

▼ To Display the MBean View

1. **Display the agent view.**

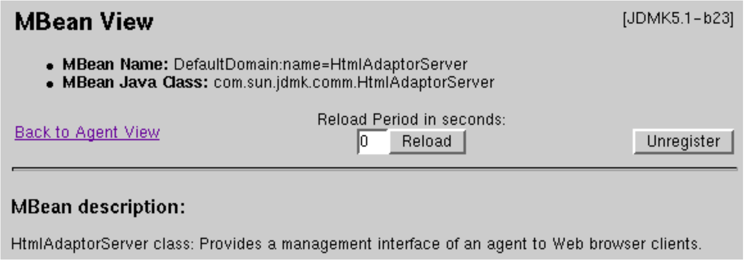
   This is the first page displayed by the HTML adaptor.

2. **In the agent view, click on the object name of the HTML adaptor MBean: `name=HTMLAdaptorServer` in the default domain.**

This will display its MBean view.

6.2.1 Header and Description

As shown in Figure 6–2, the top part of the page contains the description of the MBean and some controls for managing it.



MBean View [JDMK5.1-b23]

- **MBean Name:** DefaultDomain:name=HtmlAdaptorServer
- **MBean Java Class:** com.sun.jdmk.comm.HtmlAdaptorServer

Reload Period in seconds: 0

MBean description:

HtmlAdaptorServer class: Provides a management interface of an agent to Web browser clients.

FIGURE 6–2 Description in the MBean View

The first two lines give the object instance (object name and class name) for this MBean. The MBean name is the full object name of this MBean instance, including the domain. The key-property pairs might or might not identify the MBean to a human reader, depending on the agent developer's intention. The MBean Java class is the full class name for the Java object of which this MBean is an instance.

The reload controls include a text field for entering a reload period and a manual `Reload` button. Initially, the reload period is set to zero indicating that the contents of the MBean view are not automatically refreshed. Clicking the `Reload` button forces the page to reload, thereby updating all of the attribute values displayed. If you have entered a reload period, clicking the button will begin automatic reloading with the given period. The reload period must be at least five seconds.

Note – Use the `Reload` button of the MBean view instead of the browser's reload page button. After some operations, such as applying changes to attribute values, the browser's button will repost the form data, inadvertently performing the same operation again. To avoid undesirable side effects, always use the `Reload` button provided in the MBean view.

▼ To Set the Reload Period

1. **Type a reload period of `5` and click the `Reload` button.**

   Every five seconds the page will blink as it reloads.

2. **In another browser window, open another connection to the HTML adaptor at `http://localhost:8082/`.**

3. **Observe the new values for the `ActiveClientCount` and `LastConnectedClient` attributes in the original window.**

   You might have to try several connections before you see the attribute values change.

The reload period is reset to zero every time you open an MBean view.

The `Unregister` button is a shortcut for removing this MBean from the agent. Unregistering is covered in "6.3.1 Instantiating and Managing MBeans" on page 95.

The MBean description text provides some information about the MBean. Because standard MBeans are statically defined, they cannot describe themselves and the MBean server provides a generic text. Dynamic MBeans are required to provide their own description string at runtime according to the JMX specification. Except for the class name, this is the only way to tell standard and dynamic MBeans apart in the MBean view.

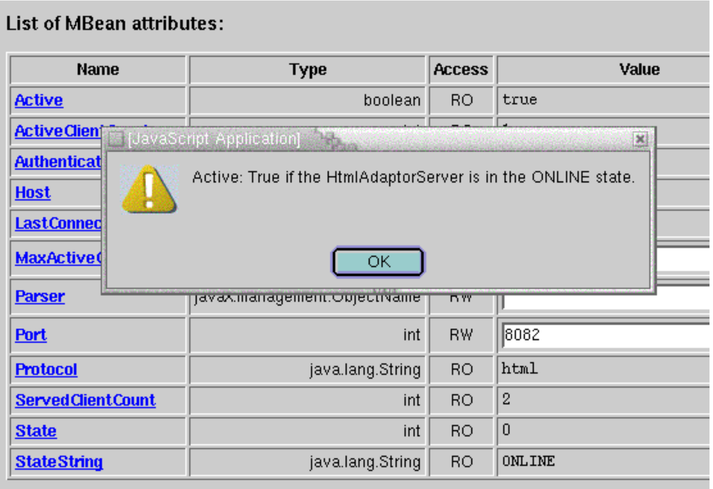
6.2.2 Table of Attributes

The second part of the MBean view is a table containing all attributes exposed by the MBean. For each attribute, this table lists its name, its Java type, its read-write access, and a string representation of its current value.

While MBean attributes can be of any type, not all types can be displayed in the MBean view. The HTML adaptor is limited to basic data types that can be displayed and entered as strings. Read-only attributes whose type supports the `toString` method are also displayed. Enumerated types that are concrete subclasses of `com.sun.jdmk.Enumerated` are displayed as a menu with a pop-up selection list. Boolean attributes are represented as true-false radio buttons. Finally, attributes with array types are represented by a link to a page that displays the array values in a table. If the attribute is writable, you can enter values for the array elements to set them.

For the complete list of supported types, see the Javadoc API of the `HtmlAdaptorServer` class. If an attribute type is not supported, this is indicated in place of its value. If there was an error when reading an attribute's value, the table shows the name of the exception that was raised and the message it contains.

The name of each attribute is a link that displays a window containing the description for this attribute. Like the MBean description, attribute descriptions can only be provided by dynamic MBeans. The MBean server inserts a generic description for standard MBean attributes. Figure 6–3 shows the attributes of the HTML adaptor with a description of the `Active` attribute.



List of MBean attributes:

Name	Type	Access	Value
Active	boolean	RO	true
ActiveClien			
Authenticat			
Host			
LastConnec			
MaxActiveC			
Parser	javax.management.ObjectName	RW	
Port	int	RW	8082
Protocol	java.lang.String	RO	html
ServedClientCount	int	RO	2
State	int	RO	0
StateString	java.lang.String	RO	ONLINE

[JavaScript Application]

Active: True if the HtmlAdaptorServer is in the ONLINE state.

FIGURE 6–3 MBean Attributes With a Description Window

▼ To View an Attribute Description

- **Click on an attribute name in the table of attributes to read its description.**

  Because the HTML adaptor is implemented as a dynamic MBean, its attribute descriptions are meaningful.

  Note – Due to the use of JavaScript commands in the generated HTML, these pop-up windows might not be available on browsers that are not JavaScript-enabled.

Writable attributes have a text field for entering new values. To set the value of a writable attribute, type or replace its current value in the text field and click the `Apply` button at the bottom of the attributes table.

Note – Do not try to modify the attributes of the HTML protocol adaptor here. See “6.3.1 Instantiating and Managing MBeans” on page 95.

Because there is only one `Apply` button for all the attributes, this systematically invokes the setter for all writable attributes, whether or not their fields have actually been modified. This might affect the MBean if setters have side effects, such as counting the number of modifications, as in the `SimpleStandard` and `SimpleDynamic` examples given in Part I

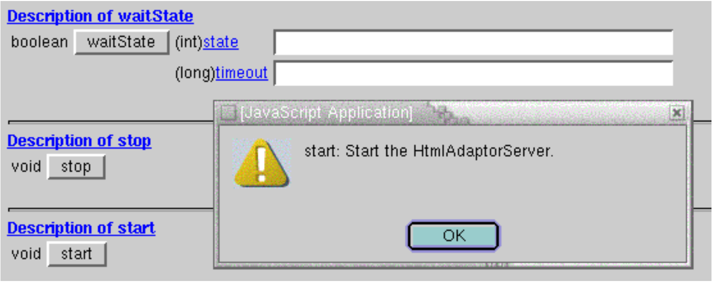
The HTML adaptor detects attributes of the `ObjectName` type and provides a link to the view of the corresponding MBean. This link is labeled `view` and is located just under the displayed value of the object name. Because MBeans often need to reference other MBeans, this provides a quick way of navigating through MBean hierarchies.

6.2.3 List of Operations

The last part of the MBean view contains all the operations exposed by the MBean. Each operation in the list is presented like a method signature. There is a return type, then a button with the operation name, and if applicable, a list of parameters, each with their type as well.

As with the table of attributes, the list of operations contains only those involving types that can be represented as strings. The return type must support the `toString` method and the type of each parameter must be one of basic data types supported by the HTML adaptor. For the complete list, see the Javadoc API of the `HtmlAdaptorServer` class.

Above each operation name is a link to its description. Parameter names are also active links that display a window with a description. Again, descriptions are only meaningful when provided by dynamic MBeans. The following figure shows some of the operations exposed by the HTML adaptor MBean and a description of the `Start` operation.



start: Start the HtmlAdaptorServer.

FIGURE 6–4 MBean Operations With a Description Window (Partial View)

We will not perform any operations on this MBean until after a brief explanation in “6.3.1 Instantiating and Managing MBeans” on page 95.

To perform an operation, fill in any and all parameter values in the corresponding text fields and click the operation’s button. The HTML adaptor displays a page with the result of the operation, either the return value if successful or the reason why the operation was unsuccessful.

6.3 Agent Administration

The agent administration page contains a form for entering MBean information when creating or unregistering MBeans. You can also instantiate an MBean through one of its public constructors. In order to instantiate an MBean, its class must be available in the agent application's classpath. Optionally, you can specify a different class loader if the agent contains other class loader MBeans.

▼ To Display the Agent Administration Page

1. **Go back to the agent view by clicking the link near the top of the MBean view page.**

2. **Click on the `Admin` button in the agent view to display the agent administration page in your browser window.**

The first two fields, `Domain` and `Keys` are mandatory for all administrative actions. The `Domain` field initially contains the string representing the agent's default domain. Together, these two fields define the object name, whether for a new MBean to be created or for an existing MBean to unregister. The Java class is the full class name of the object to be instantiated as a new MBean. This field is ignored when unregistering an MBean.

Using the pull-down menu, you can choose one of three actions:

- **Create** – Instantiates the given Java class of an MBean and registers the new instance in the MBean server. If successful, the MBean appears in the agent view. The class must have a public constructor without parameters in order to be created in this way.
- **Unregister** – Unregisters an MBean from the MBean server so that it is no longer available in the agent. The class instance is not deleted explicitly, though if no other references to it exist, it will be collected by the garbage collector.
- **Constructors** – Displays the list of public constructors at the bottom of the administration page for the given Java class. This lets you provide parameters to a specific constructor and create the MBean in this manner. This is the only way to create MBeans that do not have a constructor without parameters.

When you click the `Send Request` button, the HTML adaptor processes the action and updates the bottom of the page with the action results. You might have to scroll down to see the result. The text fields are not cleared after a request, enabling you to perform multiple operations. Clicking the `Reset` button returns the fields to their last posted value after you have modified them.

6.3.1 Instantiating and Managing MBeans

Sometimes, starting an MBean requires several steps, especially for agent services that require some sort of configuration. For example, you can instantiate another HTML adaptor for connecting to a different port. Usually, this is done programmatically in the agent application, but we must do it through the browser for the agent.

▼ To Create a New HTML Adaptor MBean

1. **On the agent administration page, fill in the fields as follows:**

`Domain`	**`Communications`**
`Keys:`	**`protocol=html,port=8088`**
`Java Class:`	**`com.sun.jdmk.comm.HtmlAdaptorServer`**
`Class Loader:`	*leave blank*

   In versions of the Java DMK prior to version 4.0, specifying the port number in the object name would initialize communication MBeans. Now, the names and contents of key properties no longer have any significance for any components of the product. We must set the port in other ways.

2. **Make sure the action selected is `Create` and send the request. If you scroll down the page, you will see whether your request was successful.**

   We cannot connect to this HTML adaptor quite yet. First we need to configure it .

3. **Go to the new HTML adaptor's MBean view with the provided link.**

   Remember that we could not modify any of the adaptor's attributes in "To View an Attribute Description" on page 92 because the implementation does not allow them to be modified while the HTML adaptor is online. Our new HTML adaptor is instantiated in the stopped state (the `StateString` attribute indicates `OFFLINE`), so we *can* change its attributes.

4. **Set the `Port` attribute to `8088` and `MaxActiveClientCount` to `2`, then click the `Apply` button.**

   If the page is reloaded and the new values are displayed, the attribute write operation was successful. You can also click the attribute names to get a description them.

5. **Scroll down the MBean view to the `Start` operation and click the `Start` button.**

   A new page is displayed to tell us the operation was successful. If you go back to the MBean view with the provided link, you can see that the `StateString` is now indicating `ONLINE`.

6. **Now you can access your agent through a browser on port 8088. Try going to a different host on the same network and connecting to the following URL:**

   `http://`*agentHostName*`:8088/`

In the above URL, *agentHostName* is the name or IP address of the host where you launched the `BaseAgent`. If you reload the MBean view of the new HTML adaptor on either browser, the name of this other host should be the new value of the `LastConnectedClient` attribute.

Through this new connection, you can stop, modify or remove the HTML adaptor MBean using port 8082. In that case, your original browser will also have to use `http://localhost:8088/` to connect. Instead, we will manage the agent from the second host.

To Instantiate MBeans With Constructors

1. **From the browser on the second host, go to the agent administration page. Fill in the fields as follows and request the constructors:**

   `Domain:` **`Standard_MBeans`**

   `Keys:` **`name=SimpleStandard,number=1`**

   `Java Class:` **`SimpleStandard`**

   `Class Loader:` *leave blank*

   The list of constructors for the `SimpleStandard` class is displayed at the bottom of the page. The MBean name is also shown. This is the object name that will be assigned to the MBean when using one of the listed constructors. As you can see, the `SimpleStandard` class only has one constructor that takes no parameters.

2. **Click on the `Create` button. The result is appended to the bottom of the page. Scroll down and go to the MBean view with the provided link.**

   Because it is a standard MBean, all of its description strings are generic. This shows the necessity of programming meaningful attribute names.

3. **In the agent view on the first browser window, click in the filter field and press `Return` to refresh the agent view.**

4. **Click the new MBean's name and set its reload period to `15`.**

5. **On the second host, type in a different string for the `State` attribute and click `Apply`.**

   On the first host, you should see the MBean's attributes are updated when the MBean view is periodically reloaded.

6. **On the second host, click the `Reset` operation button at the bottom of the MBean view page.**

   The operation result page is displayed and indicates the success of the operation.

   This page also provides the return value of the operation when it is not void. If you go back to the MBean view, you will see the result of the operation on the attributes. You should also see it on the first host after it reloads.

The browser on the second host is no longer needed and we can remove the HTML adaptor on port 8088.

▼ To Unregister an MBean

1. **Go to the administration page and fill in the object name of the HTML adaptor you want to remove (you do not need its Java class to unregister it):**

`Domain:`	**`Communications`**
`Keys:`	**`protocol=html,port=8088`**
`Java Class:`	*leave blank*
`Class Loader:`	*leave blank*

2. **Choose `Unregister` from the pull-down menu and click the `Send Request` button.**

   The result is displayed at the bottom of the page.

   You can also unregister an MBean directly from its MBean view. Click the `Unregister` button on the upper right side of the page.

6.3.2 Filtering MBeans

Because an agent can manage hundreds of MBeans, the agent view provides a filtering mechanism for the list that is displayed. An object name with wildcard characters is used as the filter, and only those MBeans that match are counted and displayed.

Filters restrict the set of MBeans listed in the agent view. This might not be particularly useful for our small agent, but it can help you find MBeans among hundreds in a complex agent. In addition, management applications use the same filter syntax when requesting an agent's MBeans through the programmatic interface of a connector. The filtering enables managers to either get lists of MBean names or find a particular MBean instance.

Filters are typed as partial object names with wildcard characters or as a full object name for which to search, using the syntax *domain*`:`*key*. Here are the basic substitution rules for filtering.

1. You can search for partial *domain* names, using the following characters:

Asterisk (`*`)	Replaces any number (including zero) of characters
Question mark (`?`)	Replaces any one character

2. An empty domain name is replaced by the default domain string; an empty key list is illegal

3. Keys are atomic; you must search for the full `property=value` key; you cannot search for a partial property name or an incomplete value
4. An asterisk (`*`) can be used to terminate the key list, where it replaces any number of any keys (complete property-value pairs)
5. You must match all keys exactly; use the form `property=value,*` to search for one key in names with multiple keys
6. Keys are unordered when filtering; giving one or more keys (and an asterisk) in any order finds all object names that contain that subset of keys

▼ To Filter MBeans

1. **Go to the administration page and create three more standard MBeans. Modify only the `number` value in their object names so that they are numbered sequentially.**

2. **In the same way as Step 1, create four dynamic MBeans starting with:**

`Domain:`	**`Dynamic_MBeans`**
`Keys:`	**`name=SimpleDynamic,number=1`**
`Java Class:`	**`SimpleDynamic`**
`Class Loader:`	*leave blank*

3. **Go back to the agent view, that should display all the new MBeans.**

4. **Type each of the filter strings given in Table 6–1 to see the resulting MBean list**

TABLE 6–1 Examples of Filter Strings

Filter String (*domain*:*key*)	Result
`Standard_MBeans:*`	Gives all of the standard MBeans we created
`*_MBeans:*`	Gives all of the standard and dynamic MBeans we created
`DefaultDomain:`	Not allowed by rule 2
`:*`	Lists all MBeans in the default domain
`*:name=Simple*,*`	Not allowed by rule 3
`*:name=SimpleStandard`	Allowed, but list is empty (rule 5)
`*:name=*`	Not allowed by rule 3
`*_??????:number=2,*`	Gives the second standard and dynamic MBean we created

TABLE 6–1 Examples of Filter Strings *(Continued)*

Filter String (*domain*:*key*)	Result
`Communications:port=8088,protocol=html`	Gives the one MBean matching the domain and both (unordered) keys
empty string	Allowed: special case equivalent to `*:*`

Notice how the MBean count is updated with each filter. This count shows the number of MBeans that were found with the current filter, which is the number of MBeans appearing on the page. It is not the total number of MBeans in the agent, unless the filter is `*:*`.

5. **When you are ready to stop the base agent example, go to the window where you started its class and press `Control-C`.**

CHAPTER 7

MBean Server Interceptors

An MBean server forwards requests it receives to its default interceptor. A feature of the Java Dynamic Management Kit (Java DMK) enables you to modify this behavior of the MBean server and replace the default interceptor by another object implementing the same interface.

The example provided demonstrates how you can use the concept of MBean server interceptors to forward requests to a specific interceptor, and to support "virtual" MBeans. The code samples in this chapter are from the files in the `MBeanServerInterceptor` example directory located in the main *examplesDir*/`current` directory (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

7.1 Overview of MBean Server Interceptors

The concept of *interceptors* exploits the proxy design pattern to enable you to modify the behavior of the MBean server. By default, the MBean server appears from the outside like a hollow shell that simply forwards every operation to the default interceptor. You can replace this default interceptor by another object implementing the same interface, to change the semantics of the MBean server. In most cases, you would use this other object to forward most or all operations to the default interceptor after doing some processing. However, you can also use it to forward some operations to other handlers instead, for instance depending on the object names involved. Figure 7–1 shows schematically how you can insert an interceptor between the MBean server and the default interceptor.

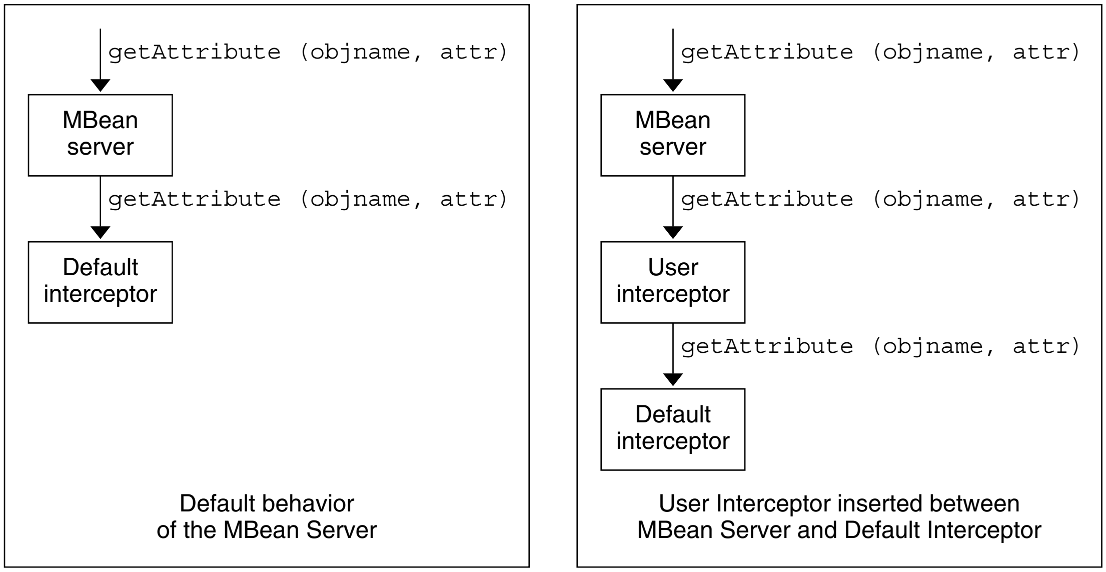

getAttribute (objname, attr)
MBean server
getAttribute (objname, attr)
Default interceptor
Default behavior of the MBean Server
getAttribute (objname, attr)
MBean server
getAttribute (objname, attr)
User interceptor
getAttribute (objname, attr)
Default interceptor
User Interceptor inserted between MBean Server and Default Interceptor

FIGURE 7–1 Inserting a User Interceptor Between the MBean Server and the Default Interceptor

Some examples of the uses of interceptors are as follows:

- *Imposing security checks* – The interceptor performs checks, for example it checks permissions, and only forwards operations that pass the security checks to the default interceptor.
- *Logging* – The interceptor forwards all operations to the default interceptor, but also logs their parameters and results.
- *Creating virtual MBeans* – The interceptor handles operations on MBeans it owns itself, and forwards others to the default interceptor. An interceptor might *own* all MBeans whose names match a particular pattern, for instance. In this way, MBeans do not necessarily have to be Java objects. This is very useful when there are a great many managed objects, or when they are very volatile.

Interceptors can be composed. When an interceptor is added, it is usually inserted between the MBean server shell and the current interceptor. Initially, the current interceptor is the default one. But if another interceptor has already been inserted, this other interceptor is the current one. Hence, a request could pass through several interceptors on its way to the default interceptor, for example a security checker and a logger.

7.2 Specifying the Behavior of an MBean Server Interceptor

The behavior to be implemented by an interceptor is specified by means of the `MBeanServerInterceptor` interface. An MBean server interceptor has essentially the same interface as an MBean server. An MBean server forwards received requests to its default interceptor, which might handle them itself or forward them to other interceptors.

7.3 Changing the Default Interceptor

The default interceptor can be changed by using the `setMBeanServerInterceptor` method of the `JdmkMBeanServer` interface. The `JdmkMBeanServer` interface provides methods for getting and setting the default `MBeanServerInterceptor` used for request treatment.

By default, the `MBeanServer` implementation returned by the `MBeanServerFactory` is not a `JdmkMBeanServer`, and does not support `MBeanServerInterceptors`. To use interceptor you have to set the `javax.management.builder.initial` System property to `com.sun.jdmk.JdmkMBeanServerBuilder` before obtaining an MBean server from the `MBeanServerFactory`. How to set this system property is shown in Step 2.

Note – Particular care must be taken when replacing the default MBean server interceptor with a user interceptor. The MBean server implemented in the Java DMK 5.1 passes requests to its default interceptor without checking the result returned, or the exceptions thrown by the interceptor.

Consequently, user interceptors, which implement most of the methods defined in the `MBeanServer` interface, must behave as specified for the corresponding `MBeanServer` methods in the JMX specification. In particular, a method in an MBean server interceptor must not throw any exceptions apart from the following:

- Exceptions explicitly declared in the `throws` clause of the same method in the interface `com.sun.jdmk.interceptor.MBeanServerInterceptor`
- `JMRuntimeException` or a subclass of it

If an MBean server interceptor does not respect this condition, and, for example, throws a `NullPointerException` exception, this might have unexpected effects on calling code, which might not be protected against such behavior.

7.4 Running the MBean Server Interceptor Example

The MBean server interceptor example in the examples directory shows you two of the main functions of MBean server interceptors, forwarding requests to a specific MBean server interceptor, and creating virtual MBeans.

The *examplesDir*`/current/MBeanServerInterceptor` directory contains the following source files:

- `MasterMBeanServerInterceptor.java`. This master interceptor receives all requests from the MBean server and, depending on the value of the domain part of the `ObjectName`, forwards them to one of the following interceptors:
  - The default MBean server interceptor.
  - Another MBean server interceptor, the `FileMBeanServerInterceptor`.
- `FileMBeanServerInterceptor.java`. This is an MBean server interceptor that mirrors the contents of a file system directory by faking MBeans which represent files and directories. These MBeans are completely virtual. The `FileMBeanServerInterceptor` owns a reserved domain name, the `file` domain in this example, which is used by the `MasterMBeanServerInterceptor` to decide which requests to divert to the `FileMBeanServerInterceptor`.

- `Agent.java`. This class implements a simple Java DMK agent which instantiates a `MasterMBeanServerInterceptor` and plugs in a `FileMBeanServerInterceptor`. This class shows how to instantiate the `MasterMBeanServerInterceptor`, how to plug it into the MBean Server, and how to plug the `DefaultMBeanServerInterceptor` and `FileMBeanServerInterceptor` into the `MasterMBeanServerInterceptor`.

▼ To Run the MBean Server Interceptor Example

1. **Compile all files in the *examplesDir*`/current/MBeanServerInterceptor` directory with the `javac` command.**

   For example, on the Solaris platform, type:

   ```
   $ cd examplesDir/current/MBeanServerInterceptor/
   $ javac -classpath classpath *.java
   ```

2. **Run the example using the classes you have just built, by typing the following command in a terminal window:**

   ```
   $ java \
   -Djavax.management.builder.initial=com.sun.jdmk.JdmkMBeanServerBuilder \
   Agent
   ```

   Here, you can see that the `javax.management.builder.initial` system property is set to `com.sun.jdmk.JdmkMBeanServerBuilder` before the Agent is started, as explained in "7.3 Changing the Default Interceptor" on page 103.

3. **Interact with the agent through the standard input and output in the window where it was started.**

4. **Load the agent's URL in your web browser:**

   ```
   http://localhost:8082/
   ```

   You only see the MBeans registered in the `DefaultMBeanInterceptor`, namely the connector and adaptor MBeans, and the `MBeanServerDelegate`.

5. **Press `Enter` to insert the `FileMBeanServerInterceptor` and view the files from the local directory as virtual MBeans.**

6. **Reload the agent's URL in your web browser to view the new MBeans:**

   ```
   http://localhost:8082/
   ```

7. **Press `Enter` to stop the agent.**

CHAPTER 8

Notification Mechanism

This chapter presents the mechanisms for sending and receiving notifications by demonstrating them locally on the agent-side. MBeans for either resources or services are the source of notifications, called *broadcasters*. Other MBeans or objects that want to receive the notifications register with one or more broadcasters, and they are called *listeners*.

Notification mechanisms are demonstrated through two sample broadcasters. The MBean server delegate that notifies listeners of MBean creation and unregistration, and an MBean that sends attribute change notifications.

The code samples are from the files in the `examplesDir/current/Notification` example directory located in the main *examplesDir* (see *Directories and Classpath* in the Preface).

This chapter covers the following topics:

- "8.1 Overview of Notifications" on page 108 introduces the `Notification` object used to send generic events, identified by their notification type string.
- "8.2 MBean Server Delegate Notifications" on page 109 explains the concepts of notification broadcasters and listeners through the simple mechanism of an MBean sending notifications and a listener object receiving them.
- "8.3 Attribute Change Notifications" on page 113 provides an example of creating subclasses of the `Notification` object to provide additional information to the listener.
- "8.4 Running the Agent Notification Example" on page 118 enables you to trigger attribute change notifications through the HTML protocol adaptor.

8.1 Overview of Notifications

The ability for resources and other entities to signal an event to their managing applications is a key functionality of management architectures. As defined in the Java Management Extensions (JMX) specification, notifications in the Java Dynamic Management Kit (Java DMK) provide a generic event mechanism whereby a listener can receive all events sent by a broadcaster.

All notifications in the Java DMK rely on the `Notification` class that itself inherits from Java event classes. A string called the *notification type* inside a `Notification` object gives the nature of the event, and other fields provide additional information to the recipient. This ensures that all MBeans, the MBean server, and remote applications can send and receive `Notification` objects and its subclasses, regardless of their inner type.

You can define new notification objects only by subclassing the `Notification` class. This ensures that the custom notifications will be compatible with the notification mechanism. New notification classes can be used to convey additional information to custom listeners, and generic listeners will still be able to access the standard `Notification` fields. However, because there are already fields provided for user data, subclassing is discouraged in the JMX architecture so that notification objects remain as universal as possible.

Listeners usually interact with notification broadcasters indirectly through the MBean server. The interface of the MBean server enables you to associate a listener with any broadcaster MBean, thereby giving you dynamic access to any of the broadcasters that are registered. In addition, the MBean metadata provided through the MBean server contains the list of notification types that the MBean broadcasts.

Figure 8–1 summarizes how listeners register with broadcasters and then receive notifications, in an agent application.

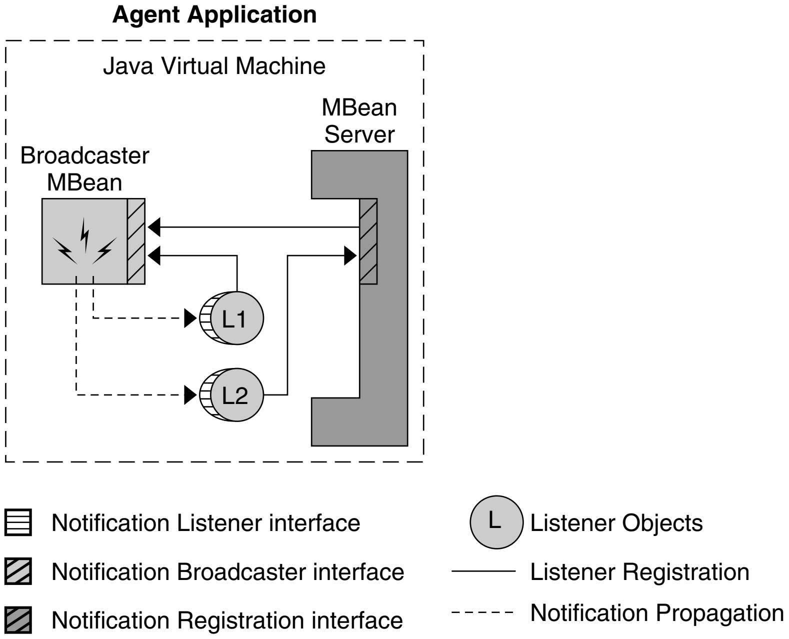


FIGURE 8–1 Listener Registration and Notification Propagation

8.2 MBean Server Delegate Notifications

The MBean server delegate object is an MBean that is automatically created and registered when an MBean server is started. It preserves the management model by serving as the management interface for the MBean server. The delegate exposes read-only information such as the vendor and version number of the MBean server. More importantly for this chapter, it sends the notifications that relate to events in the MBean server: all MBean registrations and unregistrations generate a notification.

8.2.1 `NotificationEmitter` Interface

A class must implement the `NotificationEmitter` interface to be recognized as a source of notifications in the JMX architecture. This interface provides the methods for adding or removing a notification listener to or from the emitter. When the emitter sends a notification, it must send it to all listeners that are currently registered through this interface.

This interface also specifies a method that returns information about all notifications that might be sent by the emitter. This method returns an array of MBeanNotificationInfo objects, each of which provides a name, a description string, and the type string of the notification.

As detailed in the Javadoc API, the MBeanServerDelegate class implements the NotificationEmitter interface. We know from the JMX specification that it sends notifications of the following types:

- JMX.mbean.registered
- JMX.mbean.unregistered

Note – Although emitter objects are almost always MBeans, they should not expose the methods of the NotificationEmitter interface. That is, the MBean interface of a standard MBean should never extend the NotificationEmitter interface. As we shall see in Chapter 22, the remote connector clients provide the methods needed to register for and receive notifications remotely.

The NotificationEmitter should be used in preference to the old NotificationBroadcaster class.

8.2.2 NotificationListener Interface

Listeners must implement the NotificationListener interface, and they are registered in the notification broadcasters to receive the notifications. The listener interface defines a *handler* method that receives all notifications of the broadcaster where the listener is registered. We say that a listener is *registered* when it has been added to the broadcaster's list of notification recipients. This is completely independent of any registration of either object in the MBean server.

Like the broadcaster, the listener is generic, meaning that it can handle any number of different notifications. Its algorithm usually involves determining the type of the notification and taking the appropriate action. A listener can even be registered with several broadcasters and handle all of the notifications that are sent.

Note – The handler is a *callback* method that the broadcaster calls with the notification object it wants to send. As such, it runs in the broadcaster's thread and should therefore run rapidly and return promptly. The code of the handler should rely on other threads to execute long computations or blocking operations.

In our example, the listener is a trivial class that has a constructor and the handler method. Our handler simply prints out the nature of the notification and the name of the MBean to which it applied. Other listeners on the agent side might themselves be

MBeans that process the event and update the state of their resource or the quality of their service in response. For example, the relation service must know when any MBeans participating in relations are unregistered. It does this by listening to MBean server delegate notifications.

EXAMPLE 8–1 Listener for MBean Server Delegate Notifications

```
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.MBeanServerNotification;

public class AgentListener implements NotificationListener {

         public AgentListener() {
         }

         public void handleNotification(Notification n,
                                   Object h) {

        // Process the different types of notifications fired by the
        // MBean server delegate.
        String type = n.getType();

         try {
             if (type.equals(
                     MBeanServerNotification.REGISTRATION_NOTIFICATION)) {
                     echo("\t>> \"" +
                         ((MBeanServerNotification) n).getMBeanName() +
                         "\" has been registered in the server");
                  } else if (type.equals(
                     MBeanServerNotification.UNREGISTRATION_NOTIFICATION)) {
                      echo("\t>> \"" +
                           ((MBeanServerNotification)n).getMBeanName() +
                           "\" has been unregistered from the server\n");
                  } else {
                       echo("\t>> Unknown event type (?)\n");
             }
        } catch (Exception e) {
             e.printStackTrace();
             System.exit(1);
        }
    }
}
```

In most cases, the notification object passed to the handler method is an instance of the `Notification` class. This class provides the notification type as well as a time-stamp, a sequence number, a message string, and user data of any type. All of these are provided by the broadcaster to pass any needed information to its listeners. Because listeners are usually registered through the MBean server, they only know the broadcaster by its object name, given by the `getSource` method of the notification object.

Note – The notification model does not assume that notifications will be received in the same order that they are sent. If notification order is critical to your application, your broadcaster should set the sequence numbers appropriately, and your listeners should sort the received notifications.

The MBean server delegate sends `MBeanServerNotification` objects that are subclasses of the `Notification` class. This subclass provides two constants to identify the notification types sent by the delegate and a method that gives the object name of the MBean that was registered or unregistered. Our notification handler uses these to print out the type of operation and the object name to which the operation applies.

8.2.3 Adding a Listener Through the MBean Server

Now that we have identified the objects involved, we need to add the listener to the notification broadcaster. Our example does this in the `main` method of the agent application.

EXAMPLE 8–2 Registering for MBean Server Delegate Notifications

```
AgentListener agentListener = null;
[...]

echo("\nRegistering the MBean server delegate listener...");
try {
    agentListener = new AgentListener();
    myAgent.server.addNotificationListener(
        new ObjectName(ServiceName.DELEGATE),
        agentListener, null, null);
} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}
echo("done");
```

In Example 8–2, the agent application adds the `AgentListener` instance to the delegate MBean, which is known to be a broadcaster. The object name of the MBean server delegate is given by the `DELEGATE` constant in the `ServiceName` class. The listener is added through the `addNotificationListener` method of the MBean server. This method preserves the management architecture by adding listeners to MBeans while referring only to the MBean object names.

If an MBean implements the listener interface and needs to receive certain notifications, it can add itself to a broadcaster. For example, an MBean could use its preregistration method in order to add itself as a notification listener or it could expose a method that takes the object name of the notification broadcaster MBean. In both cases, its notification handler method must be designed to process all expected notification types.

The last two parameters of the `addNotificationListener` methods of both the `MBeanServer` and the `NotificationBroadcaster` interfaces define a filter and a handback object, respectively. Filter objects are defined by the `NotificationFilter` interface and provide a callback method that the broadcaster calls before calling the notification handler. If the filter is defined by the entity that adds the listener, it prevents the handler from receiving unwanted notifications.

Handback objects are added to a broadcaster along with a listener and are returned to the designated handler with every notification. The handback object is completely untouched by the broadcaster and can be used to transmit context information from the entity that adds the listener to the handler method. The functionality of filters and handback objects is beyond the scope of this tutorial; refer to the JMX specification for their full description.

8.3 Attribute Change Notifications

In this second part of the notification example, we demonstrate attribute change notifications that can be sent by MBeans. An MBean designer can choose to send notifications whenever the value of an attribute changes or is changed. The designer can implement this mechanism in any manner, according to the level of consistency required by the management solution.

The JMX specification only provides a subclass of notifications to use to represent the attribute change events: the `AttributeChangeNotification` class.

8.3.1 `NotificationBroadcasterSupport` Class

The broadcaster in our example is a very simple MBean. The `reset` method triggers a notification whenever it is called. This policy is specific to our example. You might want to design an MBean that sends an attribute change every time the setter is called, regardless of whether or not the value is modified. Your management needs might vary.

Example 8–3 shows the code for our `SimpleStandard` MBean class (the code for its `MBean` interface has been omitted).

EXAMPLE 8–3 Broadcaster for Attribute Change Notifications

```
import javax.management.NotificationBroadcasterSupport;
import javax.management.MBeanNotificationInfo;
import javax.management.AttributeChangeNotification;

public class SimpleStandard
    extends NotificationBroadcasterSupport
    implements SimpleStandardMBean {

    public String getState() {
        return state;
    }

    public void setState(String s) {
        state = s;
        nbChanges++;
    }

    [...]

    public int getNbChanges() {
        return nbChanges;
    }

    public void reset() {
             AttributeChangeNotification acn =
                 new AttributeChangeNotification(this,
                                                    0,
                                                    0,
                                                    "NbChanges reset",
                                                    "NbChanges",
                                                    "Integer",
                                                    new Integer(nbChanges),
                                                    new Integer(0));
             state = "initial state";
          nbChanges = 0;
             nbResets++;
             sendNotification(acn);
    }

   public int getNbResets() {
           return nbResets;
   }

   public MBeanNotificationInfo[] getNotificationInfo() {
        return new MBeanNotificationInfo[] {
              new MBeanNotificationInfo(
              new String[] { AttributeChangeNotification.ATTRIBUTE_CHANGE },
              AttributeChangeNotification.class.getName(),
              "This notification is emitted when the reset() method is
            called.")
       };

    private String      state = "initial state";
```

EXAMPLE 8–3 Broadcaster for Attribute Change Notifications *(Continued)*

```
    private int         nbChanges = 0;
    private int         nbResets = 0;
}
```

This MBean sends its notifications and it implements the `NotificationBroadcaster` interface by extension of the `NotificationBroadcasterSupport` class, in order to provide all the mechanisms for adding and removing listeners and sending notifications. It manages an internal list of listeners and their handback objects and updates this list whenever listeners are added or removed. In addition, the `NotificationBroadcasterSupport` class provides the `sendNotification` method to send a notification to all listeners currently on its list.

By extending this object, our MBean inherits all of this behavior. Subclassing `NotificationBroadcasterSupport` is a quick and convenient way to implement notification broadcasters. We do not even have to call a superclass constructor because it has a default constructor. We only need to override the `getNotificationInfo` method to provide details about all of the notifications that might be sent.

8.3.2 Attribute Change Listener

Like our listener for MBean server notifications, our listener for attribute change notifications is a trivial class consisting of just the handler method.

EXAMPLE 8–4 Listener for Attribute Change Notifications

```
import javax.management.Notification;
import javax.management.NotificationListener;
import javax.management.AttributeChangeNotification;

public class SimpleStandardListener implements NotificationListener {
    [...]

    // Implementation of the NotificationListener interface
    //
    public void handleNotification(Notification notification,
                                   Object handback) {

        // Process the different types of notifications fired by the
        // simple standard MBean.
        String type = notification.getType();

        System.out.println(
            "\n\t>> SimpleStandardListener received notification:" +
            "\n\t>> ---------------------------------------------");
        try {
            if (type.equals(AttributeChangeNotification.ATTRIBUTE_CHANGE)) {
```

EXAMPLE 8–4 Listener for Attribute Change Notifications *(Continued)*

```
  System.out.println("\t>> Attribute \"" +
      ((AttributeChangeNotification)notification).getAttributeName()
      + "\" has changed");
  System.out.println("\t>> Old value = " +
      ((AttributeChangeNotification)notification).getOldValue());
  System.out.println("\t>> New value = " +
      ((AttributeChangeNotification)notification).getNewValue());

            }
            else {
                System.out.println("\t>> Unknown event type (?)\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Again, we are handling a subclass of the `Notification` class, this one specific to attribute change notifications. The `AttributeChangeNotification` class provides methods for extracting the information about the attribute, notably its name, its type and its values before and after the modification. Our handler does nothing more than display these to the user. If this handler were part of an MBean in a larger management solution, it would probably take some action, depending upon the change in value of the attribute.

As demonstrated by the broadcaster's code (see Example 8–3), the subclass can easily be instantiated and sent instead of a `Notification` object. Its constructor provides parameters for initializing all of the attribute-related values. In our example, we do not use significant values for the `sequenceNumber` and `timeStamp` parameters because our listener has no need for them. One great advantage of Java DMK is that you only need to implement the level of functionality that you require for your management solution.

8.3.3 Adding a Listener Directly to an MBean

There is nothing that statically indicates that our MBean sends attribute change notifications. In our case it is a design decision, meaning that we know that the listener will receive attribute change notifications because we wrote the MBean that way. At runtime, the MBean server exposes the list of notifications in this MBean's metadata object, allowing a manager that is interested in attribute changes to register the appropriate listener.

Being confined to the agent, our example is much simpler. First we instantiate and register our simple MBean with the agent's MBean server. Then, because we have designed them to work together, we can add our listener for attribute changes to our MBean by calling the `addNotificationListener` method on the MBean server.

EXAMPLE 8–5 Registering for Attribute Change Notifications

```
Agent myAgent = new Agent();
AgentListener agentListener = null;
SimpleStandard simpleStd = null;
ObjectName simpleStdObjectName = null;
SimpleStandardListener simpleStdListener = null;

[...]
try {
    simpleStdObjectName =
        new ObjectName("simple_mbean:class=SimpleStandard");
    simpleStd = new SimpleStandard();
    myAgent.server.registerMBean(simpleStd, simpleStdObjectName);
} catch(Exception e) {
    e.printStackTrace();
    System.exit(1);
}

echo("\nRegistering the Simple Standard MBean listener...");
try {
     simpleStdListener = new SimpleStandardListener();
     myAgent.server.addNotificationListener(simpleStdObjectName,
                                            simpleStdListener,
                                            null,
                                            null);
} catch (Exception e) {
     e.printStackTrace();
     System.exit(1);
}
echo("done");
```

There are two major implications when adding listeners directly rather than doing so using the MBean server:

- Notification objects, or in this case subclasses, contain a direct reference to the broadcaster object. This means that their `getSource` method returns a reference to the broadcaster instead of its object name. Our listener is unaffected by this issue because it never calls this method.
- This listener will need to be removed directly from the MBean instance. A listener added directly to the broadcaster object cannot be removed through the MBean server's methods, and vice versa.

The rest of the agent object's code performs the setup of the agent's MBean server and various input and output for running the example. Similar agents are presented in detail earlier in Part II.

8.4 Running the Agent Notification Example

Now that we have seen all of our notification broadcaster objects and all of our listener handlers, we are ready to run the example.

The *examplesDir*/current/Notification directory contains all of the files for the simple MBean, the listener objects, and the agent. When started, the agent application adds the MBean server delegate listener first, so that a notification can be seen for the creation of the SimpleStandard MBean. Attribute change notifications are triggered by invoking methods using the HTML adaptor.

▼ To Run the Agent Notification Example

1. **Compile all files in this directory with the javac command.**

   For example, on the Solaris platform with the Korn shell, type:

   ```
   $ cd examplesDir/current/Notification/
   $ javac -classpath classpath *.java
   ```

2. **To run the example, start the agent application:**

   ```
   $ java -classpath classpath Agent
   ```

3. **After the agent application has started and added the MBean server delegate listener, press Enter to create the simple MBean.**

   Before the next printout of the agent application, you should see the text generated by the AgentListener class. Its handler method has been called with an MBean creation notification, and it prints out the object name of our new MBean.

4. **Now that the simple MBean is registered and the SimpleStandardListener has been added as a listener, trigger attribute change notifications by invoking the reset operation through the HTML adaptor.**

   a. **Load the following URL in your browser:**

      http://localhost:8082/

      If you get an error, you might have to switch off proxies in your preference settings or substitute your host name for localhost. Any browser on your local network can also connect to this agent by using your host name in this URL.

   b. **In the operations table of our MBean view, click on the reset button.**

      Every time you do this, you should see the output of the attribute change listener in the terminal window where you launched the agent.

5. **When you have finished with the attribute change notifications, press `Enter` in the agent’s terminal window to remove our simple MBean.**

   Again, the output of the MBean server delegate listener is displayed. This time it has detected that our MBean has been unregistered from the MBean server.

6. **Press `Enter` again to stop the agent application.**

PART III

Remote Management Applications

In Part II, we saw how to access and manage a Java dynamic management agent through the HTML protocol adaptor. Protocol adaptors provide a view of an agent through communication protocols. In this part, we present protocol connectors and proxy MBeans for managing agents programmatically.

The Java Dynamic Management Kit (Java DMK) 5.1 integrates the Java Management Extensions (JMX) Remote API specification to provide the means to develop remote management applications in the Java programming language. These remote applications establish connections with agents through protocol connectors over remote method invocation (RMI) or over a new custom protocol, the JMX messaging protocol (JMXMP). Connections are established between a client object and a server object, via one of these protocols. The connector client object exposes a remote version of the MBean server interface. The connector server object in the agent transmits management requests to the MBean server and forwards any replies.

Connectors enable you to develop a management application that is both protocol independent and location-independent. Once the connection is established, the communication layer is transparent, and the manager can issue requests as if it were directly calling the MBean server. Using proxy objects that represent MBeans simplifies the design of the management application and reduces development time.

This homogeneity of the API makes it possible to develop portable management applications that can run either in an agent or in a remote management application. This simplifies the development and testing of applications, and it also allows functionality to evolve along with the management solution. As your agent and manager platforms evolve, management policies can be implemented at higher levels of management, and intelligent logic for monitoring and processing can be moved down into agents.

This part contains the following chapters:

- Chapter 9 shows how *protocol connectors* establish a connection between a management application and a Java dynamic management agent. Once the connection is established, the remote management application can access the

MBeans in the agent.

- Chapter 10 describes the three bindings to existing lookup services implemented by Java DMK, which allow connector servers to advertise themselves to connector clients, and allow clients to find connector servers.
- Chapter 11 presents the *security features* that can be enabled for a given connection. Both the connectors provide different security mechanisms, with the JMXMP connector also offering simple authentication and security layer (SASL) mechanisms.

CHAPTER 9

Protocol Connectors

Protocol connectors provide a point-to-point connection between a Java dynamic management agent and a management application. Each connector relies on a specific communication protocol, but the API that is available to the management application is identical for all connectors and is entirely protocol-independent.

A connector consists of a connector server component registered in the agent and a connector client object instance in the management application. The connector client exposes a remote version of the MBean server interface. Each connector client represents one agent to which the manager wants to connect. The connector server replies to requests from any number of connections and fulfills them through its MBean server. Once the connection is established, the remoteness of the agent is transparent to the management application, except for any communication delays.

Connectors rely on the Java serialization package to transmit data as Java objects between client and server components. Therefore, all objects needed in the exchange of management requests and responses must be instances of a serializable class. However, the data encoding and sequencing are proprietary, and the raw data of the message contents in the underlying protocol are not exposed by the connectors.

Java Dynamic Management Kit (Java DMK) version 5.1 implements the new connectors defined by Java Management Extensions (JMX) Remote API. The connectors implemented in previous versions of Java DMK are retained for backwards compatibility, and are described in Part VI. The two new connectors are based on the remote method invocation (RMI) protocol, and a new protocol called the JMX messaging protocol (JMXMP). These new connector protocols allow more sophisticated security than was previously possible to be implemented on the connectors, and are also less complicated to implement.

The code samples in this chapter are taken from the files in the `current/Connectors` example directory located in the main *examplesDir* (see Directories and Classpath in the Preface).

This chapter covers the following topics:

- "9.1 Connector Servers" on page 124 presents the connector servers defined by Java DMK.
- "9.2 Connector Clients" on page 127 presents the connector clients.
- "9.3 Examples of Connector Servers" on page 131 describes how to run the two connector server examples.
- "9.4 Remote Notifications and Heartbeat Mechanism" on page 133 explains how a connector performs notifications and monitors a connection.
- "9.5 Wrapping Legacy Connectors" on page 133 explains how you can wrap legacy connectors so they can be used through the new connector infrastructure.

9.1 Connector Servers

Connector servers on the agent side listen for management requests issued through a corresponding connector client. The connector server transmits these requests to its MBean server and forwards any response back to the management application. The connector server also forwards notifications, when the management application has registered to receive them through its connector client.

A connector server listens for incoming requests from its corresponding connector client, decodes that request and encodes the reply. Several connector clients can establish connections with the same connector server, and the connector server can handle multiple requests simultaneously. There only needs to be one connector server MBean per protocol to which the agent needs to respond. However, several connector servers for the same protocol can coexist in an agent for processing requests on different ports.

9.1.1 Connector Server Factories

The simplest way to create connector servers is to use the `JMXConnectorServerFactory` constructor defined by JMX Remote API. No instances of this class are ever created, but its single method, `newJMXConnectorServer()` is called to create instances of the `JMXConnectorServer` class. New `JMXConnectorServer` instances are created when `newJMXConnectorServer()` is passed the following parameters:

1. A `JMXServiceURL`, that serves as the address for the connector server; this URL identifies the connector protocol to be implemented, and the machine name and port number (or path or URL) at which this connector server will be bound.
2. The environment `Map` for `newJMXConnectorServer()`; this can be null.

3. A reference to the MBean server that will be exposed for remote management through this connector server. If null is given at creation time, the MBean server that will be exposed to remote management will be the MBean server in which the connector server is later registered as an MBean.

   Note that if you supply an MBean server at creation time, you can optionally register the connector server as an MBean. In that case, the MBean server in which you choose to register the connector server can optionally be the same MBean server as the one that was supplied at creation time. However, if you supply a null value for this parameter, you will need to register the connector server as an MBean in the MBean server that you wish to expose through that connector server.

In the `JMXServiceURL`, the connector protocol specified can be either RMI or JMXMP. Depending which protocol is specified, the connector server created will be either an instance of the `RMIConnectorServer` or `JMXMPConnectorServer` classes. Both of these classes inherit from the `JMXConnectorServer` class.

9.1.2 RMI Connector Server

The Java DMK RMI connector server supports the standard RMI transports, Java Remote Method Protocol (JRMP) and the Internet Inter-Object Request Broker (ORB) Protocol (IIOP). An example of an RMI connector is provided in the *examplesDir* that demonstrates an RMI connection between a server and a remote client.

The `Server` class from the RMI connector example is is shown in Example 9–1.

EXAMPLE 9–1 RMI Connector Server

```
public static void main(String[] args) {
      try {
          // Instantiate the MBean server
          //
          MBeanServer mbs = MBeanServerFactory.createMBeanServer();

          // Create an RMI connector server
          //
          JMXServiceURL url = new JMXServiceURL(
            "service:jmx:rmi:///jndi/rmi://localhost:9999/server");
          JMXConnectorServer cs =
               JMXConnectorServerFactory.newJMXConnectorServer(url,
               null, mbs);
          cs.start();
        } catch (Exception e) {
          e.printStackTrace();
        }
   }
}
```

Firstly, the `Server` class creates a new MBean server called `mbs` by calling the `createMBeanServer()` method of the `MBeanServerFactory` class. A call to `JMXServiceURL` creates a new service URL called `url`, which serves as an address for the connector server. This service URL defines the following:

- The connector will use the default RMI transport, JRMP, denoted by `rmi`.
- The RMI registry in which the RMI connector stub will be stored will be running on port `9999` on the local host, and the server will be registered under the name `server`. The port `9999` specified in the example is arbitrary; you can use any available port.

Finally, an RMI connector server named `cs` is created by calling the `JMXConnectorServerFactory` constructor, with the service URL `url`, a null environment map, and the MBean server `mbs` as parameters. The connector server `cs` is launched by calling the `start()` method of `JMXConnectorServer`, whereupon the instance of `RMIConnectorServer` that is created exports its underlying RMI server stub `server` to the RMI registry.

9.1.3 JMXMP Connector Server

The JMXMP connector protocol defined by Java DMK 5.1 is based on Java serialization over transmission control protocol (TCP) sockets. The JMXMP protocol is a custom protocol for JMX Remote API, and offers a more complete security solution than the RMI connector, as it can implement both the secure sockets layer (SSL) and the simple authentication and security layer (SASL) technologies. These optional security features are described in Chapter 11.

In the JMXMP connector, communication between server and client happens over a single TCP connection, and every message is a serialized Java object. Communication between server and client is performed in two separate streams, one for each direction, allowing multiple concurrent requests over the connection at any given time.

The JMXMP connector example is contained in the directory *examplesDir*`/current/Connectors/jmxmp`.

The code for a JMXMP connector server is shown in Example 9–2.

EXAMPLE 9–2 JMXMP Connector Server

```
public class Server {

   public static void main(String[] args) {
      try {
         // Instantiate the MBean server
         //
         MBeanServer mbs = MBeanServerFactory.createMBeanServer();

         // Create a JMXMP connector server
         //
```

EXAMPLE 9–2 JMXMP Connector Server *(Continued)*

```
         JMXServiceURL url =
              new JMXServiceURL("jmxmp", null, 5555);
         JMXConnectorServer cs =
              JMXConnectorServerFactory.newJMXConnectorServer(url,
              null, mbs);
         cs.start();
       } catch (Exception e) {
         e.printStackTrace();
       }
    }
}
```

Firstly, the `Server` class creates a new MBean server named `mbs` by calling the `createMBeanServer()` method of the `MBeanServerFactory` class.

A call to `JMXServiceURL` creates a new service URL called `url`, which serves as an address for the connector server. This service URL defines the following:

1. The connector will use the JMXMP protocol, denoted by `jmxmp`.
2. No environment map is specified, as denoted by `null`.
3. The connector server will listen for client connections on port `5555` on the local host.

Finally, a JMXMP connector server named `cs` is created by calling the constructor `JMXConnectorServerFactory`, with the service URL `url`, the null environment map, and the MBean server `mbs` as parameters. The connector server `cs` is launched by calling the `start()` method of `JMXConnectorServer`, whereupon `JMXMPConnectorServer`, which inherits from `JMXConnectorServer`, starts listening for client connections.

9.2 Connector Clients

The manager application interacts with a connector client to access an agent through an established connection. A connector client provides methods to handle the connection and access the agent.

Through the connector, the management application sends management requests to the MBeans located in a remote agent. Components of the management application access remote MBeans by calling the methods of the connector client for getting and setting attributes and calling operations on the MBeans. The connector client then returns the result, providing a complete abstraction of the communication layer.

9.2.1 Connector Factories

The simplest way to create connectors is to use the JMXConnectorFactory constructor defined by JMX Remote API. No instances of this class are ever created, but its method, connect() can be called to create a connected JMX connector using a JMXServiceURL that you pass it as a parameter. New, unconnected, JMXConnector instances can also be created by calling the other JMXConnectorFactory method, newJMXConnector(), and passing it the connector server's JMXServiceURL. An environment map can also be supplied to provide additional parameters.

The Client in these examples uses the JMXConnectorFactory.connect() method.

9.2.2 RMI Connector Client

The RMI connector Client example is shown in Example 9–3.

EXAMPLE 9–3 RMI Connector Client

```
public class Client {

  public static void main(String[] args) {
    try {
      // Create an RMI connector client
      //
      JMXServiceURL url = new JMXServiceURL(
         "service:jmx:rmi:///jndi/rmi://localhost:9999/server");
      JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
      MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();

      // Get domains from MBeanServer
      //
      String domains[] = mbsc.getDomains();
      for (int i = 0; i < domains.length; i++) {
          System.out.println("Domain[" + i + "] = " + domains[i]);
      }

      String domain = mbsc.getDefaultDomain();

      // Create SimpleStandard MBean and perform simple MBean operations
      //
      ObjectName mbeanName =
             new ObjectName("MBeans:type=SimpleStandard");
      mbsc.createMBean("SimpleStandard", mbeanName, null, null);
      System.out.println("\nMBean count = " + mbsc.getMBeanCount());
      System.out.println("\nState = " +
             mbsc.getAttribute(mbeanName, "State"));
      mbsc.setAttribute(mbeanName,
             new Attribute("State", "changed state"));

      SimpleStandardMBean proxy = (SimpleStandardMBean)
```

EXAMPLE 9–3 RMI Connector Client *(Continued)*

```
          MBeanServerInvocationHandler.newProxyInstance(
                                     mbsc,
                                     mbeanName,
                                     SimpleStandardMBean.class,
                                     false);
      System.out.println("\nState = " + proxy.getState());

      ClientListener listener = new ClientListener();
      mbsc.addNotificationListener(mbeanName, listener, null, null);

      mbsc.invoke(mbeanName, "reset", null, null);

      mbsc.removeNotificationListener(mbeanName, listener);
      mbsc.unregisterMBean(mbeanName);
      jmxc.close();
    } catch (Exception e) {      e.printStackTrace();
    }
  }
}
```

In this example, the `Client` creates an RMI connector client that is configured to connect to the RMI connector server created by `Server` in "9.1.2 RMI Connector Server" on page 125.

As you can see, the `Client` defines the same service URL `url` as that defined by `Server`. This allows the connector client to retrieve the RMI connector server stub named `server` from the RMI registry running on port `9999` of the local host, and to connect to the RMI connector server.

With the RMI registry thus identified, the connector client can be created. The connector client, `jmxc`, is an instance of the JMX Remote API interface `JMXConnector`, created by the `connect()` method of `JMXConnectorFactory`. The `connect()` method is passed the parameters `url` and a null environment map when it is called.

An instance of `MBeanServerConnection`, named `mbsc`, is then created by calling the `getMBeanServerConnection()` method of the `JMXConnector` instance `jmxc`.

The connector client is now connected to the MBean server created by `Server`, and can create MBeans and perform operations on them with the connection remaining completely transparent to both ends.

In the examples directory, there is an MBean interface and a class to define an MBean called `SimpleStandard`. As the name suggests, this is a very basic MBean of the type described in Chapter 1. The connector client creates an instance of this `SimpleStandard` MBean and registers it in the MBean server with a call to the `createMBean()` method of `MBeanServerConnection`. The client then activates notifications by calling `addNotificationListener()`, and performs the operations defined by `SimpleStandard` as if they were local MBean operations.

Finally, the client unregisters the SimpleStandard MBean and closes the connection.

9.2.3 JMXMP Connector Client

The Client.java class is shown in Example 9–4. The only difference between the client in this example and that used in the RMI connector example is in the JMX service URL. The operations this example performs on the SimpleStandard MBean are identical to those performed in the RMI connector example. Consequently, the code has been abridged.

EXAMPLE 9–4 JMXMP Connector Client

```
public class Client {

   public static void main(String[] args) {
     try {
        // Create a JMXMP connector client
        //
        System.out.println("\nCreate a JMXMP connector client");
        JMXServiceURL url =
            new JMXServiceURL("service:jmx:jmxmp://localhost:5555");
        JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
        MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();

        // Get domains from MBeanServer, create SimpleStandard MBean
        //
        String domains[] = mbsc.getDomains();
        for (int i = 0; i < domains.length; i++) {
            System.out.println("Domain[" + i + "] = " + domains[i]);
        ObjectName mbeanName = new ObjectName(
             "MBeans:type=SimpleStandard");
        mbsc.createMBean("SimpleStandard", mbeanName, null, null);

       // Perform simple MBean operations, add listener, reset MBean,
       // remove listener, unregister MBean

       [...]

       // Close MBeanServer connection
       //
       jmxc.close();
     } catch (Exception e) {
       e.printStackTrace();
     }
   }
}
```

The Client.java class creates a JMXMP connector client that is configured to connect to the JMXMP connector server created by Server.

As you can see, Client defines a service URL url that allows the connector client to find the connector server.

The connector client, jmxc, is an instance of the JMXConnector interface, created by the connect() method of JMXConnectorFactory. The connect() method is passed the parameter url when it is called.

An instance of MBeanServerConnection, named mbsc, is then created by calling the getMBeanServerConnection() method of the JMXConnector instance jmxc.

The connector client is now connected to the MBean server created by Server, and can create MBeans and perform operations on them with the connection remaining completely transparent to both ends.

The client creates and registers the SimpleStandard MBean in the MBean server with a call to the createMBean() method of MBeanServerConnection, and performs the operations defined by SimpleStandard as if they were local MBean operations.

Finally, the client unregisters SimpleStandard and closes the connection.

9.3 Examples of Connector Servers

The connector server examples demonstrate the following:

1. The server:
   a. Creates an MBean server
   b. Creates an RMI or a JMXMP connector server
2. The remote client:
   a. Creates an RMI or a JMXMP connector
   b. Creates a simple standard MBean in the MBean server, via the RMI connection
   c. Creates a generic notification listener.
3. Basic operations are performed on the MBean registered in the MBean server, via the RMI connection

9.3.1 RMI Connector Server Example

The RMI connector server example is found in the *examplesDir*/current/Connectors/rmi directory.

▼ To Run the RMI Connector Server Example

1. **Compile the Java classes.**

```
$ javac -classpath classpath *.java
```

2. **Start an RMI registry on port 9999 of the local host.**

   ```
   $ export CLASSPATH=.:classpath ; rmiregistry 9999 &
   ```

3. **Start the RMI connector server:**

   ```
   $ java -classpath .:classpath Server &
   ```

   You will see confirmation of the creation of the MBean server and the RMI connector server.

4. **Start the RMI connector client:**

   ```
   $ java -classpath .:classpath Client
   ```

   You will see confirmation of the creation of the RMI connector client and of the connection with the connector server. You will also be informed of the domain name, and the creation and registration of `SimpleStandard` MBean. The client will then perform operations on `SimpleStandard` MBean, before unregistering it.

9.3.2 JMXMP Connector Server Example

The JMXMP connector server example is found in the *examplesDir*`/current/Connectors/jmxmp` directory.

▼ To Run the JMXMP Connector Server Example

1. **Compile the Java classes.**

   ```
   $ javac -classpath classpath *.java
   ```

2. **Start the JMXMP connector server:**

   ```
   $ java -classpath .:classpath Server &
   ```

   You will see confirmation of the creation of the MBean server and the JMXMP connector server.

3. **Start the JMXMP connector client:**

   ```
   $ java -classpath .:classpath Client
   ```

   You will see confirmation of the creation of the RMI connector client and of the connection with the connector server. You will also be informed of the domain name, and the creation and registration of `SimpleStandard` MBean. The client will then perform operations on `SimpleStandard` MBean, before unregistering it.

9.4 Remote Notifications and Heartbeat Mechanism

In previous versions of Java DMK, notification forwarding over remote connectors and the heartbeat mechanism to monitor the health of the connections were performed by custom implementations written specifically for Java DMK. These mechanisms have now been standardized by the implementation of JMX Remote API in Java DMK 5.1, and consequently are integrated into the RMI and JMXMP connectors described in the preceding sections of this chapter.

9.5 Wrapping Legacy Connectors

Although it is recommended that you use the new RMI and JMXMP connector protocols defined by the JMX Remote API, it is possible for you to continue to use your existing legacy connectors alongside the new ones. This is achieved by *wrapping* the legacy connector so that it appears in a form that is compatible with the new standard connectors. Wrapping your Java DMK 5.0 RMI and HTTP(S) connectors allows applications created using Java DMK 5.1 to interoperate with existing Java DMK applications. In addition, if you want to use HTTP(S) connectors, you must wrap them.

The `JmxConnectorServerFactory` and `JmxConnectorFactory` classes are used to create wrapped legacy Java DMK connector servers and clients. These connector servers and clients expose the same interfaces as standard JMX connectors. For the JmxConnectorServerFactory to create a wrapped connector, you must ensure that the jdmkrt.jar is either in your `CLASSPATH` environment variable, or in the context of the thread that is used to create the wrapped connector.

Java DMK 5.1 defines a new interface, `JdmkLegacyConnector`, that is used to obtain the wrapped connectors.

The `JdmkLegacyConnector` interface specifies a new protocol name for each of the legacy connectors. These protocol names are passed into the `JMXServiceURL` when it is created, in the same way the RMI connector and JMXMP connectors are identified as `rmi` and `jmxmp` respectively in the service URLs. The new protocol names are listed below.

- `jdmk-http`, for the legacy HTTP connector.
- `jdmk-https`, for the legacy HTTPS connector.
- `jdmk-rmi`, for the legacy RMI connector.

The JdmkLegacyConnector also specifies a list of properties to allow the factory to obtain information defined by the user to create the legacy connectors, as follows.

- com.sun.jdmk.http.server.authinfo.list, specifying the list of AuthInfo[] for an HTTP or HTTPS server.
- com.sun.jdmk.client.localhost, specifying a String object as a local host name for an HTTP, HTTPS or RMI client.
- com.sun.jdmk.http.client.authinfo, specifying an AuthInfo[] object for an HTTP or HTTPS client to connect to its server.
- com.sun.jdmk.http.server.authinfo.list, a list of AuthInfo for an HTTP or HTTPS connector server.
- com.sun.jdmk.http.server.local.address, specifying the local InetAddress the HTTP/HTTPS connector server will bind to.

The creation of wrapped legacy RMI and HTTP connector servers is shown in Example 9–5. The code extracts in this section are taken from the classes in the *examplesDir*/current/Connectors/wrapping directory.

EXAMPLE 9–5 Wrapping Legacy Connector Servers

```
public class Server {

    public static void main(String[] args) {
         try {
           MBeanServer mbs = MBeanServerFactory.createMBeanServer();

           JMXServiceURL httpURL = new JMXServiceURL("jdmk-http",
                                                      null, 6868);
           JMXConnectorServer httpCS =
              JMXConnectorServerFactory.newJMXConnectorServer(httpURL,
                                                              null, mbs);
           ObjectName httpON =
               new ObjectName("legacyWrapper:protocol=jdmk-http,port=6868");
               mbs.registerMBean(httpCS, httpON);

            httpCS.start();

            JMXServiceURL rmiURL =
               new JMXServiceURL("jdmk-rmi", null, 8888, "/myRMI");
            JMXConnectorServer rmiCS =
                JMXConnectorServerFactory.newJMXConnectorServer(rmiURL,
                                                                null, mbs);

                ObjectName rmiON =
                new ObjectName("legacyWrapper:protocol=jdmk-rmi,port=8888");
                mbs.registerMBean(rmiCS, rmiON);

            rmiCS.start();

        } catch (Exception e) {
            e.printStackTrace();
        }
```

EXAMPLE 9–5 Wrapping Legacy Connector Servers *(Continued)*

```
    }
}
```

In Example 9–5, we see that the connector servers are created in a similar way to standard connector servers defined by the JMX Remote API. The only difference is that the connector protocols used to create the JMX service URLs are `jdmk-rmi` and `jmdk-http`. For simplicity, this example does not create an HTTPS connector.

Note – As you can see above, the wrapped connector server is registered in the MBean server before it is started. This order of events must always be respected, otherwise the wrapped connector will not work.

EXAMPLE 9–6 Wrapping Legacy Connector Clients

```
public class Client {

  public static void main(String[] args) {
       try {
           JMXServiceURL url = new JMXServiceURL("jdmk-http",
                                              null, 6868);
           connect(url);

           url = new JMXServiceURL("jdmk-rmi", null, 8888,
                                "/myRMI");
           connect(url);
    } catch (Exception e) {
        e.printStackTrace();
    }
       System.exit(0);
  }

  [...]
```

As for the creation of the connector servers, in Example 9–6, the legacy connector clients are created by passing the `jdmk-http` and `jdmk-rmi` property strings to the JMX service URLs. The rest of the `Client` example is identical to a standard JMX Remote API connector client.

▼ To Run the Legacy Connector Wrapping Example

The following example uses the classes in the *examplesDir*`/current/Connectors/wrapping` directory.

1. **Compile the Java classes**

```
$ javac -classpath classpath *.java
```

2. **Start the `Server`**

   ```
   $ java -classpath .:classpath Server &
   ```

   You will see confirmation of the creation of the wrapped connector servers, and will be prompted to start the `Client`.

3. **When prompted, start the `Client`**

   ```
   $ java -classpath .:classpath Client
   ```

   You will see the wrapped HTTP connector client connect to the HTTP server, and perform various MBean operations via the connection. Once the MBean operations have completed, the `Client` closes the HTTP connection, before opening a connection to the legacy RMI connector server, and performing further MBean operations.

9.5.1 Limitations of Wrapped Legacy Connectors

The wrapped legacy connectors do not provide the full functionality of either their JMX Remote API or their Java DMK 5.0 counterparts. The limitations of the wrapped legacy connectors are listed below.

- Not all of the methods supported by standard `JMXConnector` and `JMXConnectorServer` are supported by the wrapped legacy connectors.

  The following are the only methods supported by the wrapped legacy connector clients.

  - `addNotificationListener(ObjectName name, ObjectName listener, NotificationFilter filter, Object handback)`
  - `removeNotificationListener(ObjectName name, ObjectName listener)`
  - `removeNotificationListener(ObjectName name, ObjectName listener, NotificationFilter filter, Object handback)`
  - `removeNotificationListener(ObjectName name, NotificationListener listener, NotificationFilter filter, Object handback)`

  The wrapped legacy connector servers only support the `getConnectionIds()` method.
- A legacy connector can only use the class loader that was used to created that connector.
- Legacy servers do not send out events relating to client connections or failures, but only about state changes: `offline`, `starting`, `online`.
- A client can connect to a legacy connector server created by the JMX Remote API factory only after the server has been registered in an MBean server.

- A legacy connector client has no way of showing its local port number. Calling `Client.getConnectionId()` will return:

  protocol`://`*host*`:`*server_port* *count*

  Where the count is an integer ID that is unique within a Java virtual machine on the client side. For example:

  `jdmk-http://`*host_name*`:6666 1`

- A legacy connector server must be registered as an MBean in the MBean server which is exposed through that connector server, before the server can be started.

CHAPTER **10**

Lookup Services

The Java Management Extensions (JMX) Remote API specification defines three bindings to lookup services, using existing lookup technologies, as described in the following sections:

- "10.1 Initial Configuration" on page 139 explains the initial configurations will need to make to set up the three types of lookup service, depending on what types of registry you wish to use.
- "10.2 Service Location Protocol (SLP) Lookup Service" on page 142 describes the service location protocol lookup service.
- "10.3 Jini Lookup Service" on page 153 demonstrates the Jini lookup service.
- "10.4 Java Naming and Directory Interface (JNDI) / LDAP Lookup Service" on page 163 shows how to look up connector servers using the Java naming and directory interface (JNDI™) service over a lightweight directory access protocol (LDAP) backend.

This chapter provides an introduction to the lookup services provided by Java DMK using three sets of examples. For a full description of the lookup services, see the *JMX Remote API 1.0 Specification* document.

10.1 Initial Configuration

As shown simply in Chapter 9, if you are using remote method invocation (RMI) connectors, you can choose to use an external directory to register the connector server stubs you want to look up. The following cases are presented in the lookup service examples relating to RMI connectors:

- RMI connectors using one of the following external directories:
  - An RMI registry, for RMI connectors implementing the default Java Remote Method Protocol (JRMP) transport.

- CORBA Naming Service, for RMI connectors implementing the Internet Inter-ORB Protocol (IIOP) transport.
- LDAP, for both the IIOP and JRMP transports.

- RMI connectors without an external directory.

If you choose to register the RMI connector stubs in an external directory, some initial configuration is required, to set up your RMI registry, CORBA naming service or LDAP server. If you do not use an external directory, the RMI connector stub is encoded into the JMX service URL.

The lookup service examples for the JMX messaging protocol (JMXMP) connector do not use any external directories.

The following sections describe the external directories that you can use in conjunction with the lookup service examples that use RMI connectors. These external directories are referred to when running the three examples of lookup services that are given in the subsequent sections in this chapter.

In these examples, the addresses of the different registries are given as URLs in JNDI form. For an explanation of JNDI form, see the API documentation for the `javax.management.remote.rmi` package. If you want to run the external directories on a machine other than the local machine, you must specify that machine's host name instead of `localhost`.

10.1.1 External RMI Registry

To register the RMI connector server stubs in an external RMI registry, for use by connectors implementing the JRMP transport, start an RMI registry with the following command:

```
$ rmiregistry 9999 &
```

For your convenience when typing commands, create an environment variable for the address of the RMI registry.

```
$ jndirmi="rmi://localhost:9999"
```

10.1.2 External CORBA Naming Service

To register connector stubs in the CORBA naming service, you must start an ORB daemon, with the following commands:

```
$ rm -rf ./orb.db
$ orbd -ORBInitialPort 7777 &
```

For your convenience when typing commands, create an environment variable for the address of the CORBA naming service.

```
$ jndiiiop="iiop://localhost:7777"
```

10.1.3 External LDAP registry

To register connector stubs in an LDAP registry, you must start an LDAP server. The LDAP server you use is your choice, although the schema for representing Java objects in an LDAP directory must be known to the server. See the Request For Comments (RFC) document RFC 2713for details.

Once you have started your LDAP server, in to create a directory context under which you have the permission to create new nodes, create a new component suffix:

```
dc=Test
```

See the documentation accompanying your LDAP server for details of how to configure the server and create this suffix. Alternatively, if you already have the appropriate rights of node creation under an existing node, you can use that node instead. In that case, you must replace `dc=Test` with the name of your node wherever `dc=Test` appears in these examples.

For your convenience, set the following LDAP parameters as environment variables:

- The name of the machine running your LDAP server (*ldap_host*):

  ```
  $ ldaphost=ldap_host
  ```

- The port the LDAP server is running on (*ldap_port*):

  ```
  $ ldapport=ldap_port
  ```

- The LDAP common name attribute, which in these examples is "Directory Manager":

  ```
  $ principal="cn=Directory Manager"
  ```

- The password required by your LDAP server:

  ```
  $ credentials=your_ldap_password
  ```

- The address of the LDAP server:

  ```
  $ jndildap="ldap://$ldaphost:$ldapport"
  ```

You are now ready to run the different lookup service examples.

10.2 Service Location Protocol (SLP) Lookup Service

Java DMK 5.1 specifies how to register RMI connectors or JMXMP connectors with the SLP lookup service.

The purpose of this example is to demonstrate how a connector client can find and connect to a connector server that has registered with the SLP lookup service. This example performs the following operations:

1. The agent:
   - Creates an MBean server
   - Gets a pointer to the SLP lookup service
   - Creates a connector server
   - Registers the connector address with the SLP lookup service
2. The client:
   - Gets a pointer to the SLP lookup service
   - Looks for any connector servers registered in the SLP lookup service
   - Creates a JMX Remote API connector
   - Retrieves information about the MBeans in the MBean server

---

Note – This example assumes that you are already familiar with SLP technology. The code provided for this example conforms to Sun Microsystems' implementation of SLP, as defined by RFC 2614. Sun Microsystems' implementation of SLP is available in the Solaris operating environment in the directory `/usr/share/lib/slp`. If you are not running the Solaris operating environment, you must obtain a version of SLP that is compliant with RFC 2614, section 5.

The SLP lookup example is contained in the directory *examplesDir*`/current/Lookup/slp`. For explanations of the SLP code used in this example, see RFC 2614 and the API documentation for SLP; the explanations below concentrate on the Java DMK implementation.

10.2.1 Registering the Connector Server with SLP

Example 10–1 shows the registration of a connector server's URL with the SLP lookup service's `Advertiser`. This code is taken from the `Server` class in the SLP example directory.

EXAMPLE 10–1 Registering the Connector Server's Address with the SLP Advertiser

```
public class Server {
   public final static int JMX_DEFAULT_LEASE = 300;
   public final static String JMX_SCOPE = "DEFAULT";

   private final MBeanServer mbs;
   public Server() {
       mbs = MBeanServerFactory.createMBeanServer();
   }

[...]

   public static void register(JMXServiceURL jmxUrl, String name)
     throws ServiceLocationException {
     ServiceURL serviceURL =
          new ServiceURL(jmxUrl.toString(),
                         JMX_DEFAULT_LEASE);
     debug("ServiceType is: " + serviceURL.getServiceType());
     Vector attributes = new Vector();
     Vector attrValues = new Vector();
     attrValues.add(JMX_SCOPE);
     ServiceLocationAttribute attr1 =
          new ServiceLocationAttribute("SCOPE", attrValues);
     attributes.add(attr1);
     attrValues.removeAllElements();
     attrValues.add(name);
     ServiceLocationAttribute attr2 =
          new ServiceLocationAttribute("AgentName", attrValues);
     attributes.add(attr2);
     final Advertiser slpAdvertiser =
          ServiceLocationManager.getAdvertiser(Locale.US);
     slpAdvertiser.register(serviceURL, attributes);

   }

[...]
```

Examining this code excerpt, we see that the SLP lease `JMX_DEFAULT_LEASE` is set to a default lease of 300 seconds, which corresponds to the length of time the URL will be registered, and the initial creation of the MBean server `mbs` with a call to `MBeanServerFactory.createMBeanServer()`. In code that is not shown here, an SLP advertiser `slpAdvertiser`, and an SLP service URL `url` are defined. The `slpAdvertiser` is used to register the service URL in the SLP lookup service.

The service URL `jmxUrl` is the address of the connector server, and is obtained by a call to the `getAddress()` method of `JMXConnectorServer` when the connector server is started.

The SLP lookup attribute, namely the agent name under which the connector server address is to be registered (name), is then specified by the SLP class ServiceLocationAttribute. The AgentName attribute is mandatory, but other optional attributes, such as ProtocolType, AgentHost, and Property can also be registered in the SLP lookup service.

Finally, the JMX connector server address is registered in the SLP advertiser service with a call to the register() method of the Advertiser interface, with the serviceURL and the attributes passed in as parameters.

Now that the connector server's address has been advertised, the connector server itself is created and registered with SLP, as shown in Example 10–2.

EXAMPLE 10–2 Registering the Connector Server in the SLP Lookup Service

```
[...]

   public JMXConnectorServer rmi(String url) throws
     IOException,
     JMException,
     NamingException,
     ClassNotFoundException,
     ServiceLocationException {
     JMXServiceURL jurl = new JMXServiceURL(url);
     final HashMap env = new HashMap();
     // Environment map attributes
     [...]


     JMXConnectorServer rmis =
        JMXConnectorServerFactory.newJMXConnectorServer(jurl, env, mbs);
     final String agentName = System.getProperty("agent.name",
                                                  "DefaultAgent");
     start(rmis, agentName);

     return rmis;
  }

   public void start(JMXConnectorServer server, String agentName)
      throws IOException, ServiceLocationException {
      server.start();
      final JMXServiceURL address = server.getAddress();
      register(address,agentName);
   }
```

The service URL jurl is constructed from the string url that will be included in the command used to launch the Server at the command line. An RMI connector server named rmis is then created with the system properties defined by the environment map and the address jurl.

The connector server is then started, and the RMI connector server address is registered in the SLP lookup service under the name `agentName`. Subsequent code not shown here creates a corresponding JMXMP connector server named `jmxmp`, that is also registered with the SLP service.

10.2.2 Looking up the Connector Server

The following code examples are taken from the `Client` class in the *examplesDir*`/current/Lookup/slp` directory.

EXAMPLE 10–3 Retrieving the List of Connector Servers

```
public class Client {

    public final static String JMX_SCOPE = "DEFAULT";

    public static Locator getLocator() throws ServiceLocationException {
      final Locator slpLocator =
          ServiceLocationManager.getLocator(Locale.US);
      return slpLocator;
    }

      public static List lookup(Locator slpLocator, String name)
          throws IOException, ServiceLocationException {


          final ArrayList list = new ArrayList();
          Vector scopes = new Vector();

          scopes.add(JMX_SCOPE);
          String query =
              "(&(AgentName=" + ((name!=null)?name:"*") + "))";

          ServiceLocationEnumeration result =
              slpLocator.findServices(new ServiceType("service:jmx"),
                                      scopes, query);

          while(result.hasMoreElements()) {
                final ServiceURL surl = (ServiceURL) result.next();


             JMXServiceURL jmxUrl = new JMXServiceURL(surl.toString());
             try {
                  JMXConnector client =
                     JMXConnectorFactory.newJMXConnector(jmxUrl,null);
                  if (client != null) list.add(client);
             } catch (IOException x ) {
             [...]
             }
          }
      }
      return list;
```

EXAMPLE 10–3 Retrieving the List of Connector Servers *(Continued)*

```
    }
```

Example 10–3 first of all obtains the SLP service `Locator` by calling the `getLocator` method of the SLP class `ServiceLocationManager`. `Client` then retrieves all the connector servers registered in the SLP service under a given agent name, or under agent names that match a certain pattern. If no agent name is specified when the `Client` is started, all agent names will be considered.

A JMX Remote API service URL, `jmxUrl`, is generated for each of the agents retrieved by SLP, with each agent's SLP service URL, `surl`, passed as a parameter into the `JMXServiceURL` instance. The URL `jmxUrl` is then passed to the `newJMXConnector()` method of `JMXConnectorFactory`, to create a new connector client named `client` for each agent that is registered in the SLP service.

The connector clients retrieved are stored in an array list called `list`.

EXAMPLE 10–4 Accessing the MBeans in the Remote MBean Server

```
public static void listMBeans(MBeanServerConnection server)
     throws IOException {

     final Set names = server.queryNames(null,null);
     for (final Iterator i=names.iterator(); i.hasNext(); ) {
          ObjectName name = (ObjectName)i.next();
          System.out.println("Got MBean: "+name);
          try {
               MBeanInfo info =
                  server.getMBeanInfo((ObjectName)name);
               MBeanAttributeInfo[] attrs = info.getAttributes();
               if (attrs == null) continue;
               for (int j=0; j<attrs.length; j++) {
                    try {
                         Object o =
                         server.getAttribute(name,attrs[j].getName());
                         System.out.println("\t\t" + attrs[j].getName() +
                         " = "+o);
                    } catch (Exception x) {
                         System.err.println("JmxClient failed to get " +
                                            attrs[j].getName() + x);
                         x.printStackTrace(System.err);
                    }
     }
}
```

In Example 10–4, a reference to the `MBeanServerConnection` is retrieved for every connector client that is created from the connector server address stored in the SLP service. A list of all the MBeans and their attributes is retrieved.

EXAMPLE 10–5 Connecting to the Remote Agents

```
public static void main(String[] args) {
      try {
           final String agentName = System.getProperty("agent.name");
           final Locator slpLocator = getLocator();
           List l = lookup(slpLocator,agentName);
           int j = 1;
           for (Iterator i=l.iterator();i.hasNext();j++) {
                JMXConnector c1 = (JMXConnector) i.next();
                if (c1 != null) {
                    try {
                         c1.connect(env);
                    } catch (IOException x) {
                         System.err.println ("Connection failed: " + x);
                         x.printStackTrace(System.err);
                         continue;
                    }

                    MBeanServerConnection conn =
                         c1.getMBeanServerConnection();

                    try {
                         listMBeans(conn);
                    } catch (IOException x) {
                         x.printStackTrace(System.err);
                    }
                    try {
                         c1.close();
                    } catch (IOException x) {
                         x.printStackTrace(System.err);
                    }
                }
           }
      } catch (Exception x) {
           x.printStackTrace(System.err);
      }
}
```

In Example 10–5, the `agent.name` property is retrieved by calling the `getProperty()` method of the `System` class, and the SLP lookup service is found by calling the `getLocator()` method of `Locator`.

All the agents named `agentName` are then looked up, and connections are made to the agents discovered. If no agent is specified, then all agents are lookup up. Connections are made to the MBean server created by `Server`, and all the MBeans in it are listed, before the connection is closed down.

10.2.3 Running the SLP Lookup Service Example

In addition to the actions you performed in "10.1 Initial Configuration" on page 139, before you can run the lookup service examples that use the SLP, you must perform some further set—up actions that are specific to this example. You can then start looking up connectors using SLP in conjunction with the two connectors supported by Java DMK.

When you run the examples, to help you keep track of which agent has been created with which connector and transport, the agent names include a letter suffix. For example, the agent from the example of an RMI connector over JRMP, without an external directory, is called `test-server-a`.

▼ To Set up the SLP Lookup Service Example

The following steps are required by all of the different connector/transport combinations you can run in this example.

1. **For convenience when compiling and running the classes, define an additional environment variable.**

   In addition to the common environment variables that were set in "10.1 Initial Configuration" on page 139, you need to add the path to the SLP service. If you are using the Solaris operating environment, add the following variables:

   ```
   $ SLP_LIB=/usr/share/lib/slp
   ```

   If you are using another platform, set `SLP_LIB` appropriately for the platform you are using.

2. **Define and export the `classp` environment variable.**

   This example requires a classpath that includes the Java archive (JAR) files for SLP, as well as the JARs for Java DMK, the mandatory and optional parts of the JMX Remote API, and the JMX API reference implementation.

   ```
   classp=$SLP_LIB/slp.jar:classpath:.
   ```

3. **Start the SLP daemon.**

   If you are using the Solaris operating environment, type the following command, which requires you to know your superuser password:

   ```
   $ su root -c "java -cp $SLP_LIB/slpd.jar com.sun.slp.slpd &"
   Password: [type superuser password]
   ```

   If you are not running a Solaris system, start the SLP daemon according to the implementation of SLP you are using.

▼ To Run the SLP Lookup Service Example With an RMI Connector

This example demonstrates the use of the SLP lookup service to look up RMI connector servers that use RMI's default transport, JRMP, as well as the IIOP transport. In addition, as described in "10.1 Initial Configuration" on page 139, different external directories are used to register the RMI connector stubs.

The combinations of transports and external directories demonstrated here are:

- RMI connector over the JRMP transport, with:
  - No external directory
  - An RMI registry
  - An LDAP registry
- RMI connector over the IIOP transport, with:
  - No external directory
  - A CORBA naming service
  - An LDAP registry

Perform the following steps to run the example:

1. **Start the `Server`.**

   The command you use to start the `Server` varies according to which external directory you are using. You can start one or more of the following instances of `Server` with different transports and external registries before starting the `Client`.

   - RMI connector over JRMP, without an external directory:

     ```
     $ java -classpath .:$classp -Ddebug=true \
       -Dagent.name=test-server-a \
       -Durl ="service:jmx:rmi://" \
       slp.Server &
     ```

     In this command:

     - `debug` is set to true to provide more complete screen output when the Server runs
     - The name of the agent to be created is `test-server-a`
     - The service URL specifies that the chosen connector is an RMI connector, running over the RMI default transport JRMP.

     When `Server` is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the SLP service.

   - RMI connector over JRMP, using an RMI registry as an external directory:

     ```
     $ java -classpath .:$classp -Ddebug=true \
       -Dagent.name=test-server-b \
       -Durl="service:jmx:rmi:///jndi/${jndirmi}/server" \
     ```

```
    slp.Server &
```

In this command:

- The name of the agent created is test-server-b
- The service URL specifies the chosen connector as RMI over JRMP, and the external directory in which the RMI connector stub, server, is stored is the RMI registry you identified as jndirmi in "10.1 Initial Configuration" on page 139.

When Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the SLP service.

- RMI connector over JRMP, using LDAP as the external directory:

```
$ java -classpath .:$classp -Ddebug=true \
     -Dagent.name=test-server-c \
     -Durl="service:jmx:rmi:///jndi/${jndildap}/cn=x,dc=Test" \
     -Djava.naming.security.principal="$principal" \
     -Djava.naming.security.credentials="$credentials" \
     slp.Server &
```

In this command:

- The name of the agent created is test-server-c
- The service URL specifies the chosen connector as RMI over JRMP, and the external directory in which the RMI connector stub is stored is the LDAP server you identified as jndildap in "10.1 Initial Configuration" on page 139
- The stub is registered in the Test domain component in the LDAP server.
- The common name attribute principal and password credentials are given to gain access to the LDAP server.

When Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the SLP service under the agent name test-server-c.

- RMI connector over IIOP, without an external directory:

```
$ java -classpath .:$classp -Ddebug=true \
     -Dagent.name=test-server-d \
     -Durl="service:jmx:iiop://" \
     slp.Server &
```

In this command:

- The name of the agent created is test-server-d
- The service URL specifies the chosen connector as RMI connector over IIOP.

When the Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its automatically generated URL in the SLP service.

- RMI connector over IIOP, using CORBA naming as the external directory.

  ```
  $ java -classpath .:$classp -Ddebug=true \
       -Dagent.name=test-server-e \
       -Durl="service:jmx:iiop:///jndi/${jndiiiop}/server" \
       slp.Server &
  ```

  In this command:

  - The name of the agent created is `test-server-e`
  - The service URL specifies the chosen connector as RMI connector over IIOP. The external directory in which the RMI connector stub server is stored is the CORBA naming service you identified as `jndiiiop` in "10.1 Initial Configuration" on page 139.

- RMI connector over IIOP, using LDAP as the external directory.

  ```
  $ java -classpath .:$classp -Ddebug=true \
       -Dagent.name=test-server-f \
       -Durl="service:jmx:iiop:///jndi/${jndildap}/cn=x,dc=Test" \
       -Djava.naming.security.principal="$principal" \
       -Djava.naming.security.credentials="$credentials" \
       slp.Server &
  ```

  In this command:

  - The name of the agent created is `test-server-f`
  - The service URL specifies the chosen connector as RMI over IIOP, and the external directory in which the RMI connector stub is stored is the LDAP server you identified as `jndildap` in "10.1 Initial Configuration" on page 139.
  - The stub is registered in the `Test` domain component in the LDAP server.
  - The common name attribute `principal` and password `credentials` are given to gain access to the LDAP server.

  When `Server` is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the SLP service under the agent name `test-server-f`.

2. **Start the `Client`.**

   After starting the Server using the transport and external directory of your choice, start the Client:

   ```
   $ java -classpath .:$classp -Ddebug=true \
        -Djava.naming.security.principal="$principal" \
        -Djava.naming.security.credentials="$credentials" \
        slp.Client
   ```

   You will see output confirming the detection of the agents created by the `Server` and registered in the lookup service. You will also see the identification and confirmation of the connection made to the agents.

   To look up a specific agent, type the following command:

```
$ java -classpath .:$classp -Ddebug=true \
     -Djava.naming.security.principal="$principal" \
     -Djava.naming.security.credentials="$credentials" \
     -Dagent.name="agentName" \
    slp.Client
```

In the command shown above, *agentName* is the name of the agent you want to look up. You can also specify a partial agent name by using `*`; for example, `x*` for all agent names beginning with the letter x.

▼ To Run the SLP Lookup Service Example With a JMXMP Connector

This example demonstrates the use of the SLP lookup service to look up JMXMP connector servers.

1. **Start the Server.**

   ```
   $ java -classpath .:$classp -Ddebug=true \
        -Dagent.name=test-server-g \
        -Durl="service:jmx:jmxmp://" \
        slp.Server &
   ```

   In this command:

   - The name of the agent created is `test-server-g`
   - The service URL specifies the chosen connector as the JMXMP connector, running on the first available port.

   When the `Server` is launched, you will see confirmation of the creation of the JMXMP connector, and the registration of its automatically generated URL in the SLP service. JMXMP connector servers can be used alongside RMI connectors, and will be detected by the `Client` in exactly the same way as RMI connector servers.

2. **Start the Client.**

   After starting the Server, start the Client:

   ```
   $ java -classpath .:$classp -Ddebug=true slp.Client
   ```

   You will see output confirming the detection of the agents created by the `Server` and registered in the lookup service. You will also see the identification and confirmation of the connection made to the agents.

   To look up a specific agent, type the following command:

   ```
   $ java -classpath .:$classp -Ddebug=true \
     -Dagent.name="agentName" \
     slp.Client
   ```

   In the command shown above, *agentName* is the name of the agent you want to look up. You can also specify a partial agent name by using `*`; for example, `x*` for all agent names beginning with the letter x.

10.3 Jini Lookup Service

The purpose of this example is to demonstrate how a connector client can find and connect to a connector server that has registered with the Jini lookup service. This example performs the following operations:

- The agent:
  - Creates an MBean server
  - Creates a connector server
  - Registers the connector address with the Jini lookup service
- The client:
  - Gets a pointer to the Jini lookup service
  - Looks for any connector servers registered in the Jini lookup service
  - Creates a connector
  - Retrieves information about the MBeans in the MBean server

The Jini lookup service example is contained in the directory *examplesDir*`/current/Lookup/jini`.

Note – These examples assume that you are already familiar with the Jini network technology. The documentation for the Jini network technology is available at `http://wwws.sun.com/software/jini/specs/index.html`. You can download the Jini network technology from the Sun Microsystems Community Source Licensing page, at `http://wwws.sun.com/software/communitysource/jini/download.html`. This example has been implemented using the Jini Technology Starter Kit Version 1.2.1_001.

10.3.1 Registering the Connector Server with the Jini Lookup Service

The following code extract is taken from the `Server` class in the Jini Lookup Service examples directory.

EXAMPLE 10–6 Creating Connector Servers for Registration in the Jini Lookup Service

```
public class Server {
   private final MBeanServer mbs;
   private static boolean debug = false;
   public Server() {
     mbs = MBeanServerFactory.createMBeanServer();
   }
```

EXAMPLE 10–6 Creating Connector Servers for Registration in the Jini Lookup Service *(Continued)*

```
  public JMXConnectorServer rmi(String url)
      throws IOException, JMException, ClassNotFoundException {
      JMXServiceURL jurl = new JMXServiceURL(url);
      final HashMap env = new HashMap();
      // Environment map attributes
     [...]
     JMXConnectorServer rmis =
        JMXConnectorServerFactory.newJMXConnectorServer(jurl, env, mbs);

     final String agentName = System.getProperty("agent.name",
                                                 "DefaultAgent");

     start(rmis,env,agentName);

     return rmis;
   }

[...]
```

Example 10–6 shows the creation of an MBean server `mbs`. As was the case for the SLP examples, the JMX service URL and the agent name are passed to `Server` when it is launched at the command line.

We then see the creation of an RMI connector server named `rmis`, using the system properties defined by the environment map `env` and the address `jurl`. The RMI connector server `rmis` is started. The RMI connector server address will be registered in the Jini lookup service under the name `agentName`. Subsequent code not shown here creates a corresponding JMXMP connector server named `jmxmp`, that will also be registered with the Jini lookup service, as shown in the following code extract.

EXAMPLE 10–7 Registering the Connector Server with the Jini Lookup Service

```
public void start(JMXConnectorServer server, Map env, String agentName)
      throws IOException, ClassNotFoundException {
      server.start();
      final ServiceRegistrar registrar=getRegistrar();
      final JMXConnector proxy = server.toJMXConnector(env);
      register(registrar,proxy,agentName);
   }

   public static ServiceRegistrar getRegistrar()
      throws IOException, ClassNotFoundException,
         MalformedURLException {
      final String jurl =
         System.getProperty("jini.lookup.url","jini://localhost");
      final LookupLocator lookup = new LookupLocator(jurl);
      final ServiceRegistrar registrar = lookup.getRegistrar();
      if (registrar instanceof Administrable)
          debug("Registry is administrable.");
```

EXAMPLE 10–7 Registering the Connector Server with the Jini Lookup Service *(Continued)*

```
      return registrar;
   }

   public static ServiceRegistration register(ServiceRegistrar registrar,
                                              JMXConnector proxy,
                                              String name)
      throws IOException {
      Entry[] serviceAttrs = new Entry[] {
              new net.jini.lookup.entry.Name(name)
                      };

      System.out.println("Registering proxy: AgentName=" + name );
      debug("" + proxy);
      ServiceItem srvcItem = new ServiceItem(null, proxy, serviceAttrs);
      ServiceRegistration srvcRegistration =
             registrar.register(srvcItem, Lease.ANY);
      debug("Registered ServiceID: " +
                             srvcRegistration.getServiceID().toString());
      return srvcRegistration;
   }
```

Example 10–7 shows the creation of a connector server named `server` with the environment map `env` and the service URL `jurl`. The connector server instance `server` then gets a pointer to the Jini lookup service by calling the Jini lookup service method `LookupLocator.getRegistrar()`.

The connector server is registered in the Jini lookup service in the form of a proxy, using the Jini lookup service locator registrar and the agent name under which the connector server will be registered. The proxy is in fact a client stub for the connector server, obtained by a call to the `toJMXConnector()` method of `JMXConnectorServer`.

The registration itself is performed by a call to the `register()` method of the Jini lookup service class `ServiceRegistrar`, with an array of service items.

10.3.2 Looking up the Connector Server with the Jini Lookup Service

The following code extract is taken from the `Client` class in the Jini lookup service examples directory.

EXAMPLE 10–8 Looking up the Connector Server with the Jini Lookup Service

```
public class Client {

   private static boolean debug = false;
   public static ServiceRegistrar getRegistrar()
```

EXAMPLE 10–8 Looking up the Connector Server with the Jini Lookup Service *(Continued)*

```
       throws IOException, ClassNotFoundException, MalformedURLException {
       final String jurl =
                  System.getProperty("jini.lookup.url","jini://localhost");
       final LookupLocator lookup = new LookupLocator(jurl);
       final ServiceRegistrar registrar = lookup.getRegistrar();
       if (registrar instanceof Administrable)
               debug("Registry is administrable.");
       return registrar;
 }

   public static List lookup(ServiceRegistrar registrar,
           String name) throws IOException {
       final ArrayList list = new ArrayList();
       final Class[] classes = new Class[] {JMXConnector.class};
       final Entry[] serviceAttrs = new Entry[] {
           new net.jini.lookup.entry.Name(name)
   };

   ServiceTemplate template =
        new ServiceTemplate(null,classes,serviceAttrs);
   ServiceMatches matches =
        registrar.lookup(template, Integer.MAX_VALUE);
   for(int i = 0; i < matches.totalMatches; i++) {
        debug("Found Service: " + matches.items[i].serviceID);
        if (debug) {
           if (matches.items[i].attributeSets != null) {
                   final Entry[] attrs = matches.items[i].attributeSets;
                   for (int j = 0; j < attrs.length ; j++) {
                       debug("Attribute["+j+"]=" + attrs[j]);
               }
           }
        }


        if(matches.items[i].service != null) {
            JMXConnector c = (JMXConnector)(matches.items[i].service);
            debug("Found a JMXConnector: " + c);
            list.add(c);
        }
   }
   return list;
}

[...]
```

Example 10–8 shows how the connector client obtains a pointer to the Jini lookup service with a call to `lookup.getRegistrar()`. The client then obtains the list of the connectors registered as entries in the Jini lookup service with the agent name name. Unlike in the SLP example, the agent name you pass to `Client` when it is launched must be either an exact match of an existing agent name, or null, in which case the Jini lookup service will look up all the agents.

Once the list of connectors has been obtained, in code that is not shown here, the client connects to the MBean server started by Server, and retrieves the list of all the MBeans registered in it.

10.3.3 Running the Jini Lookup Service Example

In addition to the actions you performed in "10.1 Initial Configuration" on page 139, before you can run the lookup service examples that use the Jini lookup service, you must perform some further initial actions that are specific to this example. You can then start looking up connectors using the Jini network technology, in conjunction with the two connectors supported by Java DMK.

When you run the examples, to help you keep track of which agent has been created with which connector and transport, the agent names include a letter suffix. For example, the agent from the example of an RMI connector over JRMP, without an external directory is called `test-server-a`.

▼ To Set up the Jini Lookup Service Example

The following steps are required by all of the different connector/transport combinations you can run in this example.

1. **For your convenience when compiling and running the example classes, define some additional environment variables.**

   In addition to the common environment variables that you set in "10.1 Initial Configuration" on page 139 you can add the path to the Jini lookup service. The directory where you have installed the Jini networking technology is referred to as *jini_dir*.

   ```
   $ JINI_HOME=jini_dir
   $ JINI_LIB=$JINI_HOME/lib
   ```

2. **Define the `classp` environment variable.**

   In addition to the JAR files for the Java DMK runtime, the JMX specification and the JMX Remote API, this example requires the JAR files for the Jini lookup services core and extensions.

   ```
   $ classp=$JINI_LIB/jini-core.jar:$JINI_LIB/jini-ext.jar:classpath
   ```

3. **Create a`java.policy` file.**

   The`java.policy` file is a Java technology security policy file. A template `java.policy` file for is provided in the same directory as the classes for this example. You must complete the file to include all the necessary paths for your system. You must also rename the file from `java.policy.template` to `java.policy`.

4. **Create a `jini.properties` file.**

A properties file for UNIX platforms is provided in the same directory as the classes for this example. If you are not running a UNIX platform, you can obtain a properties file for your platform in the following directory:

jini_dir/example/launcher/jini12_*platform*.properties

5. **Update the jini.properties file.**

   You must complete the file to include all the necessary paths, host names and port numbers for your system. Even if you are not running a UNIX platform, you can use the template provided as a guide.

6. **Start the Jini networking technology StartService.**

   ```
   $ java -cp $JINI_LIB/jini-examples.jar
    com.sun.jini.example.launcher.StartService &
   ```

   This will open the StartService graphical user interface.

7. **Load your jini.properties file into StartService.**

   Click on File, Open Property File and then select your properties file from within *examplesDir*/current/Lookup/jini.

8. **Start the Jini lookup services.**

   Start the required Jini lookup services by clicking on the Run tab and then pressing the START button for each of the following:

   - RMID
   - WebServer
   - Reggie
   - LookupBrowser

9. **Compile the Client and Server classes.**

   ```
   $ javac -d . -classpath $classp Server.java Client.java
   ```

▼ To Run the Jini Lookup Service Example With an RMI Connector

This example demonstrates the use of the Jini lookup service to look up RMI connector servers that use RMI's default transport, JRMP, as well as the IIOP transport. In addition, as described in "10.1 Initial Configuration" on page 139, different external directories are used to register the RMI connector stubs.

The combinations of transports and external directories demonstrated here are:

- RMI connector over the JRMP transport, with:
  - No external directory
  - An RMI registry
  - An LDAP registry
- RMI connector over the IIOP transport, with:

- No external directory
- A CORBA naming service
- An LDAP registry

1. **Start the `Server`.**

   The command you use to start the `Server` varies according to which external directory you are using. You can start one or more of the following instances of `Server` with different transports and external registries before starting the `Client`.

   - RMI connector over JRMP, without an external directory:

     ```
     $ java -classpath .:$classp -Ddebug=true \
          -Dagent.name=test-server-a \
          -Durl="service:jmx:rmi://" \
          -Djava.security.policy=java.policy \
          jini.Server &
     ```

     In this command:

     - `debug` is set to true to provide more complete screen output when the Server runs
     - The name of the agent to be created is `test-server-a`
     - The service URL specifies that the chosen connector is an RMI connector, running over the RMI default transport JRMP.

     When `Server` is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the Jini lookup service.

   - RMI connector over JRMP, using an RMI registry as an external directory:

     ```
     $ java -classpath .:$classp -Ddebug=true \
          -Dagent.name=test-server-b \
          -Durl="service:jmx:rmi:///jndi/${jndirmi}/server" \
          -Djava.security.policy=java.policy \
          jini.Server &
     ```

     In this command:

     - The name of the agent created is `test-server-b`
     - The service URL specifies the chosen connector as RMI over JRMP, and the external directory in which the RMI connector stub, `server`, is stored is the RMI registry you identified as `jndirmi` in "10.1 Initial Configuration" on page 139

     When Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the Jini lookup service.

   - RMI connector over JRMP, using LDAP as the external directory:

     ```
     $ java -classpath .:$classp -Ddebug=true \
          -Dagent.name=test-server-c \
          -Durl="service:jmx:rmi:///jndi/${jndildap}/cn=x,dc=Test" \
     ```

```
     -Djava.security.policy=java.policy \
     -Djava.naming.security.principal="$principal" \
     -Djava.naming.security.credentials="$credentials" \
     jini.Server &
```

In this command:

- The name of the agent created is test-server-c
- The service URL specifies the chosen connector as RMI over JRMP, and the external directory in which the RMI connector stub is stored is the LDAP server you identified as jndildap in "10.1 Initial Configuration" on page 139
- The stub is registered in the Test domain component in the LDAP server.
- The common name attribute principal and password credentials are given to gain access to the LDAP server.

When Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the Jini lookup service under the agent name test-server-c.

- RMI connector over IIOP, without an external directory:

```
$ java -classpath .:$classp -Ddebug=true \
     -Dagent.name=test-server-d \
     -Durl="service:jmx:iiop://" \
     -Djava.security.policy=java.policy \
     jini.Server &
```

In this command:

- The name of the agent created is test-server-d
- The service URL specifies the chosen connector as RMI connector over IIOP.

When the Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its automatically generated URL in the Jini lookup service.

- RMI connector over IIOP, using CORBA naming as the external directory.

```
$ java -classpath .:$classp -Ddebug=true \
     -Dagent.name=test-server-e \
     -Durl="service:jmx:iiop:///jndi/${jndiiiop}/server" \
     -Djava.security.policy=java.policy \
     jini.Server &
```

In this command:

- The name of the agent created is test-server-e
- The service URL specifies the chosen connector as RMI connector over IIOP. The external directory in which the RMI connector stub server is stored is the CORBA naming service you identified as jndiiiop in "10.1 Initial Configuration" on page 139.

- RMI connector over IIOP, using LDAP as the external directory.

  ```
  $ java -classpath .:$classp -Ddebug=true \
       -Dagent.name=test-server-f \
       -Durl="service:jmx:iiop:///jndi/${jndildap}/cn=x,dc=Test" \
       -Djava.security.policy=java.policy \
       -Djava.naming.security.principal="$principal" \
       -Djava.naming.security.credentials="$credentials" \
       jini.Server &
  ```

  In this command:

  - The name of the agent created is `test-server-f`
  - The service URL specifies the chosen connector as RMI over IIOP, and the external directory in which the RMI connector stub is stored is the LDAP server you identified as `jndildap` in “10.1 Initial Configuration” on page 139.
  - The stub is registered in the `Test` domain component in the LDAP server.
  - The common name attribute `principal` and password `credentials` are given to gain access to the LDAP server.

  When `Server` is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the Jini lookup service under the agent name `test-server-f`.

2. **Start the `Client`.**

   After starting the Server using the transport and external directory of your choice, start the Client:

   ```
   $ java -classpath .:$classp -Ddebug=true \
        -Djava.security.policy=java.policy \
        jini.Client
   ```

   You will see output confirming the detection of the agents created by the `Server` and registered in the lookup service. You will also see the identification and confirmation of the connection made to the agents.

   To look up a specific agent, type the following command:

   ```
   $ java -classpath .:$classp -Ddebug=true \
        -Djava.security.policy=java.policy \
        -Dagent.name="agentName" \
        jini.Client
   ```

   In the command shown above, *agentName* is the name of the agent you want to look up. You can also specify a partial agent name by using `*`; for example, `x*` for all agent names beginning with the letter x.

▼ To Run the Jini Lookup Service Example With a JMXMP Connector

This example demonstrates the use of the Jini lookup service to look up JMXMP connector servers.

1. **Start the Server.**

   ```
   $ java -classpath .:$classp -Ddebug=true \
        -Dagent.name=test-server-g \
        -Durl="service:jmx:jmxmp://" \
        -Djava.security.policy=java.policy \
        jini.Server &
   ```

   In this command:

   - The name of the agent created is `test-server-g`
   - The service URL specifies the chosen connector as the JMXMP connector, running on the first available port.

   When the `Server` is launched, you will see confirmation of the creation of the JMXMP connector, and the registration of its automatically generated URL in the Jini lookup service. JMXMP connector servers can be used alongside RMI connectors, and will be detected by the `Client` in exactly the same way as RMI connector servers.

2. **Start the Client.**

   After starting the Server, start the Client:

   ```
   $ java -classpath .:$classp -Ddebug=true \
        -Djava.security.policy=java.policy \
        jini.Client
   ```

   You will see output confirming the detection of the agents created by the `Server` and registered in the lookup service. You will also see the identification and confirmation of the connection made to the agents.

   To look up a specific agent, type the following command:

   ```
   $ java -classpath .:$classp -Ddebug=true \
     -Djava.security.policy=java.policy \
     -Dagent.name="agentName" \
     jini.Client
   ```

   In the command shown above, *agentName* is the name of the agent you want to look up. You can also specify a partial agent name by using `*`; for example, `x*` for all agent names beginning with the letter x.

10.4 Java Naming and Directory Interface (JNDI) / LDAP Lookup Service

You can register RMI connectors or JMXMP connectors with a JNDI lookup service using an LDAP registry as a backend. This example performs the following operations:

- The agent:
  - Creates an MBean server
  - Creates a connector server
  - Registers the connector address with the LDAP server
- The client:
  - Gets a pointer to the JNDI/LDAP lookup service
  - Looks for any connector servers registered in the JNDI/LDAP lookup service
  - Creates a connector
  - Retrieves information about the MBeans in the MBean server

The JNDI/LDAP lookup example is contained in the directory *examplesDir*/`current/Lookup/ldap`.

Note – Some code provided in this example is specific to Sun's implementation of the Java 2 Platform, Standard Edition (J2SE) SDK 1.4. If you are not using Sun's implementation, you might need to adapt the code to your implementation.

10.4.1 Registering the Connector Server with the JNDI/LDAP Lookup Service

The following code extracts are taken from the `Server` class in the Jini Lookup Service examples directory.

EXAMPLE 10–9 Creating Connector Servers for Registration in the JNDI/LDAP Lookup Service

```
public class Server {
   public final static int JMX_DEFAULT_LEASE = 60;
   private static boolean debug = false;
   private final MBeanServer mbs;
   public Server() {
      mbs = MBeanServerFactory.createMBeanServer();
   }
```

EXAMPLE 10–9 Creating Connector Servers for Registration in the JNDI/LDAP Lookup Service *(Continued)*

```
  public static DirContext getRootContext() throws NamingException {
      final Hashtable env = new Hashtable();

      final String factory =
        System.getProperty(Context.INITIAL_CONTEXT_FACTORY,
                           "com.sun.jndi.ldap.LdapCtxFactory");
      final String ldapServerUrl =
        System.getProperty(Context.PROVIDER_URL);
      final String ldapUser =
        System.getProperty(Context.SECURITY_PRINCIPAL,
                           "cn=Directory Manager");
      final String ldapPasswd =
        System.getProperty(Context.SECURITY_CREDENTIALS);
      debug(Context.PROVIDER_URL + "=" + ldapServerUrl);
      debug(Context.SECURITY_PRINCIPAL + "=" + ldapUser);
      if (debug) {
                  System.out.print(Context.SECURITY_CREDENTIALS + "=");
                  final int len = (ldapPasswd==null)?0:ldapPasswd.length();
                  for (int i=0;i
                  for (int i=0;i<len;i++) System.out.print("*");
                  System.out.println();
      }
      env.put(Context.INITIAL_CONTEXT_FACTORY,factory);
      env.put(Context.SECURITY_PRINCIPAL, ldapUser);
      if (ldapServerUrl != null)
           env.put(Context.PROVIDER_URL, ldapServerUrl);
      if (ldapPasswd != null)
           env.put(Context.SECURITY_CREDENTIALS, ldapPasswd);
      InitialContext root = new InitialLdapContext(env,null);
      return (DirContext)(root.lookup(""));
  }

[...]
```

Example 10–9 shows the initial creation of an MBean server `mbs`, and the obtainment of a pointer to the root context of the LDAP directory tree in which the connector server address is to be registered. All the relevant LDAP access variables, such as the provider URL, the LDAP user name and the security credentials, are given here and passed into the environment map `env`. The environment map `env` is then passed as a parameter into a call to `InitialLdapContext`, from which the initial LDAP context is obtained.

Code that is not shown retrieves the agent name under which the connector will be registered in the LDAP server.

EXAMPLE 10–10 Registering the Connector Server Address in the LDAP Registry

```
[...]

public static void register(DirContext root,
```

EXAMPLE 10–10 Registering the Connector Server Address in the LDAP Registry *(Continued)*

```
                        JMXServiceURL jmxUrl,
                        String name)
   throws NamingException, IOException {

   final String mydn = System.getProperty("dn","cn="+name);

   debug("dn: " + mydn );

   Object o = null;
   try {
       o = root.lookup(mydn);
   } catch (NameNotFoundException n) {
       Attributes attrs = new BasicAttributes();
       Attribute objclass = new BasicAttribute("objectClass");
       objclass.add("top");
       objclass.add("javaContainer");
       objclass.add("jmxConnector");
       attrs.put(objclass);
       attrs.put("jmxAgentName", name);
       o = root.createSubcontext(mydn,attrs);
   }
   if (o == null) throw new NameNotFoundException();
   final Attributes attrs = root.getAttributes(mydn);
   final Attribute oc = attrs.get("objectClass");
   if (!oc.contains("jmxConnector")) {
       final String msg = "The supplied node [" + mydn +
         "] does not contain the jmxConnector objectclass";
       throw new NamingException(msg);
   }

   final Attributes newattrs = new BasicAttributes();
   newattrs.put("jmxAgentName",name);
   newattrs.put("jmxServiceURL",jmxUrl.toString());
   newattrs.put("jmxAgentHost",InetAddress.getLocalHost().getHostName());
   newattrs.put("jmxProtocolType",jmxUrl.getProtocol());
   newattrs.put("jmxExpirationDate",
                getExpirationDate(JMX_DEFAULT_LEASE));
   root.modifyAttributes(mydn,DirContext.REPLACE_ATTRIBUTE,newattrs);
}

[...]
```

Example 10–10 shows the registration of the JMX connector server service URL in the LDAP directory. The domain name in which the URL is registered can be passed on the command line through the `dn` System property, namely, `-Ddn=mydn`. If the `dn` system property is not specified, then you can use `cn=name`, in which `name` is the agent name. This is not mandatory, however.

The location in which the URL is registered is not important, because the client code never uses that DN directly, but instead performs an LDAP search to find the nodes that have an auxiliary JMX connector ObjectClass. What is important however, is that each URL is registered in its own LDAP node. How to name these nodes is left to the LDAP administrator. In this example, it is assumed that you have configured your LDAP server by creating a root context under which the node cn=name can be created, and that this root context has been passed to the LDAP initial context through the Context.PROVIDER_URL property (see Example 10–9).

The code shown above checks whether the node in which you will register the server URL already exists. If it does not, you try to create it (this will fail if the parent node does not exist). Since the ObjectClass is a simple auxiliary class, you can use the javaContainer ObjectClass as structural class if you need to create a new context. Once again, this is optional.

Any structural class to which the jmxConnector auxiliary class can be added is acceptable. It then checks whether the node in which you will register the server already has the jmxConnector auxiliary class. Otherwise, an exception is thrown.

At this point you are sure that the node in which you will register the URL exists, and has the appropriate jmxConnector auxiliary class. So you need only to replace the values of the attributes defined by the LDAP schema for JMX, (see jmx-schema.txt):

- jmxServiceUrl: contains the String form of the server URL, as obtained from server.getAddress() after the server was started.
- jmxAgentName: contains the JMX agent name.
- jmxProtocolType: contains the JMX protocol type, as returned by jmxUrl.getProtocolType().
- jmxAgentHost: contains the name of the agent host.
- jmxExpirationDate: contains the date at which the URL will be considered obsolete.

EXAMPLE 10–11 Registering the Connector Servers in the LDAP Server

```
[...]

   public JMXConnectorServer rmi(String url)
     throws IOException, JMException,
        NamingException, ClassNotFoundException {

     JMXServiceURL jurl = new JMXServiceURL(url);
     final HashMap env = new HashMap();
     // Prepare the environment Map

[...]

     JMXConnectorServer rmis =
        JMXConnectorServerFactory.newJMXConnectorServer(jurl, env, mbs);
```

EXAMPLE 10–11 Registering the Connector Servers in the LDAP Server *(Continued)*

```
     final String agentName = System.getProperty("agent.name",
                                             "DefaultAgent");
     start(rmis,env,agentName);
     return rmis;
   }

[...]
```

In Example 10–11, a new RMI connector server named `rmis` is created with the JMX service URL `jurl` and the appropriate LDAP properties passed to its environment map `env`. The connector server `rmis` is launched by calling `JMXConnectorServer.start()` and is registered in the LDAP server.

Subsequent code not shown here creates and registers a corresponding JMXMP connector server named `jmxmp`.

EXAMPLE 10–12 Creating the JMX Connector Server

```
[...]

    public void start(JMXConnectorServer server, Map env, String agentName)
       throws IOException, NamingException {
       server.start();
       final DirContext root=getRootContext();
       final JMXServiceURL address = server.getAddress();
       register(root,address,agentName);
    }

[...]
```

Example 10–12 shows the creation of a JMX connector server `server`, the obtainment of a pointer to the LDAP server root directory `root` and the creation of a URL for `server` named `address`. The root directory, the URL and an agent name are passed as parameters to `register()` and are registered in the LDAP server.

10.4.2 Looking up the Connector Servers with the JNDI/LDAP Lookup Service

The following code extract is taken from the `Client` class in the Jini Lookup Service examples directory.

EXAMPLE 10–13 Looking up the Connector Servers with the JNDI/LDAP Lookup Service

```
[...]

public class Client {
```

EXAMPLE 10–13 Looking up the Connector Servers with the JNDI/LDAP Lookup Service *(Continued)*

```
private static boolean debug = false;

public static void listAttributes(DirContext root, String dn)
  throws NamingException {
  final Attributes attrs = root.getAttributes(dn);
  System.out.println("dn: " + dn);
  System.out.println("attributes: " + attrs);
}

public static DirContext getRootContext() throws NamingException {
   final Hashtable env = new Hashtable();
   // Prepare environment map
   [...]

   InitialContext root = new InitialLdapContext(env,null);
   return (DirContext)(root.lookup(""));
}

// Confirm URL has not expired
[...]

public static List lookup(DirContext root, String protocolType,
                          String name)
   throws IOException, NamingException {
   final ArrayList list = new ArrayList();
   String queryProtocol =
      (protocolType==null)?"":"(jmxProtocolType="+protocolType+")";
   String query =
       "(&" + "(objectClass=jmxConnector) " +
       "(jmxServiceURL=*) " +
       queryProtocol +
       "(jmxAgentName=" + ((name!=null)?name:"*") + "))";

   SearchControls ctrls = new SearchControls();
   ctrls.setSearchScope(SearchControls.SUBTREE_SCOPE);
   final NamingEnumeration results = root.search("", query, ctrls);
   while (results.hasMore()) {
        final SearchResult r = (SearchResult) results.nextElement();
        debug("Found node: " + r.getName());
        final Attributes attrs = r.getAttributes();
        final Attribute attr = attrs.get("jmxServiceURL");
        if (attr == null) continue;
        final Attribute exp = attrs.get("jmxExpirationDate");
        if ((exp != null) && hasExpired((String)exp.get())) {
            System.out.print(r.getName() + ": ");
            System.out.println("URL expired since: " + exp.get());
            continue;
        }
        final String urlStr = (String)attr.get();
        if (urlStr.length() == 0) continue;
```

EXAMPLE 10–13 Looking up the Connector Servers with the JNDI/LDAP Lookup Service *(Continued)*

```
                debug("Found URL: "+ urlStr);


                final JMXServiceURL url = new JMXServiceURL(urlStr);
                final JMXConnector conn =
                    JMXConnectorFactory.newJMXConnector(url,null);
                list.add(conn);
                if (debug) listAttributes(root,r.getName());
            }

            return list;

    }
}
```

In Example 10–13, the `Client` firstly returns a pointer root to the LDAP directory `DirContext`, then it searches through the directory for object classes of the type `jmxConnector`. The service URL and expiry date attributes, `attr` and `exp` respectively, for the `jmxConnector` object classes are obtained, `exp` is checked to make sure that the URL has not expired and a call is made to `JMXConnectorFactory` to create a new connector `conn`. The connector `conn` is added to the list of connectors and is used to access the MBeans in the MBean server created by `Server`.

10.4.3 Running the JNDI/LDAP Lookup Service Example

In addition to the actions you performed in "10.1 Initial Configuration" on page 139, before you can run the lookup service examples that use the JNDI/LDAP lookup service, you must perform some further initial actions that are specific to this example. You can then start looking up connectors using the JNDI/LDAP network technology.

When you run the examples, to help you keep track of which agent has been created with which connector and transport, the agent names include a letter suffix. For example, the agent from the example of an RMI connector over JRMP, without an external directory is called `test-server-a`.

▼ To Set up the JNDI/LDAP Lookup Service Example

The following steps are required by all of the different connector/transport combinations you can run in this example.

1. **Stop the LDAP server you started in "10.1 Initial Configuration" on page 139.**

Do this according to the type of LDAP server you are using.

2. **Copy the JMX schema into your LDAP server's schema directory.**

   For example, if you are using Sun ONE Directory Server 5.0, you would type:

   ```
   $ cp 60jmx-schema.ldif /var/ds5/slapd-hostname/config/schema
   ```

   Otherwise, do this according to the type of LDAP server you are using.

3. **Restart the LDAP server.**

   Do this according to the type of LDAP server you are using.

4. **Define the root under which the `Server` will register its service URL.**

   You must provide the `Server` with the path to the domain component suffix `dc=Test` that you created in "10.1 Initial Configuration" on page 139.

   ```
   $ provider="ldap://$ldaphost:$ldapport/dc=Test"
   ```

5. **Compile the `Client` and `Server` classes.**

   ```
   $ javac -d . -classpath classpath Server.java Client.java
   ```

▼ To Run the JNDI/LDAP Lookup Service Example with an RMI Connector

This example demonstrates the use of the JNDI/LDAP lookup service to look up RMI connector servers that use RMI's default transport, JRMP, as well as the IIOP transport. In addition, as described in "10.1 Initial Configuration" on page 139, different external directories are used to register the RMI connector stubs.

The combinations of transports and external directories demonstrated here are:

- RMI connector over the JRMP transport, with:
  - No external directory
  - An RMI registry
  - An LDAP registry
- RMI connector over the IIOP transport, with:
  - No external directory
  - A CORBA naming service
  - An LDAP registry

Perform the following steps to run the example:

1. **Start the `Server`.**

   The command you use to start the `Server` varies according to which external directory you are using. You can start one or more of the following instances of `Server` with different transports and external registries before starting the `Client`.

- RMI connector over JRMP, without an external directory:

  ```
  $ java -classpath classpath -Ddebug=true \
       -Dagent.name=test-server-a \
       -Durl="service:jmx:rmi://" \
       -Djava.naming.provider.url="$provider" \
       -Djava.naming.security.principal="$principal" \
       -Djava.naming.security.credentials="$credentials" \
       jndi.Server &
  ```

  In this command:

  - `debug` is set to true to provide more complete screen output when the Server runs
  - The name of the agent to be created is `test-server-a`
  - The service URL specifies that the chosen connector is an RMI connector, running over the RMI default transport JRMP.

  When `Server` is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the JNDI/LDAP lookup service.

- RMI connector over JRMP, using an RMI registry as an external directory:

  ```
  $ java -classpath classpath -Ddebug=true \
       -Dagent.name=test-server-b \
       -Durl="service:jmx:rmi:///jndi/${jndirmi}/server" \
       -Djava.naming.provider.url="$provider" \
       -Djava.naming.security.principal="$principal" \
       -Djava.naming.security.credentials="$credentials" \
       jndi.Server &
  ```

  In this command:

  - The name of the agent created is `test-server-b`
  - The service URL specifies the chosen connector as RMI over JRMP, and the external directory in which the RMI connector stub, `server`, is stored is the RMI registry you identified as `jndirmi` in "10.1 Initial Configuration" on page 139

  When Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the JNDI/LDAP lookup service.

- RMI connector over JRMP, using LDAP as the external directory:

  ```
  $ java -classpath classpath -Ddebug=true \
       -Dagent.name=test-server-c \
       -Durl="service:jmx:rmi:///jndi/${jndildap}/cn=x,dc=Test" \
       -Djava.naming.provider.url="$provider" \
       -Djava.naming.security.principal="$principal" \
       -Djava.naming.security.credentials="$credentials" \
       jndi.Server &
  ```

  In this command:

    - The name of the agent created is test-server-c
    - The service URL specifies the chosen connector as RMI over JRMP, and the external directory in which the RMI connector stub is stored is the LDAP server you identified as jndildap in “10.1 Initial Configuration” on page 139
    - The stub is registered in the Test domain component in the LDAP server.
    - The common name attribute principal and password credentials are given to gain access to the LDAP server.

    When Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the JNDI/LDAP lookup service under the agent name test-server-c.

- RMI connector over IIOP, without an external directory:

  ```
  $ java -classpath classpath -Ddebug=true \
       -Dagent.name=test-server-d \
       -Durl="service:jmx:iiop://" \
       -Djava.naming.provider.url="$provider" \
       -Djava.naming.security.principal="$principal" \
       -Djava.naming.security.credentials="$credentials" \
       jndi.Server &
  ```

  In this command:

    - The name of the agent created is test-server-d
    - The service URL specifies the chosen connector as RMI connector over IIOP.

  When the Server is launched, you will see confirmation of the creation of the RMI connector, and the registration of its automatically generated URL in the JNDI/LDAP lookup service.

- RMI connector over IIOP, using CORBA naming as the external directory.

  ```
  $ java -classpath classpath -Ddebug=true \
       -Dagent.name=test-server-e \
       -Durl="service:jmx:iiop:///jndi/${jndiiiop}/server" \
       -Djava.naming.provider.url="$provider" \
       -Djava.naming.security.principal="$principal" \
       -Djava.naming.security.credentials="$credentials" \
       jndi.Server &
  ```

  In this command:

    - The name of the agent created is test-server-e
    - The service URL specifies the chosen connector as RMI connector over IIOP. The external directory in which the RMI connector stub server is stored is the CORBA naming service you identified as jndiiiop in “10.1 Initial Configuration” on page 139.

- RMI connector over IIOP, using LDAP as the external directory.

```
$ java -classpath classpath -Ddebug=true \
    -Dagent.name=test-server-f \
    -Durl="service:jmx:iiop:///jndi/${jndildap}/cn=x,dc=Test" \
    -Djava.naming.provider.url="$provider" \
    -Djava.naming.security.principal="$principal" \
    -Djava.naming.security.credentials="$credentials" \
    jndi.Server &
```

In this command:

- The name of the agent created is `test-server-f`
- The service URL specifies the chosen connector as RMI over IIOP, and the external directory in which the RMI connector stub is stored is the LDAP server you identified as `jndildap` in "10.1 Initial Configuration" on page 139.
- The stub is registered in the `Test` domain component in the LDAP server.
- The common name attribute `principal` and password `credentials` are given to gain access to the LDAP server.

When `Server` is launched, you will see confirmation of the creation of the RMI connector, and the registration of its URL in the JNDI/LDAP lookup service under the agent name `test-server-f`.

2. **Start the `Client`.**

After starting the `Server` using the transport and external directory of your choice, start the Client:

```
$ java -classpath classpath -Ddebug=true \
    -Djava.naming.provider.url="$provider" \
    -Djava.naming.security.principal="$principal" \
    -Djava.naming.security.credentials="$credentials" \
    jndi.Client
```

You will see output confirming the detection of the agents created by the `Server` and registered in the lookup service. You will also see the identification and confirmation of the connection made to the agents.

To look up a specific agent, type the following command:

```
$ java -classpath classpath -Ddebug=true \
    -Djava.naming.provider.url="$provider" \
    -Djava.naming.security.principal="$principal" \
    -Djava.naming.security.credentials="$credentials" \
    -Dagent.name=agentName \
    jndi.Client
```

In the command shown above, *agentName* is the name of the agent you want to look up. You can also specify a partial agent name by using `*`; for example, `x*` for all agent names beginning with the letter `x`.

▼ To Run the JNDI/LDAP Lookup Service Example with a JMXMP Connector

This example demonstrates the use of the JNDI/LDAP lookup service to look up JMXMP connector servers.

1. **Start the `Server`.**

   ```
   $ java -classpath classpath -Ddebug=true \
        -Dagent.name=test-server-g \
        -Durl="service:jmx:jmxmp://" \
        -Djava.naming.provider.url="$provider" \
        -Djava.naming.security.principal="$principal" \
        -Djava.naming.security.credentials="$credentials" \
        jndi.Server &
   ```

   In this command:

   - The name of the agent created is `test-server-g`
   - The service URL specifies the chosen connector as the JMXMP connector, running on the first available port.

   When the `Server` is launched, you will see confirmation of the creation of the JMXMP connector, and the registration of its automatically generated URL in the JNDI/LDAP lookup service. JMXMP connector servers can be used alongside RMI connectors, and will be detected by the `Client` in exactly the same way as RMI connector servers.

2. **Start the `Client`.**

   After starting the `Server`, start the `Client`:

   ```
   $ $ java -classpath classpath -Ddebug=true \
        -Djava.naming.provider.url="$provider" \
        -Djava.naming.security.principal="$principal" \
        -Djava.naming.security.credentials="$credentials" \
        jndi.Client
   ```

   You will see output confirming the detection of the agents created by the `Server` and registered in the JNDI/LDAP lookup service. You will also see the identification and confirmation of the connection made to the agents.

   To look up a specific agent, type the following command:

   ```
   $ $ java -classpath classpath -Ddebug=true \
        -Djava.naming.provider.url="$provider" \
        -Djava.naming.security.principal="$principal" \
        -Djava.naming.security.credentials="$credentials" \
        -Dagent.name="agentName" \
        jndi.Client
   ```

   In the command shown above, *agentName* is the name of the agent you want to look up. You can also specify a partial agent name by using `*`; for example, `x*` for all agent names beginning with the letter x.

10.5 Running the Lookup Examples Over Microsoft Active Directory

All the examples presented in this chapter can be run using the Microsoft Active Directory service. For details of how to configure Active Directory for use with these examples, consult the following file.

examplesDir`/current/Lookup/msad/README`

You will find the required utilities to configure Active Directory in same the same directory as the `README` file.

CHAPTER 11

Connector Security

Whenever considering a distributed architecture, security issues are often an added factor in the complexity of the design. This is not the case with the Java Dynamic Management Kit (Java DMK), whose security features are built into the modularity of the components.

Management solutions can evolve from basic password-based protection all the way to secure connections using cryptography simply by adding components to your connector protocols. The rest of the architecture is unchanged because it relies on the interface that is common to all connectors.

The code samples in this chapter are taken from the files in the `current/Security` example directory located in the main *examplesDir* (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "11.1 Simple Security" on page 178 presents examples of connectors that implement straightforward security based on password authentication and file access control.
- "11.2 Subject Delegation" on page 185 presents examples of connectors that use the subject delegation model to perform operations on a given authenticated connection on behalf of several different identities.
- "11.3 Fine-Grained Security" on page 189 presents examples of connectors that implement more sophisticated security mechanisms, in which permission to perform individual operations is controlled.
- "11.4 Advanced JMXMP Security Features" on page 194 shows how to implement a SASL mechanism, how to use SASL with privacy and how to configure the TLS socket factories.

Note – Most of the examples in this chapter implement a SASL mechanism. For more information about SASL, see the *The Java SASL API Programming and Deployment Guide* at the following URL.

http://java.sun.com/j2se/1.5.0/docs/guide/security/sasl/sasl-refguide.html

11.1 Simple Security

The simplest type of connector security you can use with the Java DMK is based upon encryption, user name and password authentication, and file access control. The exact implementation of these security features depends on whether you are using an RMI or a JMXMP connector, as explained in the following sections. The examples make use of the secure sockets layer (SSL) RMI socket factories and the SASL classes that are provided with Java DMK.

11.1.1 RMI Connectors With Simple Security

You can find an example of an RMI connector with simple security in the directory *examplesDir*/current/Security/rmi/simple. The properties files access.properties and password.properties are found in the config directory inside *examplesDir*/current/Security/rmi/simple.

The RMI connector server is created by the Server class, shown below.

EXAMPLE 11–1 Creating an RMI Connector Server with Simple Security

```
public class Server {

  public static void main(String[] args) {
  try {
       MBeanServer mbs = MBeanServerFactory.createMBeanServer();

       HashMap env = new HashMap();

       SslRMIClientSocketFactory  csf =
                  new SslRMIClientSocketFactory();
       SslRMIServerSocketFactory ssf =
                  new SslRMIServerSocketFactory();
       env.put(RMIConnectorServer.
                  RMI_CLIENT_SOCKET_FACTORY_ATTRIBUTE,csf);
       env.put(RMIConnectorServer.
```

EXAMPLE 11–1 Creating an RMI Connector Server with Simple Security *(Continued)*

```
                  RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE,ssf);

       env.put("jmx.remote.x.password.file",
                 "config" + File.separator + "password.properties");
       env.put("jmx.remote.x.access.file",
                 "config" + File.separator + "access.properties");

       JMXServiceURL url = new JMXServiceURL(
        "service:jmx:rmi:///jndi/rmi://localhost:9999/server");
         JMXConnectorServer cs =
            JMXConnectorServerFactory.newJMXConnectorServer(url,
                                                            env,
                                                            mbs);
       cs.start();
     } catch (Exception e) {
       e.printStackTrace();
     }
  }
}
```

The `Server` class shown in Example 11–1 creates an MBean server `mbs`, and populates an environment map `env` with a secure RMI client socket factory `csf`, a secure RMI server socket factory `ssf`, and the properties files `password.properties` and `access.properties`.

The properties file `password.properties` contains a username and password and is accessed using the `JMXAuthenticator` interface. Using the property `jmx.remote.x.password.file` is the same as creating a password-based `JMXAuthenticator` and passing it into the environment map through the `jmx.remote.authenticator` property.

The properties file `access.properties` contains a username and a level of access permission that can be either `readwrite` or `readonly`. This represents the level of access this user can have to MBean server operations. This file-based access control is implemented using the interface `MBeanServerForwarder`, which wraps the real MBean server inside an access controller MBean server. The access controller MBean server only forwards requests to the real MBean server after performing the appropriate checks.

Because the connector uses SSL, when you start the `Server` class, you will have to provide it with a pointer to the SSL `keystore` file and provide the password expected by the SSL RMI server socket factory `ssf`. This is shown in Example 11–2 below.

Server creates a service URL, named `url`, for an RMI connector that will operate over the default JRMP transport, and register an RMI connector stub in an RMI registry on port `9999` of the local host.

The MBean server `mbs`, the environment map `env` and the service URL `url` are all passed to `JMXConnectorServer` to create a new, secure connector server named `cs`.

EXAMPLE 11–2 Creating an RMI Connector Client with Simple Security

```
public class Client {

  public static void main(String[] args) {
  try {
      HashMap env = new HashMap();

      String[] credentials = new String[] { "username" , "password" };
      env.put("jmx.remote.credentials", credentials);
      JMXServiceURL url = new JMXServiceURL(
         "service:jmx:rmi:///jndi/rmi://localhost:9999/server");
      JMXConnector jmxc = JMXConnectorFactory.connect(url, env);
      MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
      String domains[] = mbsc.getDomains();
      for (int i = 0; i < domains.length; i++) {
         System.out.println("Domain[" + i + "] = " + domains[i]);
      }

      ObjectName mbeanName =
          new ObjectName("MBeans:type=SimpleStandard");
      mbsc.createMBean("SimpleStandard", mbeanName, null, null);
      // Perform MBean operations
      [...]

      mbsc.removeNotificationListener(mbeanName, listener);
      mbsc.unregisterMBean(mbeanName);
      jmxc.close();
    }  catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The Client class shown in Example 11–2 populates an environment map env with a set of credentials, namely the username and password expected by the Server. These credentials are then given to an instance of JMXConnector named jmxc when the service URL of the connector stub and the environment map are passed to JMXConnectorFactory.connect(). Through jmxc, the Client connects to the MBean server started by Server, and performs operations on the MBean SimpleStandard.

When the connection is established, the credentials supplied in the environment map env are sent to the server. The server then calls the authenticate() method of the JMXAuthenticator interface, passing the client credentials as parameters. The authenticate() method authenticates the client and returns a subject that contains the set of principals upon which the access control checks will be performed.

Because the connector uses SSL, when you start the Client class, you will have to provide it with a pointer to the SSL truststore file and provide the password, trustword, that is expected by the SSL RMI server socket factory ssf. This is shown in "To Run the RMI Connector Example With Simple Security" on page 181 below.

▼ To Run the RMI Connector Example With Simple Security

Run this example from within the *examplesDir*/current/Security/rmi/simple directory. Before running the example, make sure your *classpath* contains the runtime library for SSL RMI (see "Directories and Classpath" on page 26).

1. **Compile the example classes.**

   ```
   $ javac -classpath classpath \
         mbeans/SimpleStandard.java \
         mbeans/SimpleStandardMBean.java \
         server/Server.java \
         client/Client.java \
         client/ClientListener.java
   ```

2. **Start an RMI registry on port 9999 of the local host.**

   ```
   $ export CLASSPATH=server:classpath ; rmiregistry 9999 &
   ```

3. **Start the Server.**

   You have to provide the SSL keystore and its password when you start the Server

   ```
   $ java -classpath server:mbeans:classpath \
        -Djavax.net.ssl.keyStore=config/keystore \
        -Djavax.net.ssl.keyStorePassword=password \
        Server &
   ```

   You will see confirmation of the creation of the MBean server and of the RMI connector.

4. **Start the Client.**

   The Client requires the SSL truststore and its password when it is launched.

   ```
   $ java -classpath client:server:mbeans:classpath \
        -Djavax.net.ssl.trustStore=config/truststore \
        -Djavax.net.ssl.trustStorePassword=trustword \
        Client
   ```

   You will see confirmation of the creation of the connector client, the various MBean operations and finally the closure of the connection.

   As you can see, all the above appears to proceed in exactly the same manner as the basic RMI connector example shown in Chapter 9. However, if you were to open password.properties and change the password, you would see a java.lang.SecurityException when you launched the Client, and the connection would fail.

11.1.2 JMXMP Connectors With Simple Security

You can find an example of a JMXMP connector with simple security in the directory *examplesDir*`/current/Security/jmxmp/simple`. JMXMP connectors can be secured using Java DMK's own implementation of the Simple Authentication and Security Layer (SASL), which provides a greater level of security than the SSL security implemented by the RMI connector.

EXAMPLE 11–3 Creating a JMXMP Connector Server with Simple Security

```
public class Server {

    public static void main(String[] args) {
      try {
           MBeanServer mbs = MBeanServerFactory.createMBeanServer();
           HashMap env = new HashMap();

           Security.addProvider(
                 new com.sun.jdmk.security.sasl.Provider());
           env.put("jmx.remote.profiles", "TLS SASL/PLAIN");
           env.put("jmx.remote.sasl.callback.handler",
               new PropertiesFileCallbackHandler("config" +
                                               File.separator +
                                               "password.properties"));
           env.put("jmx.remote.x.access.file",
                 "config" + File.separator + "access.properties");

           JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);
           JMXConnectorServer cs =
              JMXConnectorServerFactory.newJMXConnectorServer(url,
                                                              env,
                                                              mbs);
           cs.start();

         } catch (Exception e) {
           e.printStackTrace();
         }
      }
  }
```

Example 11–3 shows the creation of an MBean server `mbs`, and an environment map `env`. A security provider is defined by a call to the `addProvider()` method of the `Security` class, implementing server support for the PLAIN SASL mechanism to Java DMK.

The environment map `env` is populated with the TLS and PLAIN SASL profiles. A callback handler, an instance of the `PropertiesFileCallbackHandler` class, is also passed to the environment map. The `PropertiesFileCallbackHandler` class is an implementation of the `CallbackHandler` interface, and is used by `Server` to check that the user name and password supplied by the client against the user name and password supplied in the `password.properties` file.

Finally, the access properties file, access.properties, is passed into the environment map to define the names of the users who will be authorized to access the connector server, and their level of access.

When the security provider and the environment map have been configured, a service URL, named url, is created. This URL is defined by the constructor JMXServiceURL, with its three arguments specifying JMXMP as the connector protocol, a null host name, and port number 5555 as the port upon which the connector server will listen for connections. The URL is then provided, along with the environment map env, to create an instance of JMXConnectorServer, named cs. The connector server cs is started by calling the start() method of JMXConnectorServer.

EXAMPLE 11–4 Creating a JMXMP Connector Client with Simple Security

```
public class Client {

   public static void main(String[] args) {
     try {
          HashMap env = new HashMap();
          Security.addProvider(new com.sun.security.sasl.Provider());
          env.put("jmx.remote.profiles", "TLS SASL/PLAIN");
          env.put("jmx.remote.sasl.callback.handler",
             new UserPasswordCallbackHandler("username", "password"));

          JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);
          JMXConnector jmxc = JMXConnectorFactory.connect(url, env);
          MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
          String domains[] = mbsc.getDomains();
          for (int i = 0; i < domains.length; i++) {
          System.out.println("Domain[" + i + "] = " + domains[i]);
          }

          ObjectName mbeanName =
              new ObjectName("MBeans:type=SimpleStandard");
          mbsc.createMBean("SimpleStandard", mbeanName, null, null);
          // Perform MBean operations
          //
          [...]
          mbsc.removeNotificationListener(mbeanName, listener);
          mbsc.unregisterMBean(mbeanName);
          jmxc.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
     }
 }
```

In Example 11–4, we see the creation of an environment map env, and a security provider is defined by a call to the addProvider() method of the Security class, implementing the client support for the PLAIN SASL mechanism.

A callback handler, an instance of the UserPasswordCallbackHandler class, is passed to the environment map for use by the PLAIN SASL client mechanism to retrieve the user name and password of the remote user connecting to the server.

Once the security provider and the environment map have been configured, a service URL named url is created. This URL defines JMXMP as the connector protocol, and port number 5555 as the port upon which the connector server will make connections. The URL is then provided, along with the environment map env, to create an instance of JMXConnector, named jmxc. An MBean server connection, mbsc, is made by calling the getMBeanServerConnection of the connector jmxc.

In code that is not shown here, when the secure connection to the MBean server has been established, the Client creates an MBean called SimpleStandard and performs various operations on it. Once these MBean operations have completed, the Client unregisters the MBean, and closes down the connection jmxc.

▼ To Run the JMXMP Connector Example with Simple Security

Run this example from within the *examplesDir*/current/Security/jmxmp/simple directory. Before running the example, make sure your *classpath* contains the runtime library for SASL (see "Directories and Classpath" on page 26).

1. **Compile the example classes.**

```
$ javac -classpath classpath \
      mbeans/SimpleStandard.java \
      mbeans/SimpleStandardMBean.java \
      server/Server.java \
      server/PropertiesFileCallbackHandler.java \
      client/Client.java \
      client/ClientListener.java \
      client/UserPasswordCallbackHandler.java
```

2. **Start the Server:**

   The Server requires the SSL keystore file and its password when you launch it.

```
$ java -classpath server:mbeans:classpath \
     -Djavax.net.ssl.keyStore=config/keystore \
     -Djavax.net.ssl.keyStorePassword=password \
     Server &
```

   You will see confirmation of the creation of the MBean server, the initialization of the environment map and the launching of the JMXMP connector.

3. **Start the Client:**

   The Client requires the SSL truststore and its password when it is launched.

```
$ java -classpath client:mbeans:classpath \
     -Djavax.net.ssl.trustStore=config/truststore \
     -Djavax.net.ssl.trustStorePassword=trustword \
```

Client

You will see confirmation of the creation of the JMXMP connector client, the initialization of the environment map, the connection to the MBean server and the performance of the various MBean operations followed by the closure of the connection.

As you can see, all the above appears to proceed in exactly the same manner as the basic JMXMP connector example shown in Chapter 9. However, if you were to open `access.properties` and change the access to `readonly` rather than `readwrite`, you would see a `java.lang.SecurityException` when you launched the `Client`, and the connection would fail.

11.2 Subject Delegation

If your implementation requires the client end of the connection to perform different operations on behalf of multiple users or applications, if you were to use the security mechanisms demonstrated in “11.1 Simple Security” on page 178, each different user would require one secure connection for every operation it performs. If you expect your connector clients to interact with numerous users, you can reduce the load on your system by implementing subject delegation. *Subject delegation* establishes a single secure connection for a user, and this connection can be used to perform related operations on behalf of any number of users. The connection itself is made by an *authenticated* user. If the authenticated user has been granted a `SubjectDelegationPermission` that allows it to act on behalf of another user, then operations can be performed over the connection on behalf of that user.

11.2.1 Secure RMI Connectors With Subject Delegation

You can find an example of a secure RMI connector that implements subject delegation in the directory *examplesDir*`/current/Security/rmi/subject_delegation`.

The `Server` class used in this example is identical to the one used in “11.1 Simple Security” on page 178. However, unlike the simple security example, this example implements a `java.policy` file to grant permission to certain users to perform certain actions.

EXAMPLE 11–5 A `java.policy` File

```
grant codeBase "file:installDir/lib/jmxremote.jar" {
    permission javax.management.remote.SubjectDelegationPermission
             "javax.management.remote.JMXPrincipal.delegate";
};
```

EXAMPLE 11–5 A `java.policy` File *(Continued)*

```
grant codeBase "file:server" {
    permission javax.management.remote.SubjectDelegationPermission
              "javax.management.remote.JMXPrincipal.delegate";
};

grant principal javax.management.remote.JMXPrincipal "username" {
     permission javax.management.remote.SubjectDelegationPermission
              "javax.management.remote.JMXPrincipal.delegate";
};
```

The `java.policy` file grants a user named `username` a `SubjectDelegationPermission` so it can perform operations on behalf of the user `delegate`, an instance of `JMXPrincipal` created by the `Client` class. The `java.policy` file is required when launching the `Server` class. The creation of the `JMXPrincipal` instance by the `Client` is shown below.

EXAMPLE 11–6 Creating a Delegation Subject

```
[...]

JMXConnector jmxc = JMXConnectorFactory.connect(url, env);
Subject delegationSubject =
            new Subject(true,
                Collections.singleton(new JMXPrincipal("delegate")),
                Collections.EMPTY_SET,
                Collections.EMPTY_SET);

MBeanServerConnection mbsc =
          jmxc.getMBeanServerConnection(delegationSubject);
[...]
```

The `Client` class is identical to the one used in the simple security example, except for the creation of the delegation subject shown in Example 11–6.

As before, the `Client` creates an environment map `env` that is populated with a user name `username` and a password `password`. These strings match the user name and password stored in the `password.properties` file that is held by the `Server` to authenticate users accessing the connector server.

A connector client `jmxc` is created in the same way as in the previous RMI connector examples, with the user name and password passed into the environment map `env`.

The `Client` then creates an instance of `Subject`, called `delegationSubject`, with a `Principal` that is an instance of `JMXPrincipal`, named `delegate`.

An MBean server connection, named mbsc, is created by calling the getMBeanServerConnection() method of JMXConnector, with delegationSubject passed in as a parameter. This MBean server connection therefore allows operations to be performed on the remote MBean server on behalf of the principals stored in the delegationSubject, which in this example is the JMXPrincipal named delegate.

The example continues by creating and registering the SimpleStandard MBean in the MBean server, and performing operations on it, in exactly the same way as in the previous examples.

▼ To Run the Secure RMI Connector Example With Subject Delegation

Run this example from within the *examplesDir*/current/Security/rmi/subject_delegation directory.

1. **Compile the example classes.**

   ```
   $ javac -classpath classpath \
     mbeans/SimpleStandard.java \
     mbeans/SimpleStandardMBean.java \
     server/Server.java \
     client/Client.java \
     client/ClientListener.java
   ```

2. **Start an RMI registry on port 9999 of the local host.**

   ```
   $ export CLASSPATH=server:classpath ; rmiregistry 9999 &
   ```

3. **Create a java.policy file from the java.policy.template file in the config directory.**

   You must replace @INSTALL_HOME_FOR_JDMK@ with your *installDir*.

4. **Start the Server.**

   As was the case with the simple secure RMI connector example, you have to provide the SSL keystore and its password when you start the Server, as well as the a pointer to the java.policy that grants permission to the delegate subject.

   ```
   $ java -classpath server:mbeans:classpath \
        -Djavax.net.ssl.keyStore=config/keystore \
        -Djavax.net.ssl.keyStorePassword=password \
        -Djava.security.policy=config/java.policy Server &
   ```

   You will see confirmation of the creation of the MBean server, the initialization of the environment map, the creation of the RMI connector, and the registration of the connector in the MBean server.

5. **Start the Client.**

Again, the Client requires the SSL truststore and its password when it is launched.

```
$ java -classpath client:server:mbeans:classpath \
     -Djavax.net.ssl.trustStore=config/truststore \
     -Djavax.net.ssl.trustStorePassword=trustword \
     Client
```

You will see confirmation of the creation of the connector client, the creation of the delegation subject, the connection to the MBean server and the various MBean operations followed by the closure of the connection.

11.2.2 Secure JMXMP Connectors With Subject Delegation

The example of JMXMP connectors with subject delegation is mostly identical to the example of a simple secure JMXMP connector. The only differences are in the client end of the connection, in which the delegation subject is defined, and in the java.policy file used to grant permission to the delegation subjects created.

- The Client class defines the same delegation subject as was defined for the RMI connector with subject delegation, in Example 11–6.
- The java.policy file in this example is identical to the one used for the RMI connector with subject delegation, except that it points to the jmxremote_optional.jar runtime library that is required by JMXMP connectors.

In this example, the Server class creates an MBean server, and a connector server cs, again protected by an SSL password, as was the case in the simple secure JMXMP connector example.

The Client creates a connector client named jmxc in the same way as in the previous JMXMP connector examples. The Client then creates an instance of Subject, called delegationSubject, with a Principal that is an instance of JMXPrincipal, named delegate.

An MBean server connection, named mbsc, is created by calling the getMBeanServerConnection() method of JMXConnector, with delegationSubject passed in as a parameter. This MBean server connection therefore allows operations to be performed on the remote MBean server on behalf of the principals stored in the delegationSubject, which in this example is the JMXPrincipal named delegate.

▼ To Run the Secure JMXMP Connector Example With Subject Delegation

Run this example from within the *examplesDir*/current/Security/jmxmp/subject_delegation directory.

1. **Compile the example classes.**

   ```
   $ javac -classpath classpath \
         mbeans/SimpleStandard.java \
         mbeans/SimpleStandardMBean.java \
         server/Server.java \
         server/PropertiesFileCallbackHandler.java \
         client/Client.java \
         client/ClientListener.java \
         client/UserPasswordCallbackHandler.java
   ```

2. **Create a `java.policy` file from the `java.policy.template` file in the `config` directory.**

   You must replace `@INSTALL_HOME_FOR_JDMK@` with your *installDir*.

3. **Start the Server.**

   The `Server` requires the SSL `keystore` file and its password, and a pointer to the `java.policy` file when you launch it.

   ```
   $ java -classpath server:mbeans:classpath \
        -Djavax.net.ssl.keyStore=config/keystore \
        -Djavax.net.ssl.keyStorePassword=password \
        -Djava.security.policy=config/java.policy Server &
   ```

   You will see confirmation of the creation of the MBean server, the initialization of the environment map and the launching of the JMXMP connector.

4. **Start the Client.**

   Again, the `Client` requires the SSL `truststore` and its password when it is launched.

   ```
   $ java -classpath client:mbeans:classpath \
        -Djavax.net.ssl.trustStore=config/truststore \
        -Djavax.net.ssl.trustStorePassword=trustword \
        Client
   ```

   You will see confirmation of the creation of the JMXMP connector client, the initialization of the environment map, the creation of the delegation subject, the connection to the MBean server and the performance of the various MBean operations followed by the closure of the connection.

11.3 Fine-Grained Security

You can implement a more fine-grained level of security in your connectors by managing user access through the Java Authentication and Authorization Service (JAAS) and Java 2 platform Standard Edition (J2SE) Security Architecture. JAAS and J2SE security is based on the use of security managers and policy files to allocate different levels of access to different users. Consequently, you can decide more precisely which users are allowed to perform which operations.

The two examples in this section are very similar to those shown in “11.1 Simple Security” on page 178, with the difference being that, in addition to SSL encryption, the simple, file-based access control has been replaced by policy-based access control.

11.3.1 RMI Connector With Fine-Grained Security

You can find an example of an RMI connector with fine-grained security in the directory *examplesDir*/`current/Security/rmi/fine_grained`.

The `Server` class used in this example is very similar to the one used in the RMI connector example with simple security. The only difference is that there is no `access.properties` file to map into the environment map in the fine-grained example. This was omitted so as not to make the example overly complicated. Otherwise, all the other classes and files used in this example are the same as those used in “11.1.1 RMI Connectors With Simple Security” on page 178, with the exception of the `java.policy` file, which is shown below.

EXAMPLE 11–7 A `java.policy` File for an RMI Connector With Fine-Grained Security

```
grant codeBase "file:installDir/lib/jmx.jar" {
    permission java.security.AllPermission;
};

grant codeBase "file:installDir/lib/jmxremote.jar" {
    permission java.security.AllPermission;
};

grant codeBase "file:server" {
    permission java.security.AllPermission;
};

grant codeBase "file:mbeans" {
    permission javax.management.MBeanTrustPermission "register";
};

grant principal javax.management.remote.JMXPrincipal "username" {
    permission javax.management.MBeanPermission "*", "getDomains";
    permission javax.management.MBeanPermission
             "SimpleStandard#-[-]",  "instantiate";
    permission javax.management.MBeanPermission
             "SimpleStandard#-[MBeans:type=SimpleStandard]",
             "registerMBean";
    permission javax.management.MBeanPermission
             "SimpleStandard#State[MBeans:type=SimpleStandard]",
             "getAttribute";
    permission javax.management.MBeanPermission
             "SimpleStandard#State[MBeans:type=SimpleStandard]",
             "setAttribute";
    permission javax.management.MBeanPermission
             "SimpleStandard#-[MBeans:type=SimpleStandard]",
             "addNotificationListener";
```

EXAMPLE 11–7 A `java.policy` File for an RMI Connector With Fine-Grained Security *(Continued)*

```
    permission javax.management.MBeanPermission
             "SimpleStandard#reset[MBeans:type=SimpleStandard]",
             "invoke";
    permission javax.management.MBeanPermission
              "SimpleStandard#-[MBeans:type=SimpleStandard]",
              "removeNotificationListener";
    permission javax.management.MBeanPermission
              "SimpleStandard#-[MBeans:type=SimpleStandard]",
              "unregisterMBean";
    permission javax.management.MBeanPermission
              "javax.management.MBeanServerDelegate#
              -[JMImplementation:type=MBeanServerDelegate]",
              "addNotificationListener";
    permission javax.management.MBeanPermission
              "javax.management.MBeanServerDelegate#
              -[JMImplementation:type=MBeanServerDelegate]",
              "removeNotificationListener";
};
```

The `java.policy` file shown in Example 11–7 grants the following permissions:

- All permissions to the `server` codebase, so that the connector server can create the connectors, and then perform the operations requested by remote user calls
- `MBeanTrustPermission` to the `mbeans` codebase, allowing trusted MBeans to register in the MBean server
- Permission to perform the various MBean and MBean server operations defined by the `SimpleStandard` MBean, for the user represented by a `JMXPrincipal` named `username`.

▼ To Run the RMI Connector Example With Fine-Grained Security

Run this example from within the *examplesDir*`/current/Security/rmi/fine_grained` directory.

1. **Compile the example classes.**

```
$ javac -classpath classpath \
      mbeans/SimpleStandard.java \
      mbeans/SimpleStandardMBean.java \
      server/Server.java \
      client/Client.java \
      client/ClientListener.java
```

2. **Start an RMI registry on port 9999 of the local host.**

```
$ export CLASSPATH=server:classpath ; rmiregistry 9999 &
```

3. **Create a `java.policy` file from the `java.policy.template` file in the `config` directory.**

   You must replace `@INSTALL_HOME_FOR_JDMK@` with your *installDir*.

4. **Start the `Server`.**

   You need to provide the `Server` with a pointer to the SSL `keystore`, the SSL password, the JAAS security manager and the `java.policy` file when you start the `Server` class.

   ```
   $ java -classpath server:mbeans:classpath \
        -Djavax.net.ssl.keyStore=config/keystore \
        -Djavax.net.ssl.keyStorePassword=password \
        -Djava.security.manager \
        -Djava.security.policy=config/java.policy Server &
   ```

   You will see confirmation of the initialization of the environment map, the creation of the MBean server and of the RMI connector.

5. **Start the `Client`.**

   Again, the `Client` requires the SSL `truststore` and its password when it is launched.

   ```
   $ java -classpath client:server:mbeans:classpath \
        -Djavax.net.ssl.trustStore=config/truststore \
        -Djavax.net.ssl.trustStorePassword=trustword \
        Client
   ```

   You will see confirmation of the creation of the connector client, the connection to the RMI server and the various MBean operations followed by the closure of the connection.

11.3.2 JMXMP Connectors With Fine-Grained Security

The example of JMXMP connectors with fine-grained security is mostly identical to the example of a simple secure JMXMP connector. The only difference is in the `java.policy` file used to grant permissions. The `java.policy` file is in turn mostly identical to the one used in “11.3.1 RMI Connector With Fine-Grained Security” on page 190, except for the addition of a codebase for SASL, as shown below.

EXAMPLE 11–8 A `java.policy` File for a JMXMP Connector With Fine-Grained Security

```
grant codeBase "file:installDir/lib/jmx.jar" {
    permission java.security.AllPermission;
};

grant codeBase "file:installDir/lib/jmxremote.jar" {
    permission java.security.AllPermission;
};
```

EXAMPLE 11–8 A `java.policy` File for a JMXMP Connector With Fine-Grained Security *(Continued)*

```
grant codeBase "file:installDir/lib/jmxremote_optional.jar" {
    permission java.security.AllPermission;
};

grant codeBase "file:installDir/lib/sasl.jar" {
    permission java.security.AllPermission;
};

grant codeBase "file:installDir/lib/sunsasl.jar" {
    permission java.security.AllPermission;
};

grant codeBase "file:installDir/lib/jdmkrt.jar" {
    permission java.security.AllPermission;
};

grant codeBase "file:server" {
    permission java.security.AllPermission;
};

[...]
```

This `java.policy` file grants the following permissions:

- All permissions to the runtime libraries for each of JMX, JMX Remote API, Java DMK and SASL.
- All permissions to the `server` codebase, so that the connector server can create the connectors, and then perform the operations requested by remote user calls.
- The remaining permissions, which are not shown in the example above, grant the same permissions to perform MBean operations as were defined in Example 11–7.

▼ To Run the JMXMP Connector Example With Fine-Grained Security

Run this example from within the *examplesDir*`/current/Security/jmxmp/fine_grained` directory.

1. **Compile the example classes.**

```
$ javac -classpath classpath \
      mbeans/SimpleStandard.java \
      mbeans/SimpleStandardMBean.java \
      server/Server.java \
      server/PropertiesFileCallbackHandler.java \
      client/Client.java \
      client/ClientListener.java \
      client/UserPasswordCallbackHandler.java
```

2. **Create a `java.policy` file from the `java.policy.template` file in the `config` directory.**

   You must replace `@INSTALL_HOME_FOR_JDMK@` with your *installDir*.

3. **Start the `Server`.**

   You need to provide the `Server` with a pointer to the SSL `keystore`, the SSL password, the JAAS security manager and the `java.policy` file when you start the `Server` class.

   ```
   $ java -classpath server:mbeans:classpath \
        -Djavax.net.ssl.keyStore=config/keystore \
        -Djavax.net.ssl.keyStorePassword=password \
        -Djava.security.manager \
        -Djava.security.policy=config/java.policy Server &
   ```

   You will see confirmation of the creation of the MBean server, the initialization of the environment map and the launching of the JMXMP connector and its registration in the MBean server.

4. **Start the `Client`.**

   Again, the `Client` requires the SSL `truststore` and its password when it is launched.

   ```
   $ java -classpath client:mbeans:classpath \
        -Djavax.net.ssl.trustStore=config/truststore \
        -Djavax.net.ssl.trustStorePassword=trustword \
        Client
   ```

   You will see confirmation of the creation of the JMXMP connector client, the initialization of the environment map, the connection to the MBean server and the performance of the various MBean operations followed by the closure of the connection.

11.4 Advanced JMXMP Security Features

In addition to the simple SASL authentication demonstrated in "11.1.2 JMXMP Connectors With Simple Security" on page 182 above, the JMXMP connector can also implement a more complete security solution based on the DIGEST-MD5 SASL mechanism. This allows you to make communication *private*. These examples also show how to customize your SASL implementation by adding your own version of the SASL provider. A third implementation allows you to supply a custom configuration of the TLS socket factory.

The security features described in this section all relate exclusively to the JMXMP connector.

11.4.1 SASL Privacy

As stated above, Java DMK enables you to make your JMXMP connections private by using a combination of the SASL/DIGEST-MD5 profile, for user authentication and encryption, and file access control based on the MBeanServerForwarder interface, for user access level authorization.

There is an example of an JMXMP connector implementing SASL privacy in the current/Security/jmxmp/sasl_privacy directory in the main *examplesDir*.

EXAMPLE 11–9 Implementing SASL Privacy in a JMXMP Connector Server

```
[...]
import javax.security.sasl.Sasl;

public class Server {

    public static void main(String[] args) {
        try {
            MBeanServer mbs = MBeanServerFactory.createMBeanServer();
            HashMap env = new HashMap();

            Security.addProvider(new com.sun.security.sasl.Provider());
            env.put("jmx.remote.profiles", "SASL/DIGEST-MD5");
            env.put(Sasl.QOP, "auth-conf");
            env.put("jmx.remote.sasl.callback.handler",
                     new DigestMD5ServerCallbackHandler("config" +
                                                        File.separator +
                                                        "password.properties",
                                                        null,
                                                        null));

            env.put("jmx.remote.x.access.file",
                        "config" + File.separator + "access.properties");

            JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);
            JMXConnectorServer cs =
                JMXConnectorServerFactory.newJMXConnectorServer(url,
                                                                env,
                                                                mbs);
            cs.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

This example is similar to the simple secure JMXMP example shown in "11.1.2 JMXMP Connectors With Simple Security" on page 182. It is structured in the same way, with an environment map env being populated with the required security configurations before being passed into a new instance of JMXConnectorServer.

However, as you can see, unlike in the simple secure JMXMP example, which used the PLAIN SASL mechanism, this example uses the DIGEST-MD5 SASL mechanism. The SASL DIGEST-MD5 profile defines the digital signature that must be exchanged between the client and server for a connection to be made, based on the HTTP Digest Authentication mechanism. The HTTP Digest Authentication mechanism was developed by the Internet Engineering Task Force as RFC 2831.

With the protocol profile set to DIGEST-MD5, the level of protection to be provided is defined by the quality of protection property, Sasl.QOP. In this example, the Sasl.QOP is set to auth-conf, denoting that the types of protection to be implemented are authentication and confidentiality, namely encryption. The encryption cypher used is dictated by SASL and can be controlled by properties specific to the DIGEST-MD5 SASL mechanism.

A instance of the callback handler provided with this example, DigestMD5ServerCallbackHandler, is created with the required password file, config/password.properities, passed to it. Similarly, the name of the user allowed to connect to the server, and the level of access granted to that user, are defined when the config/access.properties file is passed into the environment map. These properties must be matched by the client end of the connector, if the connection is to succeed. In this example, the level of access rights granted is readwrite, and the approved user is called username. The password.properties file provides the password for user username that is expected by the SASL/DIGEST-MD5 profile.

The DigestMD5ServerCallbackHandler class, being an implementation of the standard Java callback interface CallbackHandler, allows the server to retrieve the authentication information it needs from the properties files described above. This example does not implement canonical naming or proxy files, so the second and third parameters passed to DigestMD5ServerCallbackHandler when it is instantiated are null. The DIGEST-MD5 server mechanism will then be able to compare them with the user credentials supplied remotely by the client.

The environment map containing all of the above is then used to create the connector server instance, cs.

EXAMPLE 11–10 Implementing SASL Privacy in a JMXMP Connector Client

```
public class Client {

    public static void main(String[] args) {
        try {
            HshMap env = new HashMap();

            Security.addProvider(new com.sun.security.sasl.Provider());
            env.put("jmx.remote.profiles", "SASL/DIGEST-MD5");
            env.put(Sasl.QOP, "auth-conf");
            env.put("jmx.remote.sasl.callback.handler",
                    new DigestMD5ClientCallbackHandler("username",
                                                       "password"));
```

EXAMPLE 11–10 Implementing SASL Privacy in a JMXMP Connector Client *(Continued)*

```
            JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);
            JMXConnector jmxc = JMXConnectorFactory.connect(url, env);

            MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();

            String domains[] = mbsc.getDomains();
            for (int i = 0; i < domains.length; i++) {
                System.out.println("\tDomain[" + i + "] = " + domains[i]);
            }

            ObjectName mbeanName = new ObjectName(
                                        "MBeans:type=SimpleStandard");
            System.out.println("\nCreate SimpleStandard MBean...");
            mbsc.createMBean("SimpleStandard", mbeanName, null, null);

            // Perform MBean operations, before unregistering MBean
            //  & closing connection
            [...]
            mbsc.unregisterMBean(mbeanName);

            jmxc.close();
            System.out.println("\nBye! Bye!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

As shown in Example 11–10 above, the client end of the private connection is created using a very similar set of properties as the server end. However, the client end of the connector supplies the user and password information via a different callback handler, `DigestMD5ClientCallbackHandler`. Like `DigestMD5ServerCallbackHandler`, `DigestMD5ClientCallbackHandler` is an implementation of the `CallbackHandler` interface. `DigestMD5ClientCallbackHandler` is passed the username and password expected by `DigestMD5ServerCallbackHandler` when it is instantiated, which it then passes to the `DIGEST-MD5` client mechanism. The client `DIGEST-MD5` mechanism then makes them available for comparison with the properties held by the server, thus allowing the connection to the MBean server to be established. Operations can then be performed on the MBeans the MBean server contains.

▼ To Run the Secure JMXMP Connector Example with SASL Privacy

Run this example from within the *examplesDir*`/current/Security/jmxmp/sasl_privacy` directory.

1. **Compile the Java classes.**

```
$ javac -classpath classpath \
      mbeans/SimpleStandard.java \
      mbeans/SimpleStandardMBean.java \
      server/Server.java \
      server/DigestMD5ServerCallbackHandler.java \
      client/Client.java \
      client/ClientListener.java \
      client/DigestMD5ClientCallbackHandler.java
```

2. **Start the `Server`.**

   ```
   $ java -classpath server:mbeans:classpath Server &
   ```

   You will see confirmation of the creation of the MBean server, the initialization of the environment map and the launching of the JMXMP connector and its registration in the MBean server.

3. **Start the `Client`.**

   ```
   $ java -classpath client:mbeans:classpath Client
   ```

   You will see confirmation of the creation of the JMXMP connector client, the initialization of the environment map, the connection to the MBean server and the performance of the various MBean operations followed by the closure of the connection.

11.4.2 SASL Provider

In addition to the simple SASL security and the privacy-based security you can apply to JMXMP connectors, you can also define a custom SASL provider mechanism by defining your own SASL `Provider`.

The SASL mechanism used in this example is a straightforward customized mechanism called `SAMPLE SASL`. The example is found in the subdirectories of *examplesDir*`/current/Security/jmxmp/sasl_provider`. The `SAMPLE SASL` mechanism is defined in the `SampleServer` class, shown below.

EXAMPLE 11–11 Custom SASL Server

```
import java.io.*;
import javax.security.sasl.*;

public class SampleServer implements SaslServer {

    private String username;
    private boolean completed = false;

    public String getMechanismName() {
        return "SAMPLE";
    }

    public byte[] evaluateResponse(byte[] responseData)
```

EXAMPLE 11–11 Custom SASL Server *(Continued)*

```
            throws SaslException {
        if (completed) {
            throw new SaslException("Already completed");
        }
        completed = true;
        username = new String(responseData);
        return null;
    }

    public String getAuthorizationID() {
        return username;
    }

    public boolean isComplete() {
        return completed;
    }

    public byte[] unwrap(byte[] incoming, int offset, int len)
        throws SaslException {
        throw new SaslException("No negotiated security layer");
    }

    public byte[] wrap(byte[] outgoing, int offset, int len)
        throws SaslException {
        throw new SaslException("No negotiated security layer");
    }

    public Object getNegotiatedProperty(String propName) {
        if (propName.equals(Sasl.QOP)) {
            return "auth";
        }
        return null;
    }

    public void dispose() throws SaslException {
    }
}
```

The custom `SASL` `SAMPLE` mechanism defined by the `SampleServer` class, shown above, extends the SASL interface `SaslServer`. It simply calls the standard `SaslServer` methods one by one, to perform the following security operations:

- A call to `getMechanismName` establishes that the SASL mechanism used must be `SAMPLE`.
- Any response from the client end of the connection is evaluated by `evaluateResponse`, and an appropriate challenge to this response is generated.
- The authorization ID for this connection is built from the client's user name.

EXAMPLE 11–12 Custom SASL Server

```
public class SampleClient implements SaslClient {

    private byte[] username;
    private boolean completed = false;

    public SampleClient(String authorizationID) throws SaslException {
        if (authorizationID != null) {
            try {
                username = ((String)authorizationID).getBytes("UTF8");
            } catch (UnsupportedEncodingException e) {
                throw new SaslException("Cannot convert " + authorizationID
                                        + " into UTF-8", e);
            }
        } else {
            username = new byte[0];
        }
    }

    public String getMechanismName() {
        return "SAMPLE";
    }

    public boolean hasInitialResponse() {
        return true;
    }

    public byte[] evaluateChallenge(byte[] challengeData)
            throws SaslException {
        if (completed) {
            throw new SaslException("Already completed");
        }
        completed = true;
        return username;
    }

    public boolean isComplete() {
        return completed;
    }

    public byte[] unwrap(byte[] incoming, int offset, int len)
        throws SaslException {
        throw new SaslException("No negotiated security layer");
    }

    public byte[] wrap(byte[] outgoing, int offset, int len)
        throws SaslException {
        throw new SaslException("No negotiated security layer");
    }

    public Object getNegotiatedProperty(String propName) {
        if (propName.equals(Sasl.QOP)) {
            return "auth";
        }
```

EXAMPLE 11–12 Custom SASL Server *(Continued)*

```
        return null;
    }

    public void dispose() throws SaslException {
    }
}
```

The `SampleClient` class is an extension of the `SaslClient` interface, and calls the standard `SaslClient` methods one by one, to perform the following operations that are expected by the `SampleServer`:

- Defines a username, based on the authorization ID granted by the server.
- A call to `getMechanismName` establishes that the SASL mechanism used must be `SAMPLE`.
- Establishes that this SASL client implements the initial response mechanism expected by the server.
- Evaluates a challenge send by the server to the client following the client's initial response, and replies accordingly. If the client can provide the correct user name, then the `isComplete` method returns `true`, and the connection can proceed.

With these SASL server and client mechanisms defined, they can then be instantiated by their respective factories.

EXAMPLE 11–13 A Custom SASL Server Factory

```
public class ServerFactory implements SaslServerFactory {

     public SaslServer createSaslServer(String mechs,
                       String protocol,
                       String serverName,
                       Map props,
                       CallbackHandler cbh)
          throws SaslException {
             if (mechs.equals("SAMPLE")) {
                 return new SampleServer();
             }
             return null;
     }

    public String[] getMechanismNames(Map props) {
            return new String[]{"SAMPLE"};
    }
}
```

This basic implementation of the `SaslServerFactory` interface creates `SampleServer` instances when it is called with the SASL mechanism parameter set to `SAMPLE`. None of the other parameters are used by this mechanism.

The SampleClientFactory creates SampleClient instances in exactly the same way as the SampleServerFactory creates SampleServer instances.

EXAMPLE 11–14 SASL SAMPLE Provider Class

```
public final class Provider extends java.security.Provider {
    public Provider() {
    super("SampleSasl", 1.0, "SAMPLE SASL MECHANISM PROVIDER");
    put("SaslClientFactory.SAMPLE", "ClientFactory");
    put("SaslServerFactory.SAMPLE", "ServerFactory");
    }
}
```

The SASL SAMPLE Provider constructor shown above specifies the name of the provider (SampleSasl), the version of this provider (in this case, 1.0) and a description of the provider. It then defines the server and client factories that are used to create SASL servers and clients implementing this SASL mechanism.

With the mechanisms above thus defined, all the Server and Client classes used in this example require the correct Provider to be installed in the environment.

EXAMPLE 11–15 Adding a Provider to a JMX Connector Server

```
public class Server {

    public static void main(String[] args) {
        try {
            MBeanServer mbs = MBeanServerFactory.createMBeanServer();
            HashMap env = new HashMap();

            Security.addProvider(new Provider());
            env.put("jmx.remote.profiles", "SASL/SAMPLE");
            env.put("jmx.remote.x.access.file",
                  "config" + File.separator + "access.properties");

            JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);
            JMXConnectorServer cs =
                JMXConnectorServerFactory.newJMXConnectorServer(url,
                                                                 env,
                                                                 mbs);

            cs.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The SAMPLE SASL mechanism is passed into the Client in the same way.

▼ To Run the Secure JMXMP Connector Example with a SASL Provider

Run this example from within the *examplesDir*/current/Security/jmxmp/sasl_provider directory.

1. **Compile the Java classes.**

```
$ javac -classpath classpath \
      mbeans/SimpleStandard.java \
      mbeans/SimpleStandardMBean.java \
      server/Server.java \
      client/Client.java \
      client/ClientListener.java \
      sample/Provider.java \
      sample/ClientFactory.java \
      sample/ServerFactory.java \
      sample/SampleClient.java \
      sample/SampleServer.java
```

2. **Start the `Server`.**

```
$ java -classpath server:sample:mbeans:classpath Server &
```

You will see confirmation of the creation of the MBean server, the initialization of the environment map and the launching of the JMXMP connector and its registration in the MBean server.

3. **Start the `Client`.**

```
$ java -classpath client:sample:mbeans:classpath Client
```

You will see confirmation of the creation of the JMXMP connector client, the initialization of the environment map, the connection to the MBean server and the performance of the various MBean operations followed by the closure of the connection.

11.4.3 TLS Socket Factory

Your JMXMP connections can also be secured using an implementation of Transport Layer Security (TLS) sockets, as shown in the following example. This example is taken from the sub-directories of *examplesDir*/current/Security/jmxmp/tls_factory. The example shows how to provide a custom configured TLS factory for use by the client and the server.

EXAMPLE 11–16 Securing a JMXMP Connector Server Using TLS Socket Factories

```
public class Server {

    public static void main(String[] args) {
      try {
           MBeanServer mbs = MBeanServerFactory.createMBeanServer();
```

EXAMPLE 11–16 Securing a JMXMP Connector Server Using TLS Socket Factories *(Continued)*

```
            HashMap env = new HashMap();
            String keystore = "config" + File.separator + "keystore";
            char keystorepass[] = "password".toCharArray();
            char keypassword[] = "password".toCharArray();
            KeyStore ks = KeyStore.getInstance("JKS");
            ks.load(new FileInputStream(keystore), keystorepass);
            KeyManagerFactory kmf =
               KeyManagerFactory.getInstance("SunX509");
            kmf.init(ks, keypassword);
            SSLContext ctx = SSLContext.getInstance("TLSv1");
            ctx.init(kmf.getKeyManagers(), null, null);
            SSLSocketFactory ssf = ctx.getSocketFactory();
            env.put("jmx.remote.profiles", "TLS");
            env.put("jmx.remote.tls.socket.factory", ssf);
            env.put("jmx.remote.tls.enabled.protocols", "TLSv1");
            env.put("jmx.remote.tls.enabled.cipher.suites",
                  "SSL_RSA_WITH_NULL_MD5");

            JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);
            JMXConnectorServer cs =
               JMXConnectorServerFactory.newJMXConnectorServer(url,
                                                                env,
                                                                mbs);
            cs.start();

          } catch (Exception e) {
            e.printStackTrace();
          }
      }
  }
```

Example 11–16 shows the creation of an MBean server `mbs`, an environment map `env`, and a key store object `ks`. The key store object `ks` is initialized by loading in the input stream `keystore` and the key store password, `keystorepass`, that grants access to the key store.

An instance of the SSL class `KeyManagerFactory`, named `kmf`, that implements the SunX509 key management algorithm, is then initialized. The `kmf` instance is passed the key information contained in the key store `ks` and the key password `keypassword` that grants access to the keys in the key store.

The SSL context, `ctx`, is set to use the protocol version TLSv1, and is initialized with `kmf` as its key manager. Finally, an instance of `SSLSocketFactory`, named `ssf`, is created by calling the `getSocketFactory()` method of the SSL context `ctx`.

The environment map `env` is populated with the TLS profiles, the SSL socket factory `ssf`, the TLSv1 protocol version, and the `SSL_RSA_WITH_NULL_MD5` cipher suite.

Finally, with an instance of JMXConnectorServer, named cs, is created. The connector server cs is started by a call to the start() method of JMXConnectorServer.

EXAMPLE 11–17 Securing a JMXMP Connector Client Using TLS Socket Factories

```
public class Client {

   public static void main(String[] args) {
     try {
          HashMap env = new HashMap();
          String truststore = "config" + File.separator + "truststore";
          char truststorepass[] = "trustword".toCharArray();
          KeyStore ks = KeyStore.getInstance("JKS");
          ks.load(new FileInputStream(truststore), truststorepass);
          TrustManagerFactory tmf =
             TrustManagerFactory.getInstance("SunX509");
          tmf.init(ks);
          SSLContext ctx = SSLContext.getInstance("TLSv1");
          ctx.init(null, tmf.getTrustManagers(), null);
          SSLSocketFactory ssf = ctx.getSocketFactory();
          env.put("jmx.remote.profiles", "TLS");
          env.put("jmx.remote.tls.socket.factory", ssf);
          env.put("jmx.remote.tls.enabled.protocols", "TLSv1");
          env.put("jmx.remote.tls.enabled.cipher.suites",
             "SSL_RSA_WITH_NULL_MD5");

          JMXServiceURL url = new JMXServiceURL("jmxmp", null, 5555);
          JMXConnector jmxc = JMXConnectorFactory.connect(url, env);
          MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
          String domains[] = mbsc.getDomains();
          for (int i = 0; i < domains.length; i++) {
          System.out.println("Domain[" + i + "] = " + domains[i]);
          }

          ObjectName mbeanName =
              new ObjectName("MBeans:type=SimpleStandard");
          mbsc.createMBean("SimpleStandard", mbeanName, null, null);
          // Perform MBean operations
          //
          [...]
          mbsc.removeNotificationListener(mbeanName, listener);
          mbsc.unregisterMBean(mbeanName);
          jmxc.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
     }
}
```

As for the Server, Example 11–17 shows the creation of an environment map env, and a key store object ks for the JMXMP connector Client. The key store object ks is initialized by loading in the input stream truststore and the password, truststorepass, that grants access to the trust store.

An instance of the SSL class TrustManagerFactory, named tmf, that implements the SunX509 key management algorithm, is then initialized. The tmf instance is passed the key information contained in the key store ks.

The SSL context, ctx, is set to use the protocol version TLSv1, and is initialized with tmf as its trust manager. Finally, an instance of SSLSocketFactory, named ssf, is created by calling the getSocketFactory() method of the SSL context ctx.

The environment map env is populated with the the SSL socket factory ssf, the TLSv1 protocol version, and the SSL_RSA_WITH_NULL_MD5 cipher suite.

An instance of JMXConnector, named jmxc is then created, and an MBean server connection, mbsc, is made by calling the getMBeanServerConnection method of the JMX Remote API connector jmxc.

In code that is not shown here, when the connection to the MBean server has been established, the Client creates an MBean called SimpleStandard and performs various operations on it. Once these MBean operations have completed, the Client unregisters the MBean, and closes down the connection jmxc.

▼ To Run the Secure JMXMP Connector Example with TLS Socket Factories

Run this example from within the *examplesDir*/current/Security/jmxmp/tls_factory directory.

1. **Compile the Java classes.**

```
$ javac -classpath classpath \
      mbeans/SimpleStandard.java \
      mbeans/SimpleStandardMBean.java \
      server/Server.java \
      client/Client.java \
      client/ClientListener.java
```

2. **Start the Server.**

```
$ java -classpath server:mbeans:classpath Server &
```

You will see confirmation of the creation of the MBean server, the initialization of the environment map and the launching of the JMXMP connector and its registration in the MBean server.

3. **Start the Client.**

```
$ java -classpath client:mbeans:classpath Client
```

You will see confirmation of the creation of the JMXMP connector client, the initialization of the environment map, the connection to the MBean server and the performance of the various MBean operations followed by the closure of the connection.

PART **IV**

Agent Services

The minimal and base agents presented in Part II are manageable but are created empty. In this part we will examine agent services, MBeans that interact with your resources to provide management intelligence at the agent level. These services allow your agents to perform local operations on their resources that used to be performed by the remote manager. This frees you from having to implement them in the management application, and it reduces the amount of communication throughout your management solution.

In the Java dynamic management architecture, agents perform their own monitoring, avoiding constant manager polling. Agents also handle their own logical relations between objects, providing advanced operations that no longer require a relational database in the manager.

Java dynamic management agents are also much smarter, using discovery to be aware of their peers. They also connect to other agents to mirror remote resources, thus providing a single point of entry into a whole hierarchy of agents. Finally, agents are freed from their initial environment settings through dynamic downloading, allowing managers to push new classes to an agent effectively. As new resources and new services are developed, they can be loaded into deployed agents, boosting their capabilities without affecting their availability.

The benefit of agent services is increased by their dynamic nature. Agent services can come and go as they are required, either as determined by the needs of an agent's smart resources or on demand from the management application. The agent capability of self-management through the services is completely scalable: small devices might only allow monitors with some logic, whereas a server might embed the algorithms and networking capabilities to oversee its terminals autonomously.

This part contains the following chapters:

- Chapter 12 shows how the *m-let class loader* downloads new classes to the agent from a given URL (Universal Resource Locator). An m-let is a management applet: the HTML-style tag at the target URL that contains information about the classes to download. A manager can store new MBean classes anywhere, update its m-let file to reference them, and instruct an agent to load the classes. The new classes are created as MBeans, registered in the MBean server and ready to be managed.

- Chapter 13 shows how the *relation service* creates associations between MBeans, allowing for consistency checking against defined roles and relation types. New types can be defined dynamically to create new relations between existing objects. All relations are managed through the service so that it can expose query operations for finding related MBeans. Relations themselves can also be implemented as MBeans, allowing them to expose attributes and operations that act upon the MBeans in the relation.
- Chapter 14 shows how *cascading agents* allow a manager to access a hierarchy of agents through a single point-of-access. The subagents can be spread across the network, but their resources can be controlled by the manager through a connection to a single master agent. MBeans of a subagent are mirrored in the master agent and respond as expected to all management operations, including their removal. No special proxy classes are needed for the mirror MBean, meaning that every MBean object can be mirrored anywhere without requiring any class loading.
- Chapter 15 describes how the *discovery service* lets applications find Java dynamic management agents that are configured to be found. Using active discovery, the client broadcasts a discovery request over the network. Agents that have a discovery responder registered in their MBean server will automatically send a response. The client can then keep a list of reachable agents by using passive discovery to detect when responders are activated and deactivated. Along with the agent's address, the discovery response contains version information from the agent's delegate MBean and the list of available protocol adaptors and connectors.

CHAPTER **12**

M-Let Class Loader

The “dynamic” in Java Dynamic Management Kit (Java DMK) not only stands for dynamic loading, it also stands for dynamic *down*loading. The agent service that provides this functionality is the m-let class loader. *M-let* stands for *management applet*, an HTML-style tag that tells the class loader how to retrieve the desired class. Using this information, the m-let loader can retrieve an MBean class from a remote location given as a URL and create it in the agent.

The m-let resides in a separate text file that acts as a loading manifest. The contents of the m-let file let you specify any number of classes to load, possibly a different source for each, arguments for the class constructor, and the object name for the instantiated MBean. Because this mechanism is sometimes too heavy, the m-let loader can also be used to load classes directly and create MBeans in the agent.

The m-let loader is a service implemented as an MBean, so it can be called either directly by the agent or remotely by a management application. It can also be managed remotely, which allows a manager to effectively “push” MBeans to an agent: the manager instantiates the m-let loader in an agent and instructs it to load classes from a predetermined location.

This chapter covers the following topics:

- “12.1 M-Let Loader” on page 209
- “12.2 Secure Class Loading” on page 214

12.1 M-Let Loader

In the Java DMK 5.1, the m-let loader is a class loader object that extends the `URLClassLoader` class of the `java.net` package to simplify the downloading service it provides.

The m-let loader service is an instance of the `MLet` class in the `javax.management.loading` package. It is also an MBean that can be accessed remotely. It provides m-let file loading and a shortcut method. In addition, can be used directly as a class loader, without requiring an m-let file.

We will start by demonstrating the usage of the m-let service as it would be used in an agent or in an MBean. In our example, the agent application creates an MBean server and then the m-let loader.

EXAMPLE 12–1 Instantiating the `MLet` Class

```
// Parse command line arguments.
[...]

// Instantiate the MBean server
MBeanServer server = MBeanServerFactory.createMBeanServer();
String domain = server.getDefaultDomain();

// Create a new MLet MBean and add it to the MBeanServer.
String mletClass = "javax.management.loading.MLet";
ObjectName mletName = new ObjectName(domain + ":name=" + mletClass);
server.createMBean(mletClass, mletName);
```

There is no special initialization that needs to be done before loading classes through an m-let file.

12.1.1 Loading MBeans from a URL

In order to download an MBean, we must first have its corresponding m-let definition in an HTML file. In our example, we define the following file with two `MLET` tags.

EXAMPLE 12–2 The M-Let File

```
<HTML>
<MLET
  CODE=EquilateralTriangle.class
  ARCHIVE=EquilateralTriangle.jar
  NAME=MLetExample:name=EquilateralTriangle,id=1
>
<ARG TYPE=java.lang.Integer VALUE=8>
</MLET>
<MLET
  CODE=EquilateralTriangle.class
  ARCHIVE=EquilateralTriangle.jar
  NAME=MLetExample:name=EquilateralTriangle,id=2
>
<ARG TYPE=java.lang.Integer VALUE=15>
</MLET>
</HTML>
```

This file tells the m-let loader to create two MBeans with the given object names, using the given classes in the JAR files. The JAR files must be located in the same directory as this file, regardless of whether the directory is on a local or remote host. The MLET tag can also specify a CODEBASE, which is an alternate location for the JAR file. The MLET tag is fully defined in the JMX specification.

To download the MBeans specified in the m-let file we call the getMBeansFromURL method and analyze the result.

EXAMPLE 12–3 Calling the getMBeansFromURL Method

```
// the url_2 string is read from the command line
echo("\tURL = " + url_2);
Object mletParams_2[] = {url_2};
String mletSignature_2[] = {"java.lang.String"};
Set mbeanSet = (Set) server.invoke(mletName, "getMBeansFromURL",
     mletParams_2, mletSignature_2);

for (Iterator i = mbeanSet.iterator(); i.hasNext(); ) {
    Object element = i.next();
    if (element instanceof ObjectInstance) {
        // Success, we display the new MBean's name
        echo("\tOBJECT NAME =
            " + ((ObjectInstance)element).getObjectName());
    } else {
        // Failure, we display why
        echo("\tEXCEPTION = " + ((Throwable)element).getMessage());
    }
}
```

The m-let loader is the class loader, and it handles just a list of code-bases that it has accessed directly. You can view this list by calling the getURLs method of the m-let loader MBean.

This behavior means that the getMBeansFromURL method does not need to return the object names of class loaders it has used. Instead it just returns either the object instance of the downloaded and registered MBean or a Throwable object in case of an error or an exception. These are returned in a Set object containing as many elements as there are MLET tags in the target m-let file.

12.1.2 Shortcut for Loading MBeans

This behavior also simplifies any repeated loading of the classes after they have been loaded from an m-let file. Because the m-let loader has already used the code-base of the MBean, it is available to be used again. All you need to do is specify the object name of the m-let loader as the class loader when creating the MBean.

You can also load other MBeans in the same code-base, once the code-base has been accessed by a call to the getMBeansFromURL method. In our example we will just download another MBean of the EquilateralTriangle class.

EXAMPLE 12–4 Reloading Classes in the M-Let Class Loader

```
// Create another EquilateralTriangle MBean from its class
// in the EquilateralTriangle.jar file.
String triangleClass = "EquilateralTriangle";
ObjectName triangleName = new ObjectName(
    "MLetExample:name=" + triangleClass + ",id=3");
Object triangleParams[] = {new Integer(20)};
String triangleSignature[] = {"java.lang.Integer"};

server.createMBean(triangleClass, triangleName, mletName,
                   triangleParams, triangleSignature);
```

Again, loading classes from known code-bases or reloading a class directly from its JAR file implies that the agent or MBean programmer has some knowledge of the code-bases and JAR file contents at runtime.

12.1.3 Loading MBeans Directly

Because the m-let loader object is a class loader, you can use it to load classes directly, without needing to define an m-let file.

Before you can load an MBean directly, you need to add the URL of its code-base to the m-let loader's internal list. Then we just use the m-let loader's object name as the class loader name when creating the MBean. Here is the code to do this in the agent example.

EXAMPLE 12–5 Using the M-Let MBean as a Class Loader

```
// Add a new URL to the MLet class loader
// The url_1 string is read from the command line
Object mletParams_1[] = {url_1};
String mletSignature_1[] = {"java.lang.String"};
server.invoke(mletName, "addURL", mletParams_1, mletSignature_1);

// Create a Square MBean from its class in the Square.jar file.
String squareClass = "Square";
ObjectName squareName = new ObjectName(
    "MLetExample:name=" + squareClass);
Object squareParams[] = {new Integer(10)};
String squareSignature[] = {"java.lang.Integer"};
server.createMBean(squareClass, squareName, mletName,
                   squareParams, squareSignature);
```

You only need to add the URL to the m-let loader the first time you download a class. Once it is added, you can load it as many times as necessary by calling `createMBean` directly.

Because this loading mechanism does not use the `MLET` tag, the programmer must ensure that either the downloaded class provides its own object name or, as in Example 12–5, the agent provides one.

The fact that the m-let loader is also a class loader into which you can load multiple URLs raises the issue of name spaces. If there are two classes with the same name within the code-bases defined by the set of all URLs, the m-let loader will load one of them non-deterministically. To specify one of them precisely, do not add the URL of the second code-base to the m-let loader. Instead, create a second m-let loader MBean to which you can add the URL for the second version of the class. In this case, you will have one m-let MBean that can load one version of the class and another m-let MBean that can load the other.

12.1.4 Running the M-Let Agent Example

This example is located in the *examplesDir*`/current/MLet/` directory. See "Directories and Classpath" in the Preface for details.

In our example, we have two MBeans representing geometrical shapes. Before running the example, we compile them and create a JAR file for each. We also compile the agent application at the same time.

```
$ cd examplesDir/current/MLet/
$ javac -classpath classpath *.java

$ jar cf Square.jar Square.class SquareMBean.class
$ rm Square.class SquareMBean.class

$ jar cf EquilateralTriangle.jar EquilateralTriangle.class \
EquilateralTriangleMBean.class
$ rm EquilateralTriangle.class EquilateralTriangleMBean.class
```

The agent command line requires you to specify first the URL of a JAR file for directly loading the `Square` class, then the URL of the m-let file. We have left these files in the examples directory, but you could place them on a remote host. With the Korn shell on the Solaris platform, type the following command:

```
$ java -classpath classpath Agent \
file:${PWD}/Square.jar file:${PWD}/GeometricShapes.html
```

In the output of the agent, you can see it create the m-let loader MBean, and then download classes to create MBeans. It starts with the direct loading of the `Square` class, and then loads from the HTML file that specifies two `EquilateralTriangle` MBeans to be loaded. Once these have been loaded, we can see the third one that is loaded through the class loader shortcut.

The agent then starts an HTML adaptor so that we can easily view the new MBeans. In them we can see that the values contained in the `ARG` tags of the m-let file were used to initialize the MBeans. Point your web browser to the following URL and click on the MBeans in the `MLetExample` domain:

```
http://localhost:8082/
```

When you have finished, press Control-C in the window where you started the agent.

12.2 Secure Class Loading

Because class loading exposes an agent to external classes, the Java DMK offers security within the m-let service.

12.2.1 Code Signing

Code signing is a security measure that you can use to identify the originator of a downloaded class. The m-let service will enforce code signatures if it is instantiated in secure mode. One of the constructors of the MLet class takes a boolean parameter that specifies the security mode. For obvious security reasons, the security mode cannot be modified once the m-let service is instantiated.

When the m-let service is running in secure mode, it will only load classes and native libraries that are signed by a trusted party. A trusted party is identified by a key: this key was used to sign the code and a copy of the key is given to all parties that want to download the signed class. Therefore, you must identify trusted keys in your agent before attempting to download their signed classes.

Note – Downloading native libraries always requires a custom security manager, regardless of whether they are trusted or not.

In the MLet class, security is not determined when you instantiate the m-let service. Rather, security is enabled or disabled for your entire agent application, including any class loaders used by the m-let service.

To enable security, start your agent applications with the java.lang.SecurityManager property on the command line. Then, when the m-let service loads a class through one of its class loaders, the class loader will check the origin and signature of the class against the list of trusted origins and signatures.

The tools involved in signing a class file are the jar, keytool, and jarsigner utilities. On the host where the agent application will download a class, you define a set of permissions for signatures and URL origins. Then, you need to use the policytool utility to generate a java.policy file containing the trusted signatures. Refer to the Java 2 Platform Standard Edition (J2SE) documentation for the description of these utilities.

When the agent application is started with a security manager, it will check this policy file to ensure that the origin and signature of a downloaded class match a trusted origin and a trusted signature. If they do not match, the code is not trusted and cannot be loaded.

When the agent application is started without the security manager, all classes and native libraries can be downloaded and instantiated, regardless of their origin and signature, or lack thereof.

CHAPTER 13

Relation Service

The relation service defines and maintains logical relations between MBeans in an agent. It acts as a central repository of relation types and relation instances, and it ensures the consistency of all relations it contains. Local and remote applications that access the agent can define new types of relations and then declare a new instance of a relation between MBeans.

You can only create relations that respect a known relation type, and the service allows you to create new relation types dynamically. Then you can access the relations and retrieve the MBeans in specific roles. The relation service listens for MBeans being unregistered and removes relation instances that no longer fulfill their relation type. It sends notifications of its own to signal when relation events occur.

Java Dynamic Management Kit (Java DMK) also exposes the interface classes for defining your own relation objects. By creating relation instances as objects, you can implement them as MBeans that can expose operations on the relation they represent.

Like the other services, the relation service is instrumented as an MBean, allowing remote access and management.

The code samples in this topic are taken from the files in the `Relation` directory located in the main *examplesDir* (see “Directories and Classpath” in the Preface).

This chapter covers the following topics:

- “13.1 Defining Relations” on page 218 demonstrates how to create relations through the relation service.
- “13.2 Operations of the Relation Service” on page 223 covers the operations for manipulating relations and their consequences.
- “13.3 Objects Representing Relations” on page 226 shows how to define relations outside of the relation service, yet under its control.
- “13.4 Running the Relation Service Example” on page 231 demonstrates the functionality of this service.

13.1 Defining Relations

A relation is composed of named *roles*, each of which defines the cardinality and class of MBeans that will be put into association with the other roles. A set of one or more roles defines a *relation type*. The relation type is a template for all relation instances that associate MBeans representing its roles. We use the term *relation* to mean a specific instance of a relation that associates existing MBeans according to the roles in its defining relation type.

For example, we can say that `Books` and `Owner` are roles. `Books` represents any number of owned books of a given MBean class, and `Owner` is a single book owner of another MBean class. We might define a relation type containing these two roles and call it `Personal Library`. It represents the concept of book ownership.

The following diagram represents this sample relation, as compared to the unified modeling language (UML) modeling of its corresponding association.

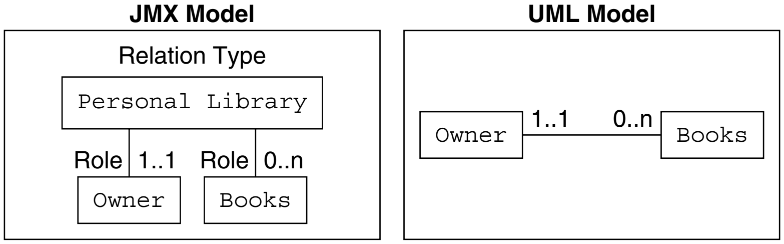


FIGURE 13–1 Comparison of the Relation Models

There is a slight difference between the two models. The UML association implies that each one of the `Books` can only have one owner. Our relation type only models a set of roles, guaranteeing that a relation instance has one `Owner` MBean and any number of MBeans in the role of `Books`.

Note – The relation service does not do inter-relation consistency checks; if they are needed they are the responsibility of the designer. In our example, the designer needs to ensure that the same book MBean does not participate in two different `Personal Library` relations, while allowing it for an owner MBean.

In the rest of this topic, we will see how to handle the roles, relation types and relation instances through the relation service. We will use the names and relations presented in the programming example. First of all, we instantiate the relation service as we would any other MBean and register it in our agent's MBean server.

EXAMPLE 13–1 Relation Service MBean

```
// Build ObjectName of RelationService
    //
String relServClassName = RelationService.class.getName();
ObjectName relServObjName = createMBeanName(relServClassName, 1);

[...]

Object[] params = new Object[1];
params[0] = new Boolean(true);
String[] signature = new String[1];
signature[0] = "boolean";

server.createMBean(theRelServClassName, theRelServObjName,
                   params, signature);
```

The relation service exposes an attribute called `Active` that is `false` until its MBean is registered with the MBean server. All of the operations that handle relation or role MBeans, either directly or indirectly, will throw an exception when the service is not active.

13.1.1 Defining Role Information

Before we can create relation instances, we need a relation type, and before we can define a relation type, we need to represent the information about its roles. This information includes:

- A name string (required)
- The name of the MBean class that fulfills this role (required)
- Read-write permissions (the default is both readable and writable)
- The multiplicity, expressed as a single range (the default is 1..1)
- A description string (the default is a null string)

The name can be any string that is manipulated by the Java `String` class. It will be used to retrieve the corresponding MBeans when accessing a relation instance. The multiplicity is limited to the range between the minimum and maximum number of MBeans required for this role to fulfilled. The read-write permissions apply to all of the MBeans in the role, since the value of a role is read or written as a complete list.

The role information is represented by the `RoleInfo` class. In our example, we define two different roles.

EXAMPLE 13–2 Instantiating `RoleInfo` Objects

```
// Define two roles in an array
// - container: SimpleStandard class/read-write access/multiplicity: 1..1
// - contained: SimpleStandard class/read-write access/multiplicity: 0..n

RoleInfo[] roleInfoArray = new RoleInfo[2];
String role1Name = "container";
```

EXAMPLE 13–2 Instantiating `RoleInfo` Objects *(Continued)*

```
roleInfoArray[0] =
    createRoleInfo( role1Name, "SimpleStandard",
                  true, true,
                  1, 1,
                  null);

String role2Name = "contained";
roleInfoArray[1] =
    createRoleInfo( role2Name, "SimpleStandard",
                  true, true,
                  0, -1,
                  null);
```

We build an array of `RoleInfo` objects that is intended to define a relation type, so it needs to define a valid set of roles. All role names must be unique within the array, and none of the array's elements can be null. Also, the minimum and maximum cardinalities must define a range of at least one integer.

13.1.2 Defining Relation Types

We define a relation type in the relation service by associating a name for the relation type with a non-empty array of `RoleInfo` objects. These are the parameters to the service's `createRelationType` method that we call through the MBean server.

EXAMPLE 13–3 Defining a Relation Type

```
try {
    String relTypeName = "myRelationType";

    Object[] params = new Object[2];
    params[0] = theRelTypeName;
    params[1] = theRoleInfoArray;
    String[] signature = new String[2];
    signature[0] = "java.lang.String";
    // get the string representing the "RoleInfo[]" object
    try {
         signature[1] =
                  (theRoleInfoArray.getClass()).getName();
         } catch (Exception exc) {
             throw exc;
         }
         server.invoke(theRelServObjName,
                         "createRelationType",
                         params,
                         signature);
    } catch (Exception e) {
         echo("\tCould not create the relation type "
              + relTypeName);
```

EXAMPLE 13–3 Defining a Relation Type *(Continued)*

```
        printException(e);
    }
}
```

The relation type name given by the calling process must be unique among all relation type names already created in the relation service. Relations will refer to this name to define their type, and the service will verify that the roles of the relation match the role information in this type.

The relation service provides methods for managing the list of relation types it stores. The `removeRelationType` removes the type's definition from the relation service and also removes all relation instances of this type. This mechanism is further covered in "13.2.3 Maintaining Consistency" on page 224.

Other methods give the list of all currently defined relation types or the role information associated with a given type. Role information is always obtained through the name of the relation type where it is defined. Here we show the subroutine that our example uses to print out all role and type information.

EXAMPLE 13–4 Retrieving Relation Types and Role Information

```
try {
    echo("\n-> Retrieve all relation types");
    Object[] params1 = new Object[0];
    String[] signature1 = new String[0];
    ArrayList relTypeNameList = (ArrayList)
    (server.invoke( relServObjName, "getAllRelationTypeNames",
                    params1, signature1));

    for (Iterator relTypeNameIter = relTypeNameList.iterator();
         relTypeNameIter.hasNext(); ) {
        String currRelTypeName = (String)(relTypeNameIter.next());
        echo("\n-> Print role info for relation type " +
             currRelTypeName);
        Object[] params2 = new Object[1];
        params2[0] = currRelTypeName;
        String[] signature2 = new String[1];
        signature2[0] = "java.lang.String";
        ArrayList roleInfoList = (ArrayList)
            (server.invoke( relServObjName,"getRoleInfos",
                            params2, signature2));
        printList(roleInfoList);
    }
} catch (Exception e) {
    echo("\tCould not browse the relation types");
    printException(e);
}
```

13.1.3 Creating Relations

Now that we have defined a relation type, we can use it as a template for creating a relation. A *relation* is a set of roles that fulfills all of the role information of the relation type. The `Role` object contains a role name and value which is the list of MBeans that fulfills the role. The `RoleList` object contains the set of roles used when setting a relation or getting its role values.

In order to create a relation we must provide a set of roles whose values will initialize the relation correctly. In our example we use an existing `SimpleStandard` MBean in each role that we have defined. Their object names are added to the value list for each role. Then each `Role` object is added to the role list.

EXAMPLE 13–5 Initializing Role Objects and Creating a Relation

```
[...] // define object names and create SimpleStandard MBeans

// Instantiate the roles using the object names of the MBeans
ArrayList role1Value = new ArrayList();
role1Value.add(mbeanObjectName1);
Role role1 = new Role(role1Name, role1Value);

ArrayList role2Value = new ArrayList();
role2Value.add(mbeanObjectName2);
Role role2 = new Role(role2Name, role2Value);

RoleList roleList1 = new RoleList();
roleList1.add(role1);
roleList1.add(role2);

String relId1 = relTypeName + "_internal_1";

try {
    Object[] params = new Object[3];
    params[0] = theRelId1;
    params[1] = theRelTypeName;
    params[2] = roleList1;
    String[] signature = new String[3];
    signature[0] = "java.lang.String";
    signature[1] = "java.lang.String";
    signature[2] = "javax.management.relation.RoleList";
    server.invoke(theRelServObjName, "createRelation",
                  params, signature);
} catch(Exception e) {
    echo("\tCould not create the relation " + RelId1);
    printException(e);
}
```

The createRelation method will raise an exception if the provided roles do not fulfill the specified relation type. You can omit Role objects for roles that allow a cardinality of 0; their values will be initialized with an empty list of object names. The relation service will check all provided object names to ensure they are registered with the MBean server. Also, the relation identifier name is a string that must be unique among all relations in the service.

The corresponding removeRelation method is also exposed for management operations. It is also called internally to keep all relations coherent, as described in "13.2.3 Maintaining Consistency" on page 224. In both cases, removing a relation means that you can no longer access it through the relation service, and the isRelation operation will return false when given its relation identifier. Also, its participating MBeans will no longer be associated through the roles in which they belonged. The MBeans continue to exist unaltered otherwise and can continue to participate in other relations.

13.2 Operations of the Relation Service

In addition to the creation and removal methods for relation types and instances, the relation service provides operations for finding related MBeans, determining the role of a given MBean, and accessing the MBeans of a given role or relation.

13.2.1 Query Operations

Once relations have been defined, the relation service allows you to do searches based on the association of objects that the relations represent. The following operations perform queries on the relations:

- findAssociatedMBeans: Returns a list of all object names referenced in any relation where a given object name appears; each of these object names is mapped to the list of relation identifiers where the two MBeans are related, since the pair can be related through different relation instances.

  Two optional parameters let you specify a relation type and role name. When either or both of these are specified, the only relations to be considered are those of the given type and/or where the given object name appears in the named role.
- findReferencingRelations: Takes an object name and returns the list of relation identifiers where it is referenced; each identifier is mapped to the list of roles in which the corresponding MBean appears in that relation; again, you can specify the relation type and/or role name in which the given MBean must appear.
- getReferencedMBeans: Returns a list of all MBeans currently in a given relation; their object names are mapped to the role names where they are referenced, since the same MBean can appear in more than one role of the same relation.

- findRelationsOfType: Returns the list of identifiers of all relations that were created or added with a given relation type.
- getRelationTypeName: This method is the inverse of the previous, returning the relation type name of the relation with a given identifier.

13.2.2 Accessing Roles

Once you have a relation identifier, you will probably want to access its role values. The relation service provides getters and setters for roles, as well as bulk operations to get or set several or all roles in a relation. Remember that the value of a role is a list of MBeans, and a role is identified by a name string.

Input parameters to setter operations are the same Role and RoleList classes that are used to create relations. Bulk getters and setters return a RoleResult object that contains separate lists for successful and failed individual role operations.

Inside a role result, the list of roles and values that were successfully accessed are given in a RoleList instance. The information about roles that could not be read or written is returned in a RoleUnresolvedList. This list contains RoleUnresolved objects that name the role that could not be accessed and an error code explaining the reason, such as an invalid role name or the wrong cardinality for a role. The error codes are defined as static final fields in the RoleStatus class.

13.2.3 Maintaining Consistency

All relation service operations that set the role of a relation always verify that the role value fulfills its definition in the relation type. An incorrect MBean type, the wrong cardinality of the role, or an unknown role name will prevent that role from being modified, guaranteeing that the relation always fulfills it relation type.

As we shall see in “13.3 Objects Representing Relations” on page 226, relations and relation types can also be objects that are external to the relation service. In this case, they are responsible for maintaining their own role-consistency. The relation service MBean exposes methods to assist these objects in verifying that their roles conform to their defined relation type. In all cases of an external implementation of a relation, the object designer is responsible for ensuring its coherence.

Removing a relation type can cause existing relations to cease to have a defining type. The policy of the removeRelationType operation is to assume that the caller is aware of existing relations of the given type. Instead of forbidding the operation, this method removes the relations that were defined by the given type. It is the designer’s responsibility to first call the findRelationsOfType operation to determine if any existing relations will be affected.

MBeans participating in a relation can be removed from the MBean server by some other management operation, thereby modifying the cardinality of a role where they were referenced. In order to maintain consistency in this case, the relation service must remove all relations in which a role no longer has the cardinality defined in its relation type. The process of determining invalid relations and removing them is called purging.

The relation service listens for unregistration notifications of the MBean server delegate, and will need to purge its MBeans whenever one is received. It must determine if the MBean removed was involved in any relations, and if so, whether its removal violates the cardinality of each role where it appears. The relation service exposes the boolean `PurgeFlag` attribute that the programmer must set to determine whether purges are done automatically or not.

When the purge flag is `true`, the relation service will purge its data immediately after every unregistration notification. However, the purge operation can be resource intensive for large sets of relation data. In this case the managing application can set the purge flag to `false` and only call the `purgeRelations` operation to purge the data when needed.

For example, if there are many MBean unregistrations and few relation operations, it might make sense to only purge the relation service manually before each operation. Or the automatic purge flag can be temporarily set to false while executing time-critical operations that need to remove MBeans, but that will not access the relation service.

There are two possible consequences of an unpurged relation service. Roles in a relation can reference object names that no longer have an associated MBean. Or worse, the object name might have been reassigned to another MBean, leading to a totally incoherent state. The designer of the management solution is responsible for setting the purge policy so that operations will always access consistent relation values.

13.2.4 Relation Service Notifications

The relation service is a notification broadcaster that notifies listeners of events affecting relations. It sends `RelationNotification` objects in the following cases:

- A new relation is **created**: The notification contains its identifier and its relation type
- An existing relation is **updated**: In addition to the relation identifier and type, the notification contains the list of names, the list of new values, and the list of old values for all roles that have been modified
- A relation is **removed**: The notification contains its identifier and its relation type

There are three equivalent notification types for events affecting relations defined as external MBeans (see "13.3 Objects Representing Relations" on page 226). In addition to the role and relation information, these notifications contain the object name of this MBean.

13.3 Objects Representing Relations

When creating relation types and instances, their representations are handled internally and only accessed through the interface of the relation service. However, the service also allows you to create external objects that represent relations, and then add them under the service's control to access them through its operations.

One advantage of representing relation types and instances as classes is that they can perform their initialization in the class constructor. Applications can then instantiate the classes and add them to the relation service directly, without needing to code for their creation. The relation type classes can be downloaded to agent applications for use throughout the management architecture.

Another advantage is that relations are represented as MBeans and must be registered in the MBean server. This means that management applications can get role information directly from the relation MBean instead of going through the relation service.

The main advantage of an external relation class is that it can be extended to add properties and methods to a relation. They can be accessible through attributes and operations in the relation MBean so that they are also exposed for management. These extensions can represent more complex relations and allow more flexible management architectures.

With the power of manipulating relations comes the responsibility of maintaining the relation model. A relation MBean can expose an operation for adding an object name to a role, but its implementation must first ensure that the new role value will conform to the relation's type. Then, it must also instruct the relation service to send a role update notification. The relation service MBean exposes methods for maintaining consistency and performing required operations. It is the programmer's responsibility to call the relation service when necessary in order to maintain the consistency of its relation data.

13.3.1 `RelationTypeSupport` Class

A class must implement the `RelationType` interface in order to be considered a representation of a relation type. The methods of this interface are used to access the role information that makes up a relation type. Since relation types are immutable within the relation service, there are no methods for modifying the role information.

The `RelationTypeSupport` class is the implementation of this interface that is used internally by the relation service to represent a relation type. By extending this class, you can quickly write new relation type classes with all the required functionality. The class has a method for adding roles to the information that is exposed; this method can be called by the class constructor to initialize all roles. Our simple example does just this, and there is little other functionality that can be added to a relation type object.

EXAMPLE 13–6 Extending the `RelationTypeSupport` Class

```
import javax.management.relation.*;

public class SimpleRelationType extends RelationTypeSupport {

    // Constructor
    public SimpleRelationType(String theRelTypeName) {

        super(theRelTypeName);

        // Defines the information for two roles
        // - primary: SimpleStandard class/read-write/cardinality=2
        // - secondary: SimpleStandard class/read-only/cardinality=2
        try {
            RoleInfo primaryRoleInfo =
                new RoleInfo("primary", "SimpleStandard",
                             true, true,
                             2, 2,
                             "Primary description");
            addRoleInfo(primaryRoleInfo);

            RoleInfo secondaryRoleInfo =
                new RoleInfo("secondary", "SimpleStandard",
                             true, false,
                             2, 2,
                             "Secondary description");
            addRoleInfo(secondaryRoleInfo);
        } catch (Exception exc) {
            throw new RuntimeException(exc.getMessage());
        }
    }
}
```

We now use our class to instantiate an object representing a relation type. We then call the relation service's `addRelationType` operation to make this type available in the relation service. Thereafter, it is manipulated through the service's operations and there is no way to distinguish it from other relation types that have been defined.

EXAMPLE 13–7 Adding an Externally Defined Relation Type

```
String usrRelTypeName = "SimpleRelationType";
SimpleRelationType usrRelType =
    new SimpleRelationType("SimpleRelationType");
try {
    Object[] params = new Object[1];
```

EXAMPLE 13–7 Adding an Externally Defined Relation Type *(Continued)*

```
    params[0] = usrRelType;
    String[] signature = new String[1];
    signature[0] = "javax.management.relation.RelationType";
    server.invoke( relServObjName, "addRelationType",
                   params, signature);
} catch(Exception e) {
    echo("\tCannot add user relation type");
    printException(e);
}
```

The role information defined by a relation type should never change once the type has been added to the relation service. This is why the `RelationTypeSupport` class is not an MBean: it would make no sense to manage it remotely. All of the information about its roles is available remotely through the relation service MBean.

13.3.2 `RelationSupport` Class

The external class representation of a relation instance must implement the `Relation` interface so that it can be handled by the relation service. The `RelationSupport` class is the implementation provided which is also used internally by the service.

The methods of the relation interface expose all of the information needed to operate on a relation instance: the getters and setters for roles and the defining relation type. Because the relation support class must represent any possible relation type, it has a constructor that takes a relation type and a role list, and it uses a generic mechanism internally to represent any roles.

You could implement a simpler relation class that implements a specific relation type, in which case it would know and initialize its own roles. The class could also interact with the relation service to create its specific relation type before adding itself as a relation.

However, the simplest way to define a relation object is to extend the `RelationSupport` class and provide any additional functionality you require. In doing so, you can rely on the relation support class's own methods for getting and setting roles, thereby taking advantage of their built-in consistency mechanisms.

EXAMPLE 13–8 Extending the `RelationSupport` Class

```
import javax.management.ObjectName;
import javax.management.relation.*;

public class SimpleRelation extends RelationSupport
    implements SimpleRelationMBean {

    // Constructor
    public SimpleRelation(String theRelId,
```

EXAMPLE 13–8 Extending the `RelationSupport` Class *(Continued)*

```
                          ObjectName theRelServiceName,
                          String theRelTypeName,
                          RoleList theRoleList)
        throws InvalidRoleValueException,
               IllegalArgumentException {

        super(theRelId, theRelServiceName, theRelTypeName, theRoleList);
    }

    [...] // Implementation of the SimpleRelationMBean interface
}
```

Our MBean's `SimpleRelationMBean` interface extends the `RelationSupportMBean` in order to expose its operations for management. In order to represent a relation, the class must be an MBean registered in the MBean server. This allows the relation service to rely on unregistration notifications in order to find out if the object name is no longer valid.

When instantiating our `SimpleRelation` MBean, we use the relation type defined in Example 13–3. We also reuse the role list from Example 13–5 that contains a single MBean in each of two roles. Before adding the relation to the relation service, we must create it as an MBean in the MBean server. We then call the `addRelation` operation of the relation service with our relation's object name.

EXAMPLE 13–9 Creating an External Relation MBean

```
// Using relTypeName="myRelationType"
// and roleList1={container,contained}

String relMBeanClassName = "SimpleRelation";
String relId2 = relTypeName + "_relMBean_2";
ObjectName relMBeanObjName1 = createMBeanName(relMBeanClassName, 2);
createRelationMBean(relMBeanObjName1,
                    relMBeanClassName,
                    relId2,
                    relTypeName,
                    roleList1,
                    relServObjName);
try {
    Object[] params1 = new Object[4];
    params1[0] = theRelId2;
    params1[1] = theRelServObjName;
    params1[2] = theRelTypeName;
    params1[3] = roleList1;
    String[] signature1 = new String[4];
    signature1[0] = "java.lang.String";
    signature1[1] = "javax.management.ObjectName";
    signature1[2] = "java.lang.String";
    signature1[3] = "javax.management.relation.RoleList";
    server.createMBean(theRelMBeanClassName, theRelMBeanObjName,
                       params1, signature1);
```

EXAMPLE 13–9 Creating an External Relation MBean *(Continued)*

```
    addRelation(theRelMBeanObjName, theRelServObjName);

} catch(Exception e) {
    echo("\t Could not create relation MBean for relation " + relId2);
    printException(e);
}

// Add our new MBean as a relation to the relation service
try {
    Object[] params2 = new Object[1];
    params2[0] = theRelMBeanObjName;
    String[] signature2 = new String[1];
    signature2[0] = "javax.management.ObjectName";
    server.invoke(theRelServObjName, "addRelation",
                  params2, signature2);
} catch(Exception e) {
    echo("\t Could not add relation MBean " + relMBeanObjName);
    printException(e);
}
```

After a relation instance is added to the relation service, it can be accessed normally like other relations. Management operations can either operate on the MBean directly or access it through its identifier in the relation service. Two methods of the service are specific to external relation MBeans:

- `isRelationMBean` - Takes a relation identifier and returns the object name of the relation MBean, if it is defined
- `isRelation` - Takes an object name and returns its relation identifier, if it is a relation MBean that has been added to the relation service

In our example, our MBean does not expose any functionality that interacts with the relation service. It relies fully on the support class and only adds the implementation of its own simple MBean interface. In a more realistic example, the MBean would expose attributes or operations related to its role values.

For example, `Personal Library` could be a unary relation type containing the `Books` role. We could then design an `Owner` MBean to be a relation of this type. In addition to exposing attributes about the owner, the MBean would give the number of books in the library, return an alphabetized list of book titles, and provide an operation for selling a book. All of these would need to access the role list, and the sale operation would need to modify the role to remove a book MBean.

All of these operations would need to keep the consistency of the relation. To do this, the relation service exposes several methods that relation MBeans must call:

- `checkRoleReading`: Verifies the read access for the role of a given relation type, before it is read

- `checkRoleWriting`: Verifies the write access, multiplicity and MBean class of a role value before it can be written
- `updateRoleMap`: Provides the old and new values of a role so that the relation service can update its internal lists of referenced MBeans
- `SendRoleUpdateNotification`: Instructs the relation service to send a notification containing the given old and new values of a role; since the relation service must be the broadcaster for all relation notifications, relation MBeans must call this method after modifying a role value

13.4 Running the Relation Service Example

The *examplesDir*`/current/Relation` directory contains all of the files for the agent application and the associated MBeans.

▼ To Run the Relation Service Example

1. **Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, type:**

   ```
   $ cd examplesDir/current/Relation/
   $ javac -classpath classpath *.java
   ```

2. **Start the relation service application with the following command. Be sure that the classes for the `SimpleRelationType` and `SimpleRelation` MBean can be found in its *classpath*.**

   ```
   $ java -classpath classpath RelationAgent
   ```

   When started, the application first creates the `RelationService` MBean and then begins its long demonstration of operations on roles, types and relations. The output of each step is separated by a horizontal line to detect its starting point in the scrolling display.

3. **Press `Enter` to step through the example when the application pauses, or press `Control-C` at any time to exit.**

CHAPTER 14

Cascading Service

The cascading service enables you to access the MBeans of a subagent directly through the MBean server of a master agent. The cascading service has been completely overhauled in Java Dynamic Management Kit (Java DMK) 5.1 to allow it to operate over the connector protocols defined by the Java Management Extensions (JMX) Remote API. The legacy cascading service is now deprecated. The examples of the legacy cascading service have been retained in Chapter 25, for reasons of backwards compatibility. However, when using legacy Java DMK connectors, you should use the new `CasdingServiceMBean` with wrapped legacy connectors rather than relying on the deprecated legacy cascading agent API.

The service is implemented in a `CascadingServiceMBean`, which makes it possible to mount *source MBean servers* (that are possibly located in subagents) into a *target MBean server* (that is located in a Master Agent) in a manner that is somewhat analogous to a File System mount operation. Several source MBean servers can be mounted in the same target MBean server, provided that different *target paths* are used. A source MBean server mounted from a subagent can itself be a Master Agent in which source MBean servers from another level of subagents are mounted, and so on and so on, forming a hierarchy of cascading agents.

By connecting to the root of an agent hierarchy, managers can have a single access point to many resources and services. All MBeans in the hierarchy are manageable through the top master agent, and a manager does not need to worry about their physical location. Like the other services, the cascading service is implemented as an MBean that can be managed dynamically. This enables the manager to control the structure of the agent hierarchy, adding and removing subagents as necessary.

In particular, the cascading service MBean can work with any protocol connector, including any custom implementation. Those supplied by the Java DMK give you the choice of using remote method invocation (RMI), or the JMX messaging protocol (JMXMP). The legacy connector protocols implemented by previous versions of Java DMK can also be used, as long as they have been wrapped for use with the JMX Remote API, as described in “9.5 Wrapping Legacy Connectors” on page 133.

The cascading service also lets you specify a filter for selecting precisely the source MBeans that are cascaded in the target MBean server. This mechanism lets you limit the number of MBeans that are cascaded in the top agent of a large cascading hierarchy. For a general overview of the cascading service, see the *Java Dynamic Management Kit 5.1 Getting Started Guide*.

The code samples in this topic are from the files in the `current/Cascading` directory located in the main *examplesDir* (see *Directories and Classpath* in the preface).

This chapter contains the following topics:

- "14.1 `CascadingService` MBean" on page 234 describes the main component of the cascading service.
- "14.2 Cascaded MBeans in the Master Agent" on page 238 gives more details about what you can and cannot do through the cascading service.
- "14.3 Running the Cascading Example" on page 242 lets you interact with cascading agents through their HTML protocol adaptors.

14.1 `CascadingService` MBean

You should create a maximum of one `CascadingService` MBean in any target MBean server. The same `CascadingService` MBean can be used to mount multiple source MBean servers, that are located in multiple subagents in the Master Agent's target MBean server. The `CascadingService` creates one `ProxyCascadingAgent` behind the scenes for every mount operation it performs. The creation of that `ProxyCascadingAgent` is entirely hidden from the user. The complexity of the `ProxyCascadingAgents` is thus hidden by the `CascadingService` MBean. You should not attempt to use `CascadingAgents` directly, as was the case in earlier versions of Java DMK. Applications should always use the `CascadingServiceMBean` instead. The `CascadingServiceMBean` handles connections to the subagents and cascades all of that agent's registered MBeans into the master agent's target MBean server. No other classes are required or need to be generated to represent the MBeans.

The agent whose MBean server contains an active cascading service is called a *master agent* in relation to the subagent that is cascaded. The MBean server in which the MBeans are cascaded is called the *target MBean server*. An agent to which the cascading service is connected is called a *subagent* in relation to its master agent. Its MBean server is named the *source MBean server*. The source MBean server is the MBean server in which the MBeans actually reside. The `CascadingService` MBean creates *target MBeans* in the target MBean server to represent the subagent's *source MBeans*. For each source MBean, a `CascadingProxy` is registered behind the scenes in the target MBean server. See "14.2 Cascaded MBeans in the Master Agent" on page 238 for a description of these objects. The operation that makes it possible to cascade a source MBean server located in a subagent to the target MBean server in the Master Agent is called a *mount operation*, because of its similarity to a file system mount operation.

A master agent can have any number of subagents, each controlled individually by the same `CascadingService` MBean. Each mount operation is in fact implemented behind the scenes through a different `CascadingAgent` object created by the `CascadingService` MBean. The complexity of the `CascadingAgent` API is however completely abstracted by the `CascadingServiceMBean`. A subagent can itself contain cascading agents and cascaded MBeans, mounted from another layer of subagents, all of which will appear to be cascaded again in the master agent. This effectively enables cascading hierarchies of arbitrary depth and width.

Two master agents can also connect to the same subagent. This is similar to the situation where two managers connect to the same agent and can access the same MBean. If the implementation of a management solution permits such a condition, it is the designer's responsibility to handle any synchronization issues in the MBean.

The connection between two agents resembles the connection between a manager and an agent. The cascading service MBean relies on a connector client, and the subagent must have the corresponding connector server. The subagent's connector server must already be instantiated, be registered with its MBean server, and be ready to receive connections. The `CascadingServiceMBean` mount operation accepts a `JMXServiceURL` and a `Map` from which it obtains a `JMXConnector` from the `JMXConnectorFactory`. The mount operation will create the connection, and the *unmount* operation will close it.

EXAMPLE 14–1 Mounting MBeans from a Subagent

```
[...]

private void mountSubagents(JMXServiceURL[] agentURLs) {
         mountPoints = new String[agentURLs.length];
        for (int i=0;i<agentURLs.length;i++) {
             try {
               final String mountPointID =
                       cascadingService.mount(agentURLs[i],null,
                                         new ObjectName("ExportDomain:*"),
                                         "subagents/agent"+(i+1));

                     mountPoints[i] = mountPointID;
                  echo(mountPointID);
              } catch (Exception x) {
                   echo("\t!!! Could not mount subagent#"+(i+1)+" at " +
                        agentURLs[i] + "\n\tError is: " + x);
                   x.printStackTrace();
                   echo("\nEXITING...\n");
                   System.exit(1);
              }
        }
}
```

EXAMPLE 14–1 Mounting MBeans from a Subagent *(Continued)*

```
    [...]
```

In Example 14–1, source MBean servers located in subagents are mounted using a defined `JMXServiceURL`, to cascade remote MBeans from the domain `ExportedDomain`. By default, all MBeans of the subagent are cascaded in the master agent, but in this example we provide an object name pattern so as only to select those in `ExportedDomain`. Example 14–1 is taken from the example class `MasterAgent`, in the directory *examplesDir*`/current/Cascading`. To emulate file system mount operations, a different target path for each mounted subagent is used. In this example, we use `subagents/agent#x`, where `#x` starts at `1`, `x` is incremented for each subagent, and preserves the order of the JMX Service URLs that are given as input.

Note – You should use a different target path for every mount operation. The Java DMK cascading service implementation does not enforce this rule, but applications that are concerned with naming coherency should not break it.

EXAMPLE 14–2 Creating the Cascading Service

```
[...]

    private void createCascadingService() {

        printline();
        echo("Creating the CascadingService" +
              " MBean within the MBeanServer:");
        try {
                CascadingService service  = new CascadingService();
             ObjectInstance   cascadingInstance =
                  server.registerMBean(service, null);
             echo("\tCLASS NAME  = " + cascadingInstance.getClassName());
             echo("\tOBJECT NAME = " + cascadingInstance.getObjectName());
                 cascadingService = service;
          } catch (Exception e) {
             echo("\t!!! Could not create the " +
                  CascadingService.class.getName() + " MBean !!!");
             e.printStackTrace();
             echo("\nEXITING...\n");
             System.exit(1);
          }
    }

[...]
```

As shown in Example 14–2, before the subagent's source MBean servers can be mounted, the `CascadingService` MBean must be created and registered in, or tied to, the master agent's target MBean server. The `CascadingService` MBean in this example is registered in the master agent's target MBean server with a null object name, so that it is registered with its default `ObjectName`, that is defined in the `CascadingServiceMBean` interface.

EXAMPLE 14–3 Unmounting MBeans from a Subagent

```
[...]

private void unmountAll() {
      println();

          echo("Unregistering CascadingServiceMBean");
       try {
              server.unregisterMBean(CascadingServiceMBean.
                                         CASCADING_SERVICE_DEFAULT_NAME);
          } catch (Exception x) {
              echo("\t!!! Could not unregister " + CascadingServiceMBean.
                    CASCADING_SERVICE_DEFAULT_NAME + "\n\tError is: " + x);
              x.printStackTrace();
              echo("\nEXITING...\n");
              System.exit(1);
       }

           echo("Unmounting all mount points");

       final String mounts[] = cascadingService.getMountPointIDs();
           for (int i=0;i<mounts.length;i++) {
                try {
                      if (cascadingService.unmount(mounts[i]))
                          echo("unmounted "+mounts[i]);
                } catch (Exception x) {
                      echo("\t!!! Could not unmount " + mounts[i] +
                           "\n\tError is: " + x);
                      x.printStackTrace();
                      echo("\nEXITING...\n");
                      System.exit(1);
                }
           }
  }

[...]
```

When the master agent is stopped, it stops the cascading service by unmounting all the subagents that are currently mounted. This is done by first unregistering the `CascadingServiceMBean` from its target MBean server, to ensure that nobody can create new mount points while the removal is ongoing. Then, for each currently mounted mount point, the `unmount` operation is called, so that all proxies for the associated cascaded MBeans are unregistered from the target MBean server. The MBean server delegate in the master agent sends an unregistration notification for each cascaded MBean as it is removed.

14.2 Cascaded MBeans in the Master Agent

Once the the cascading service has mounted a source MBean server in the target MBean server, you can interact directly through the target MBean server with the target MBeans representing the subagent's MBeans. You can access and manage these MBeans as if you were connected to the subagent and accessing or managing the original MBeans. The target MBeans that you interact with are in fact `CascadingProxy` MBean objects registered in the master agent's target MBean server with the same object name as the source MBean, but with a domain that is prefixed by the target path used in the mount operation.

All management operations that you can perform on the original MBean can be performed identically through the target MBeanServer, using that target's object name. You can modify attributes, invoke operations and add or remove listeners, all with exactly the same result as if the manager were connected to the subagent when performing the action.

The behavior of a target `CascadingProxy` MBean is to transmit the action transparently to the subagent's MBean and return with an answer or result. The actual computation is performed by the original MBean running in its own source MBean server.

In Example 14–4, there is a timer MBean that was created in the subagent. Once the subagent has been mounted in the target MBean server, the timer is operated through its local target `CascadingProxy` MBean. It is not possible to have the direct reference to a source MBean, but it is possible to obtain a local proxy using the `MBeanServerInvocationHandler` class, as if it were a local MBean registered in the target MBean server. The fact that the operations are actually routed to a subagent through a `JMXConnector` remains hidden.

EXAMPLE 14–4 Managing Cascaded MBeans

```
private void doCascadingOperations() {

    echo("\nPress Enter to continue...\n");
    waitForEnterPressed();
    println();
    try {
         if (! cascadingService.isMounted(mountPoints[0])) {
                echo(mountPoints[0] + ": \n\t" + "not mounted! ");
                echo("Cannot do cascading operations.");
                return;
            }

         ObjectName timerName =
            new ObjectName("subagents/agent1/ExportDomain:type=Timer");
```

```
             echo(">>> Get Timer MBean \""+timerName+"\"");
         TimerMBean timer = (TimerMBean) MBeanServerInvocationHandler.
           newProxyInstance(server, timerName, TimerMBean.class, true);

        echo("\n>>> Ask the Timer MBean to send a " +
             "Timer Notification every 3 seconds");
        Date currentDate = new Date();

        timer.addNotification("Timer","Message",null,
                              new Date(currentDate.getTime() +
                              2),3000);

        timer.start();

        echo("\n>>> Add listener to the Timer MBean");
        server.addNotificationListener(timerName, this, null, null);
        echo("\tListener added successfully");
        echo("\nPress Enter to remove the listener from the " +
                 "Timer MBean...");
        waitForEnterPressed();

              if (cascadingService.isMounted(mountPoints[0])) {
                 try {
                server.removeNotificationListener(timerName,
                this);
                 } catch (Exception x) {
                echo("Unexpected exception while unregistering
                     listener:");
                     echo("\tError is: " + x);
                 }
            } else {
             echo(mountPoints[0] + ": \n\t" + "already
                  unmounted! ");
            }

        } catch (Exception e) {

        e.printStackTrace();
        System.exit(1);
    }
}
```

Example 14–4 obtains a `TimerMBean` proxy for the `Timer` MBean from the subagent `#1 ExportedDomain`, and then performs operations on it. In particular, it registers for timer notifications and starts the timer.

To the managing application, the target MBean in the master agent target MBean server *is* the MBean. Unregistering a cascaded MBean in the master agent will not unregister the cascaded MBean in the subagent, but will only unregister the `CascadingProxy` from the target MBean server. You should not attempt to unregister

MBeans mounted from a source MBean server through the target MBean server. If you want to create or unregister MBeans in a subagent, you should not do so through the CascadingService, but should use a JMXConnector directly connected to the subagent.

The cascading is almost totally transparent. A manager has no direct way of knowing whether an object is a cascaded object or not. It can attempt to guess the topology of the cascading hierarchy by examining the domain path of the object names, provided that the subagents have been mounted using a meaningful and coherent target path, but the underlying connectivity is completely transparent. The managing application can however be informed of failed or closed mount points by registering for notifications with the CascadingServiceMBean.

Likewise, some operations might fail due to the fact that the CascadingProxy object registered in the target MBean server is not the real source MBean. For example, unregistering the CascadingProxy will not cause the original source MBean to be unregistered. The CascadingProxy might reappear later if an external event makes the CascadingService update its view of the subagents. Similarly, trying to call addNotificationListener(ObjectName,ObjectName,...) with the object name of a target CascadingProxy as listener (namely the second parameter) will fail, because this operation cannot be routed to the source MBean.

14.2.1 Class of a Cascaded MBean

Proxy MBeans registered in the target MBeanServers when mounting source MBean servers are implemented as instances of the CascadingProxy class. The CascadingProxy MBean is locally registered in the target MBean server, and has the same MBeanInfo as its source MBean. In particular, if the source MBean is a model MBean, the MBeanInfo exposed by its CascadingProxy is ModelMBeanInfo. The exposed MBean information also contains the class name of the original MBean, and not the class name of the CascadingProxy. Exposing this borrowed MBeanInfo guarantees that the cascading service is as transparent as possible.

The symmetry of the Java dynamic management architecture means that this cascading mechanism is scalable to any number of levels. The cascaded object of a cascaded object is again an instance of the CascadingProxy class, and it borrows the same MBeanInfo. Any operation on the top target object is propagated to its source subagent, where the intermediate cascaded object will send it to its own source subagent, and so forth. The cost of cascading is the cost of accessing the subagents. The depth of your cascading hierarchy should be adapted to your management solution.

Because the cascading service MBean instantiates and controls all cascaded MBeans, the CascadingProxy class should never be instantiated through a management operation, nor by the code of the agent application. It is described here only to provide a better understanding of the internal behavior of the cascading service.

14.2.2 Cascading Issues

In this section, we explain some of the design issues that are determined by the implementation of the cascading service.

14.2.2.1 Dynamic Cascading

When an MBean is unregistered from the subagent, the cascading service automatically removes its corresponding target `CascadingProxy` MBean from the master agent's target MBean server. When a new MBean is registered in the subagent's source MBean server, the cascading service registers a new `CascadingProxy` mirroring this source MBean with the master agent's MBean server. Note that if an `ObjectName` pattern was provided when performing the mount operation, only those source MBeans whose source `ObjectName` satisfy that `ObjectName` pattern are considered.

Both these mechanisms scale to cascading hierarchies. Adding or removing an MBean in the subagent will trigger a notification that any cascading service connected to the subagent will receive. This will start a chain reaction up to the top of the hierarchy. Removing an MBean from the middle of a hierarchy also triggers a similar reaction up to the top target MBean, which is finally removed. However, as explained in the previous section, corresponding source MBeans that are lower in the hierarchy will not be affected. The cascading service reflects the content of a source MBean server into a target MBean server, but it cannot be used to create or remove MBeans in the source MBean server. Calls to `createMBean`, `registerMBean`, and `unregisterMBean` only affect the local target MBean server.

14.2.2.2 MBean Patterns and Filtering

When the cascading service MBean is instantiated, you can pass it an object name pattern. This object name pattern is applied to the list of MBeans in the subagent's source MBean server to determine those MBeans that are to be cascaded into the master agent's target MBean server. The filtering is activated for the life of the of the mount operation, namely, until unmount is called for that mount point. Care must be taken when using the `ObjectName` pattern, so as not to implement cascading hierarchies that are not analogous to File System mount operations.

The object name pattern is used to filter any new MBean that is registered in the source MBean server. If the new MBean meets the filter criteria, it will become visible and be cascaded into the target MBean server.

14.2.2.3 Using Target Paths

Although the API also allows you to implement different cascading schemes, your applications should only implement those schemes that can be compared to a regular File System mount, as follows.

- When calling the `CascadingServiceMBean.mount` operation, always use a non null `targetPath`. The target path can be assimilated to a target mount point in the File System analogy.
- Never use a `targetPath` under which MBeans are already registered in the target MBean server. Using such a target path could cause a name conflict when mounting the source MBeans to the target MBean server.
- Never give the same `targetPath` to two different mount operations. Like in the file system analogy, you should not attempt to mount two sources to the same target path.

The present implementation does not enforce those rules, but applications that are concerned with naming consistency and coherency should make sure to respect them.

14.3 Running the Cascading Example

The *examplesDir*`/current/Cascading` directory contains all of the files for the master agent and subagent applications, along with a simple MBean.

▼ To Run the Cascading Example

1. **Compile all files in this directory with the `javac` command.**

```
$ cd examplesDir/current/Cascading/
$ javac -classpath classpath *.java
```

2. **Start the subagent in another terminal window with the following command. Be sure that the classes for the `SimpleStandard` MBean can be found in its** ***classpath*****.**

```
$ java -classpath classpath -Durl="subagent_input_url" SubAgent
```

Here, *subagent_url* is the URL that is passed to the `JMXConnectorServerFactory` to create the subagent's `JMXConnectorServer`. You can use one of the following URLs:

- `service:jmx:jmxmp://`
- `service:jmx:rmi://`
- `service:jmx:iiop://`

Alternatively, you can specify your own URL, for example.

```
service:jmx:jmxmp://host_name:port_number
```

Note that you can start many subagents in different terminal windows, for example.

```
$ java -Dhtml.port=8082 -Durl="subagent1_input_url" SubAgent
$ java -Dhtml.port=8182 -Durl="subagent2_input_url" SubAgent
...
```

However you start the subagent, it will inform you onscreen of the URL, *subagent-actual-url*, at which it can be found.

3. **Wait for the agent to be completely initialized, then start the master agent with the following command:**

   ```
   $ java -classpath classpath MasterAgent subagent-actual-url
   ```

   In the command above, *subagent-actual-url* is the URL you were given when you launched the subagent. If you started more than one subagent, you can start the `MasterAgent` as shown in the following command.

   ```
   $ java -classpath classpath MasterAgent subagent1-actual-url
   subagent2-actual-ur ...
   ```

   When started, the master agent application first creates the `CascadingService` MBean and then mounts the subagent's `ExportDomain` in its target MBean server. The master agent then performs operations on the cascaded MBeans of the subagent. Press `Enter` to step through the example when the application pauses.

4. **You can interact with the example through the HTML adaptor of the master agent and subagent.**

   This is demonstrated in "How to Interact with a Cascade Hierarchy" on page 243.

5. **If you are still receiving timer notifications on the master agent, press `Enter` once more to remove the listener, but leave both agent applications running.**

▼ How to Interact with a Cascade Hierarchy

1. **Open two browser windows side by side and load the following URLs:**

Subagent	`http://`*subAgent_hostname*`:8082/`
Master Agent	`http://`*MasterAgent_hostname*`:8084/`

   In the subagent, you should see the timer MBean in the `ExportDomain` and a `SimpleStandard` MBean in the `DefaultDomain`.

   The master agent is recognizable by the cascading service MBean in `com.sun.jdmk`. Otherwise it has identical timer MBeans registered in the `subagent/`*agent#*`/ExportDomain`: this is the cascaded version of the timer in the subagent. The `SimpleStandard` MBean is not cascaded because our cascading service instance filters with the following object name pattern:

   ```
   ExportDomain:*
   ```

2. **Create four MBeans of the `SimpleStandard` class in following order:**

On the Master Agent:	`ExportDomain:name=SimpleStandard,number=1`
On the Subagent:	`ExportDomain:name=SimpleStandard,number=1`
	`ExportDomain:name=SimpleStandard,number=2`
	`ExportDomain:name=SimpleStandard,number=3`

3. **Reload the agent view on the master agent.**

   The cascaded MBeans for the last two agents have been created automatically. Look at the MBean view of either of these cascaded MBeans on the master agent. Their class name appears as `SimpleStandard`.

4. **In the master agent, set a new value for the `State` string attribute of all 3 of its `SimpleStandard` MBeans.**

   When you look at the corresponding MBeans in the subagent, you see that all the MBeans were updated through the master agent.

5. **In the subagent, invoke the `reset` operation of all 3 of its `SimpleStandard` MBeans.**

   When you inspect the MBeans in the master agent, the values for all were reset. Remember that the HTML adaptor must get the values of attributes for displaying them, so they were correctly retrieved from the cascaded MBeans that we reset.

6. **In the subagent, unregister MBeans `number=1` and `number=3`, then update the agent view on the master agent.**

   The cascaded MBean is automatically removed by the cascading service.

7. **Invoke the `stop` operation of the `CascadingService` MBean in the master agent.**

   The last cascaded MBean for the timer is removed from the master agent. The two agents are no longer connected.

8. **If you have finished with the agents, press `Enter` in both of their terminal windows to exit the applications.**

CHAPTER 15

Discovery Service

The discovery service enables you to discover Java dynamic management agents in a network. This service relies on a discovery client object that sends out multicast requests to find agents. In order to be discovered, an agent must have a registered discovery responder in its MBean server. Applications can also use a discovery monitor that detects when discovery responders are started or stopped.

The combination of these functions enables interested applications to establish a list of active agents and keep it up to date. In addition to knowing about the existence of an agent, the discovery service provides the version information from an MBean server's delegate and the list of communication MBeans that are currently registered. The discovery service uses the term *communicators* to designate a set of MBeans consisting of protocol adaptors and connector server objects.

Often, the discovery client and the discovery monitor are located in a manager application that wants to know about the available agents. However, agent applications are free to use the discovery service because they might require such information for cascading (see "Cascading Agents") or for any other reason. To simplify using the discovery service in an agent, all of its components are implemented as MBeans.

The code samples in this topic are found in the `current/Discovery` directory located in the main *examplesDir* (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "15.1 Active Discovery" on page 246 covers how the discovery client initiates a search for discovery responders.
- "15.2 Passive Discovery" on page 253 explains the roles of the discovery responder and discovery monitor.
- "15.3 Running the Discovery Example" on page 259 demonstrates both active and passive discovery.

15.1 Active Discovery

In active discovery, the discovery client initiates searches for agents on the network. It involves a discovery client that sends the discovery request and a discovery responder in each agent that responds. Each instance of the responder supplies the return information about the agent in which it is registered. The return information is represented in the discovery client by a vector of discovery response objects.

The application containing the discovery client can initiate a search at any time. For example, it might do a search when it is first started and periodically search again for information about the communicators that might have changed. For each search, the discovery client broadcasts a request and waits for the return information from any responders.

In the following sections, we describe each of these objects in further detail.

15.1.1 Discovery Client

The `DiscoveryClient` class provides methods to discover agents. The active discovery operation sends a discovery request to a multicast group and waits for responses. These messages are proprietary and are not exposed to the user. Discovery clients can only discover agents listening on the same multicast group and port, so your design must coordinate this information between the discovery client and responders.

You can instantiate and perform searches from multiple discovery clients in a single application. Each discovery client can be configured to use different multicast groups or ports, enabling you to discover different groups of agents.

The discovery services enable users to specify a local interface from which to send out multicast messages. This is useful when working on a system that has multihome interfaces connecting to disconnected multinetworks. In addition, the `DiscoveryResponder` constructor enables you to specify the host address to be used to build the discovery response. The address can be specified either as an IPV4 or IPv6 address, or as a host name.

EXAMPLE 15–1 Instantiating and Initializing a Discovery Client

```
public class Client {
    public static void main(String[] args) throws Exception {
           BufferedReader br
               = new BufferedReader(new InputStreamReader(System.in));

           discoveryClient = new DiscoveryClient();
           discoveryClient.start();
```

EXAMPLE 15–1 Instantiating and Initializing a Discovery Client *(Continued)*

```
       while(true) {
          echo("\n>>> Press return to discover connector servers;");
          echo(">>> Type a host name then return to discover connector
            servers running on that machine;");
          echo(">>> Type bye then return to stop the client.");

        String s = br.readLine();

          if (s.equalsIgnoreCase("bye")) {
               System.exit(0);
          } else {
               try {
                    discover(s);
               } catch (Exception e) {
                    echo("Got exception "+e);
                    e.printStackTrace();
               }
          }
       }
}
```

Once you have created the discovery client, before initiating searches, you must call the discovery client's `start` method, as shown in Example 15–1. This will create its multicast socket and join the multicast group used for broadcasting its discovery request. The default multicast group is `224.224.224.224` and the default port is `9000`. These can be set to other values through the `multicastGroup` and `multicastPort` attributes, but only when the `state` of the discovery client is `OFFLINE`.

The scope of the discovery request depends on the time-to-live used by the multicast socket. Time-to-live is defined by the Java class `java.net.MulticastSocket` to be a number between 1 and 255. By default, the time-to-live is 1, which corresponds to the host's local area network. You can modify this value at any time by setting the discovery client's `TimeToLive` attribute.

By default, a discovery client waits for responses for one second after it has sent a discovery request. This period can be customized by setting a value in milliseconds for its `TimeOut` attribute. When setting this attribute, you should take into account estimated time for a round-trip of a network packet using the given time-to-live.

Once started, the discovery client in the example above awaits user inputs before starting searches.

15.1.2 Performing a Discovery Operation

An application triggers a search operation by invoking the findMBeanServers or findCommunicators methods on an active DiscoveryClient object. Using the current settings, it will send the multicast request and block for the timeout period. At the end of the timeout period, these methods return the responses that were received.

Both methods return a vector of DiscoveryResponse objects. This class exposes methods for retrieving information about the MBean server and the registered communicator MBeans in the agent. The MBean server information is the same as that exposed by that agent's MBean server delegate. The communicators are identified by ConnectorAddress objects and indexed by object name in a hash table.

Both search methods return the information about the agent's MBean server. The hash table of communicator MBeans is always empty for discovery responses returned by the findMBeanServers method. Otherwise, you can extract object names and protocol information from the hash table. One way of distinguishing the communicator MBeans is to rely on the default names provided by the ServiceName class.

Note – All discovery messages sent between components of the discovery service are compatible between applications running different versions of the Java platform or between versions 5.0 and 5.1 of the Java DMK. However, these different configurations are not compatible for subsequent management operations through connectors. You can use the getImplementationVersion method of the DiscoveryResponse object to determine both the Java platform and product version numbers.

In our example, we request all information about the agents and print out all information in the discovery responses.

EXAMPLE 15–2 Performing a Discovery Operation

```
private static void discover(String host) throws Exception {
       Vector v = null;
       if (host ==  null || host.equals("")) {
          v = discoveryClient.findCommunicators();
       } else {
          v= discoveryClient.findCommunicators(host);
       }

       if (v.size() == 0) {
          echo("No connector server has been found.");

          return;
       }

       for (int i=0; i<v.size(); i++) {
            DiscoveryResponse dr = (DiscoveryResponse)v.get(i);
            JMXServiceURL url = null;
```

EXAMPLE 15–2 Performing a Discovery Operation *(Continued)*

```
            // legacy servers
            Collection c = dr.getObjectList().values();
            for (Iterator iter=c.iterator(); iter.hasNext();) {
                 Object o = iter.next();

                 if (!(o instanceof ConnectorAddress)) {
                     continue;
                 }

                 ConnectorAddress ca = (ConnectorAddress)o;
                 if (ca.getConnectorType().equals("SUN RMI")) {
                     url = new JMXServiceURL("jdmk-rmi",
                                                ((RmiConnectorAddress)ca).getHost(),
                                                ((RmiConnectorAddress)ca).getPort());
                // Repeat for jdmk-http and jdmk-https connectors
           [...]
             } else {
                    echo("Got an unknown protocol: "+ca.getConnectorType());

                    continue;
                }

                echo("\nFound a legacy server which is registered
                  as a legacy MBean: "
                      +url.getProtocol()+"
                 "+url.getHost()+"
                 "+url.getPort());
                echo("Connecting to that server.");
                JMXConnector jc = JMXConnectorFactory.connect(url);
                echo("Its default domain is
                 "+jc.getMBeanServerConnection().getDefaultDomain());
                echo("Closing the connection to that server.");
                jc.close();
           }

           // JMX-remote servers
           JMXServiceURL[] urls = dr.getServerAddresses();

           echo("");
            for (int ii=0; ii<urls.length; ii++) {
              echo("\nFound a server which is registered
                as a JMXConnectorServerMBean: "
                    +urls[ii]);
              echo("Connecting to that server.");
              JMXConnector jc = JMXConnectorFactory.connect(urls[ii]);
              echo("Its default domain is
              "+jc.getMBeanServerConnection().getDefaultDomain());
              echo("Closing the connection to that server.");
              jc.close();
           }
        }
}
```

On the agent side, the discovery responder automatically replies to discovery requests. Any active, registered responder in the same multicast group that is reached within the given time-to-live of the request will respond. It will automatically gather the requested information about its MBean server and send the response. The settings of the responder do not affect its automatic reply to discovery requests. In "15.2.1 Discovery Responder" on page 255 we will cover how its settings control passive discovery.

The discovery client can search for agents via all the connector protocols supported by Java DMK, both current and legacy. Java DMK 5.1 introduces a new `DiscoveryResponse` method, `getServerAddresses`, which is used to get the addresses of any servers registered in an MBean server as a `JMXConnectorServerMBean`. The `getServerAddresses` method can discover servers that are either instances of the JMX Remote API `JMXConnectorServer`, or legacy servers wrapped to appear as such. This new method ensures compatibility between the following pairs of discovery clients and servers created using versions 5.0 and 5.1 of Java DMK.

- **A Java DMK 5.0 discovery client and a Java DMK 5.1 discovery server**:

  A 5.0 client can discover a 5.1 connector server, but only if it is a legacy server registered as a legacy MBean.
- **A Java DMK 5.1 discovery client and a Java DMK 5.0 discovery server**:

  The method `DiscoveryResponse.getObjectlist()` will return a hashtable of 5.0 connector servers.

  The method `DiscoveryResponse.getServerAddresses()` will return an empty table.
- **A Java DMK 5.1 discovery client and a Java DMK 5.1 discovery server**:

  The method `DiscoveryResponse.getServerAddresses()` will return a table of the addresses of all the servers that are registered as either legacy or JMX Remote API `JMXConnectorServerMBean` MBeans.

  The method `DiscoveryResponse.getObjects()` will return a vector of legacy servers that are registered as legacy MBeans.

All the above relations between versions are also true for the passive discovery monitor.

In active discovery, the discovery client controls all parameters of a search it initiates, including the response mode of the discovery responder. The discovery client determines whether responses are sent back on a different socket (unicast) or sent to the same multicast group. The default is unicast: if you want to use the multicast response mode, set the `PointToPointResponse` attribute to `false` before initiating the discovery.

15.1.2.1 Unicast Response Mode

When the `PointToPointResponse` boolean attribute is `true`, the discovery client specifies unicast mode in its discovery requests. The responder will create a datagram socket for sending the response only to the discovery client. As shown in the following diagram, each responder will send its response directly back to the discovery client. The datagram socket used by each responder is bound to its local host address; this cannot be customized.

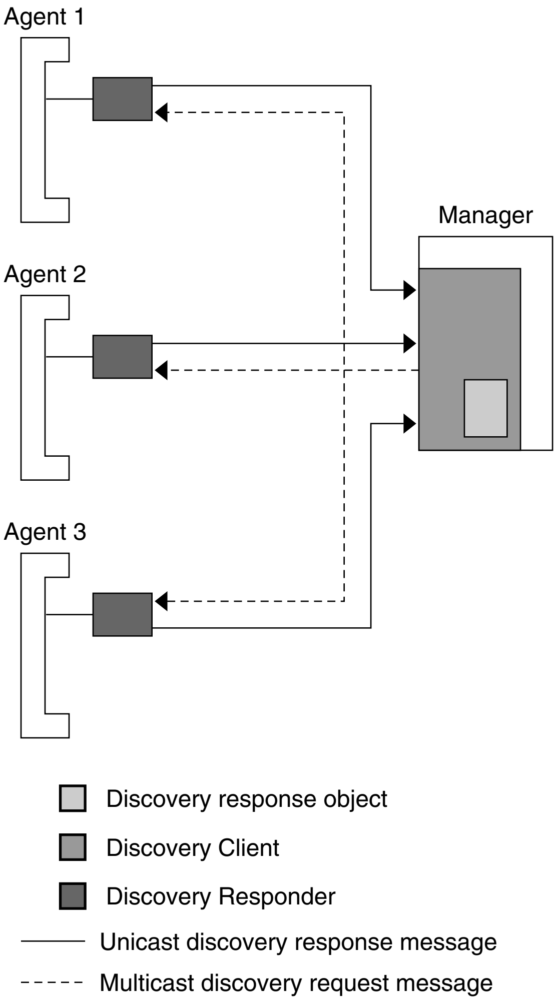

Agent 1
Manager
Agent 2
Agent 3
Discovery response object
Discovery Client
Discovery Responder
Unicast discovery response message
Multicast discovery request message

FIGURE 15–1 Unicast Response Mode

15.1.2.2 Multicast Response Mode

When the `PointToPointResponse` boolean attribute is `false`, the discovery client specifies multicast mode in its requests. The discovery responder will use the existing multicast socket to send response, broadcasting it to the same multicast group as the request. As shown in the following diagram, every member of the multicast group will receive the message, but only the discovery client can make use of its contents. Multicast mode avoids having to open another socket for the response, but all of the responses will create traffic in each application's socket.

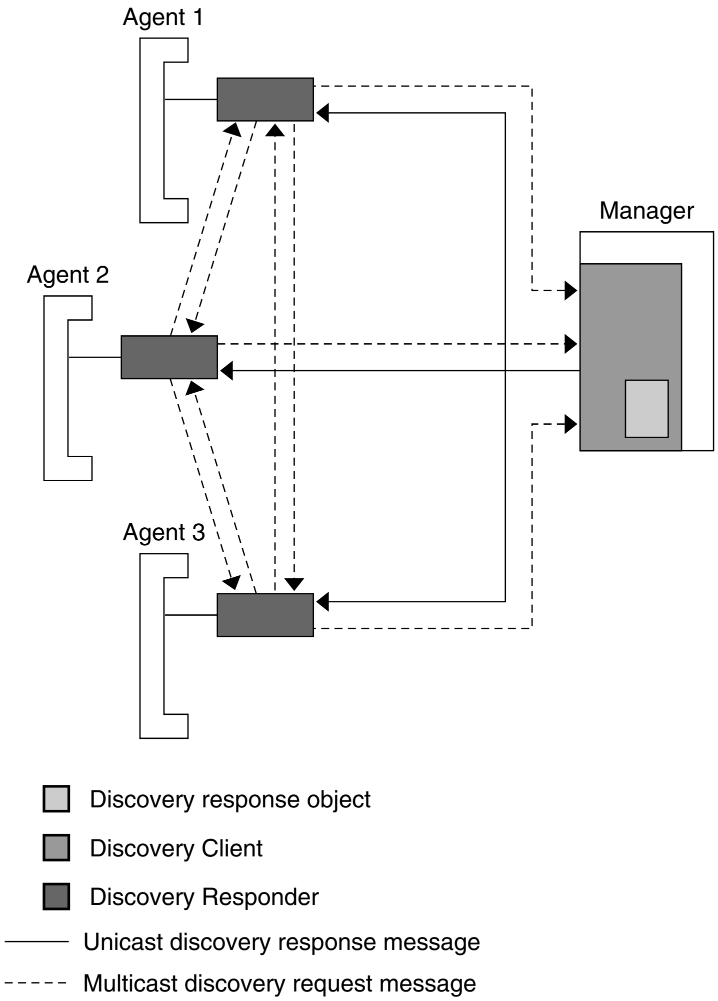

Agent 1
Manager
Agent 2
Agent 3
Discovery response object
Discovery Client
Discovery Responder
Unicast discovery response message
Multicast discovery request message

FIGURE 15–2 Multicast Response Mode

15.2 Passive Discovery

In passive discovery, the entity seeking knowledge about agents listens for the activation or deactivation of their discovery responders. When discovery responders are started or stopped, they send out a proprietary message that contains all discovery response information. The `DiscoveryMonitor` object waits to receive any of these messages from the multicast group.

A discovery monitor is often associated with a discovery client. By relying on the information from both, you can keep an up-to-date list of all agents in a given multicast group.

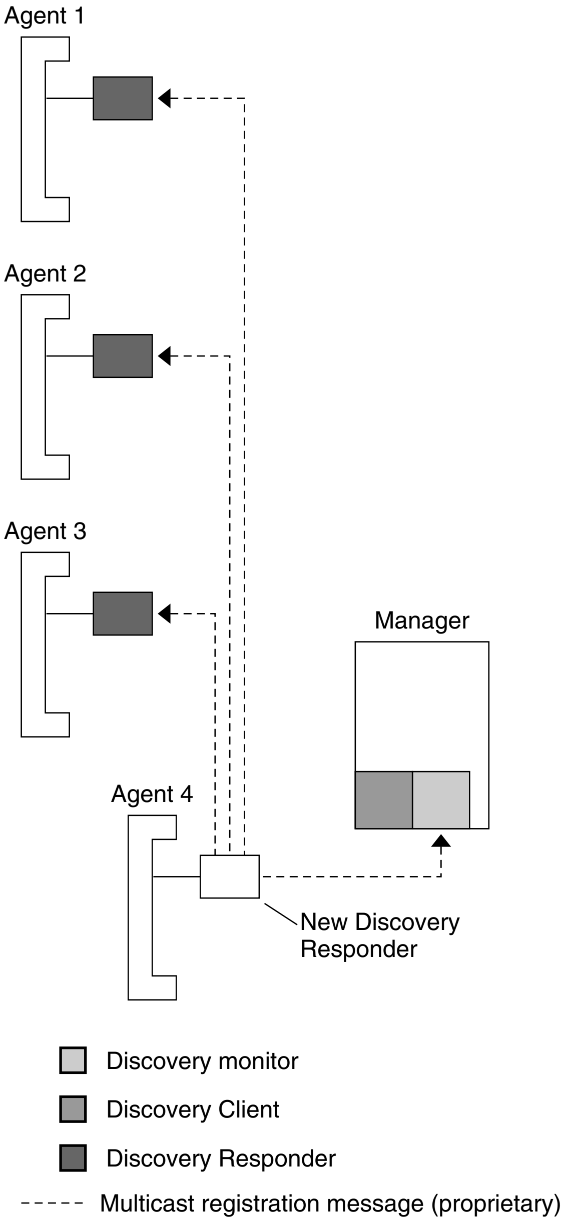


FIGURE 15–3 Passive Discovery of Discovery Responders

Therefore, configuring and starting the discovery responder is an important step to the overall discovery strategy of your applications.

15.2.1 Discovery Responder

The agents that are configured to be discovered must have an active DiscoveryResponder registered in their MBean server. The responder plays a role in both active and passive discovery:

- When the start operation of a discovery responder is invoked, it sends out a multicast message indicating that it has been activated.
- When the discovery responder is active, it automatically responds to discovery requests.
- When the responder's MBean is unregistered or its stop operation is invoked, it sends out a multicast message to indicate that it will be deactivated.

Both types of messages are proprietary and their contents are not exposed to the user. These messages contain information about the MBean server, its delegate's information and a list of communicator MBeans, unless not requested by the discovery client.

In our example we create the discovery responder in the MBean server and then activate it. Then, we create different connector servers that will discover the agent passively, due to its active discovery responder.

EXAMPLE 15–3 Creating a Discovery Responder

```
public class Responder {
    public static void main(String[] args) {
          try {
              MBeanServer myMBeanServer =
                MBeanServerFactory.createMBeanServer();

              echo("\nCreate and register a DiscoveryResponder MBean.");
              ObjectName dc =
               new ObjectName("DiscoveryExample:name=DiscoveryResponder");
          myMBeanServer.createMBean("com.sun.jdmk.discovery.DiscoveryResponder",
                                      dc);
              myMBeanServer.invoke(dc, "start", null, null);

              // Create an HtmlAdaptorServer on the default port.
              [...]

          // Create JMX Remote API connector servers
                JMXServiceURL url;
              JMXConnectorServer server;
              ObjectName on;

              // rmi
              url = new JMXServiceURL("rmi", null, 0);
              server =
             JMXConnectorServerFactory.newJMXConnectorServer(url,
                                                              null,
                                                              myMBeanServer);
              server.start();
```

EXAMPLE 15–3 Creating a Discovery Responder *(Continued)*

```
              url = server.getAddress();

             on = new ObjectName("jmx-remote:protocol=rmi");
              myMBeanServer.registerMBean(server, on);

          // Create RMI/IIOP connector server
          [...]

          // Create JMXMP connector server
          [...]

          // stop/start the responder to send a notification
              myMBeanServer.invoke(dc, "stop", null, null);
           Thread.sleep(100);
              myMBeanServer.invoke(dc, "start", null, null);

          // Create wrapped legacy RMI and HTTP connector servers
          [...]

              // Repeat for the other current and legacy connector protocols
          [...]

              // stop/start the responder to allow a Monitor to find them
              myMBeanServer.invoke(dc, "stop", null, null);
           Thread.sleep(100);
              myMBeanServer.invoke(dc, "start", null, null);

           [...]

              echo("All servers have been registered.");

              echo("\n>>> Press return to exit.");
              System.in.read();
              System.exit(0);
          } catch (Exception e) {
              e.printStackTrace();
       }
}
```

The discovery responder has attributes for exposing a multicast group and a multicast port. These attributes define a multicast socket that the responder will use to receive discovery requests. It will also send activation and deactivation messages to this multicast group. When sending automatic responses to discovery requests, the time-to-live is provided by the discovery client. The responder's time-to-live attribute is *only* used when sending activation and deactivation messages.

We use the default settings of the discovery responder that are the multicast group 224.224.224.224 on port 9000 with time-to-live of 1. In order to modify these values, you need to set the corresponding attributes *before* starting the discovery responder

MBean. You can also specify them in the class constructor. If the responder is active, you will need to stop it before trying to set any of these attributes. In that way, it will send a deactivation message using the old values and then an activation message with the new values.

15.2.2 Discovery Monitor

The discovery monitor is a notification broadcaster: when it receives an activation or deactivation message from a discovery responder, it sends a discovery responder notification to its listeners. Once its parameters are configured and the monitor is activated, the discovery is completely passive. You can add or remove listeners at any time.

The `DiscoveryMonitor` MBean has multicast group and multicast port attributes that determine the multicast socket where it will receive responder messages. Like the other components of the discovery service, the default multicast group is 224.224.224.224 and the default port is 9000. You can specify other values for the group and port either in the constructor or through attribute setters when the monitor is off-line.

Example 15–4 shows a discovery monitor being started, and then a listener being added.

EXAMPLE 15–4 Instantiating and Starting a Discovery Monitor

```
public class Monitor {
    public static void main(String[] args) throws Exception {

       echo("Create a DiscoveryMonitor.");
          DiscoveryMonitor dm = new DiscoveryMonitor();
          dm.start();

          echo("Add a listener to receive monitor notifications.");
          dm.addNotificationListener(new MyListener(), null, null);

          echo("Waiting for new server notifications ...");

          echo("\nType any key to stop the monitor.");
          System.in.read();

          dm.stop();
          echo("\nThe monitor has been stopped.");

          System.exit(0);
    }
```

The discovery monitor must be activated with the `start` operation before it will receive responder messages and send notifications. If it is being used as an MBean, it will be stopped automatically if it is unregistered from its MBean server. If it is not used as an MBean, as is the case here, you should invoke its `stop` method before your application exits.

15.2.3 Discovery Responder Notifications

When it receives a responder's activation or deactivation message, the discovery monitor sends notification objects of the `DiscoveryResponderNotification` class. This notification contains the new state of the discovery responder (`ONLINE` or `OFFLINE`) and a `DiscoveryResponse` object with information from the agent where the responder is located.

The listener could use this information to update a list of agents in the network. In our example, the listener is the agent application itself, and the handler method only prints out the information in the notification.

EXAMPLE 15–5 Discovery Responder Notification Handler

```
private static class MyListener implements NotificationListener {
       public void handleNotification(Notification notif, Object handback) {

          try {
               DiscoveryResponderNotification dn
                               = (DiscoveryResponderNotification)notif;
            DiscoveryResponse dr = (DiscoveryResponse)dn.getEventInfo()   ;

                 JMXServiceURL url = null;

                 // legacy servers
                 Collection c = dr.getObjectList().values();
                 for (Iterator iter=c.iterator(); iter.hasNext();) {
                      Object o = iter.next();

                      if (!(o instanceof ConnectorAddress)) {
                           continue;
                      }

                      ConnectorAddress ca = (ConnectorAddress)o;
                      if (ca.getConnectorType().equals("SUN RMI")) {
                           url = new JMXServiceURL("jdmk-rmi",
                             ((RmiConnectorAddress)ca).getHost(),
                             ((RmiConnectorAddress)ca).getPort());
                      // Repeat for jdmk-http and jdmk-https connectors
                      [...]

                  } else {
                                continue;
                      }
```

EXAMPLE 15–5 Discovery Responder Notification Handler *(Continued)*

```
              echo("\nFound a legacy server which is registered as
                   a legacy MBean: "
                   +url.getProtocol()+" "+url.getHost()+" "
                   +url.getPort());
                  echo("Connecting to that server.");
              JMXConnector jc = JMXConnectorFactory.connect(url);
                     jc.close();
             }

             // JMX Remote API servers
             JMXServiceURL[] urls = dr.getServerAddresses();

             echo("");
             for (int ii=0; ii<urls.length; ii++) {
                  echo("\nFound a server which is registered as a
                  JMXConnectorServerMBean: "
                        +urls[ii]);
                  echo("Connecting to that server.");
                  JMXConnector jc = JMXConnectorFactory.connect(urls[ii]);
                  echo("Its default domain is
                   "+jc.getMBeanServerConnection().getDefaultDomain());
                    jc.close();
             }
        } catch (Exception e) {
             echo("Got unexpected exception: "+e);
             e.printStackTrace();
        }
  }
}
```

15.3 Running the Discovery Example

The *examplesDir*/`current/Discovery` directory contains the source file for the application that demonstrates the discovery service.

▼ To Run the Discovery Example

1. **Compile the Java classes.**

   $ **javac -classpath** *classpath* ***.java**

2. **Start the `Client`.**

   $ **java -classpath** *classpath* **Client**

You will be prompted to press Enter to discover connector servers, to provide the host name of a machine where servers are running or to exit the discovery component. However, there are not yet any servers to discover, because the Responder has not yet been started.

3. **In a second terminal window, start the Monitor**

   ```
   $ java -classpath classpath Monitor
   ```

   You will see confirmation that the monitor has started, and that it is passively listening for notifications,

4. **In a third terminal window, start the Responder.**

   ```
   $ java -classpath classpath Responder
   ```

   You will see the creation of the different connector servers. Then you will be invited to create further connector servers by pressing Enter.

   In the window in which you started the Monitor, you will see the connector servers created by Responder being discovered.

5. **Press Enter in the terminal window in which you started the Client.**

   You will see the Client discovering the connector servers created by the Responder, and connecting to them to retrieve information about them, before closing the connections.

   You can continue discovering and connecting to connector servers for as long as the Responder is active.

6. **When you have finished, stop the Responder, the Monitor and the Client.**

PART V

SNMP Interoperability

From the outset, the Java Dynamic Management Kit (Java DMK) was designed to be compatible with existing management standards. The Simple Network Management Protocol (SNMP) is the most widely used of these, and the Java DMK provides the tools to integrate Java technology-based solutions into the SNMP world.

Using the SNMP interoperability with Java dynamic management solutions, you can develop agents that can be accessed through SNMP and through other protocols. You can also develop managers in the Java programming language that access both SNMP agents and Java dynamic management agents.

The agent and manager tools of the Java DMK are completely independent. SNMP agents developed with this toolkit can be accessed by any SNMP manager, and the SNMP manager API lets you connect to any SNMP agent. The sample applications in this part use the toolkit on both agent and manager sides, but this is only one possible configuration.

This part contains the following chapters:

- Chapter 16 describes how to create an *SNMP agent*. This chapter demonstrates how the SNMP protocol adaptor makes a Java dynamic management agent also act as an SNMP agent. The MBeans generated by the `mibgen` tool represent SNMP MIBs that can be accessed by any SNMP manager connecting to the SNMP adaptor. This lets you implement your MIB through Java code and take advantage of the agent services. The example applications also demonstrate how traps are sent through the SNMP protocol adaptor.
- Chapter 17 shows you how to use the *SNMP manager API* to develop an SNMP manager in the Java programming language. An SNMP manager handles Java objects representing peers, parameters, sessions, and requests to access SNMP agents and perform management operations. Two examples demonstrate synchronous and asynchronous manager applications, and a third example shows how managers can communicate through inform requests.

- Chapter 18 shows you how to implement SNMP tables, with differing levels of complexity. Straightforward tables that implement the RowStatus convention are demonstrated. These tables are then extended to add instrumentation to the tables. Finally, how to implement virtual tables is also explained.
- Chapter 19 groups all of the information about *creating secure SNMP agents and managers*. User-based security model (USM) acess control lists (ACL) provide authentication and privacy to requests. Custom packet encoding between managers and agents is also possible, letting you develop any level of communication security you need.
- Chapter 20 describes the *SNMP Master Agent* and provides examples showing how to make a subagent interact with a manager via three different types of SNMP master agent.

Note – The Java packaging of the SNMP classes for Java DMK 5.1 has changed. In Java DMK 5.0, the SNMP classes were included in the `SUNWjsnmp` package, and they required a separate Java archive (JAR) file, `jsnmpapi.jar`. In Java DMK 5.1, the SNMP classes are packaged in the `SUNWjdmk-runtime` package, and require the same `jdmkrt.jar` JAR file as the rest of the current Java DMK classes. This new arrangement avoids the issue of potentially conflicting versions of the `SUNWjsnmp` package encountered under Java DMK 5.0.

In addition, the SNMP API delivered with Java DMK 5.0 is now deprecated. The SNMP API in Java DMK 5.1 is effectively a completely new SNMP API, that introduces a more orthodox system of Java class naming.

To use existing SNMP implementations that you created using Java DMK 5.0 alongside SNMP implementations created using Java DMK 5.1, you must translate the class names of the 5.0 implementations into the new format. How to perform this translation is explained in the Release Notes.

To continue to use SNMP implementations you created using version 5.0 of Java DMK under version 5.1, a new JAR file called `legacysnmp.jar` is provided. You must add this new JAR to your classpath when running your Java DMK 5.0 SNMP implementations under Java DMK 5.1.

All the examples of SNMP code given in the `/examples/current/Snmp` directory have already been translated to implement the new class naming system.

CHAPTER 16

Creating an SNMP Agent

Using the Java Dynamic Management Kit (Java DMK), you can create an agent application that is both an SNMP agent and a normal Java Management Extensions (JMX) agent. SNMP MIBs can be represented as MBeans and the SNMP protocol adaptor exposes them to SNMP managers. Only MBeans derived from MIBs can be managed through the SNMP protocol adaptor, but other managers can view and access them through other protocols.

The `mibgen` tool provided generates the code for MBeans that represent a MIB. Groups and table entries are represented as MBean instances that expose the SNMP variables through their attributes. The `mibgen` tool creates skeleton getters and setters that only read or write internal variables. To expose the behavior of your host machine or device, you need to implement the code that accesses the host- or device-specific functionality.

There are two SNMP protocol adaptors in the Java DMK 5.1.

- The first SNMP protocol adaptor interacts with your customized MIB MBeans to implement a compatible SNMPv1 and SNMPv2c agent. It also provides the mechanisms for sending traps and implementing both community-based access control and message-level data security (see “Security Mechanisms in the SNMP Toolkit”).
- The second protocol adaptor implements SNMPv3 compatible agents, as well as SNMP agents of the two previous versions. It also handles the IP-based access control of SNMPv1/v2. This adaptor provides user-based security and user-based access control (see “Security Mechanisms in the SNMP Toolkit”).

The program listings in this tutorial show only functional code: comments and output statements have been modified or removed for space considerations. However, all management functionality has been retained for the various demonstrations. The complete source code is available in the `current/Snmp/Agent` example directory located in the main *examplesDir* (see “Directories and Classpath” in the Preface).

This chapter covers the following topics:

- "16.1 MIB Development Process" on page 264 explains how MIBs are implemented as MBeans
- "16.2 SNMP Protocol Adaptor" on page 266 shows how to build an SNMP agent with the components of the Java DMK
- "16.2.6 Managing the SNMP Adaptors" on page 272 provides the basic SNMPv3 security information you need for the examples in this chapter
- "16.3 Sending Traps" on page 275 demonstrates the SNMP trap mechanism in the SNMP adaptors
- "16.4 Standalone SNMP Agents" on page 283 demonstrates an alternative way of implementing an SNMP agent
- "16.5 Multiple Agents" on page 287 shows how to implement an agent instantiating more than one SNMPv3 adaptor

16.1 MIB Development Process

Here we describe the process for making MIBs manageable through the SNMP protocol adaptor of the Java DMK. In our example, we demonstrate this process on a subset of the MIB-II defined by RFC 1213.

Once you have defined the MIB you want to manage in your SNMP agent you need to generate its MBean representation using the `mibgen` tool. This tool generates MBeans that represent the whole MIB, each of its groups and nested groups, and each of its table entries. This command-line tool and its output are fully described in the *Java Dynamic Management Kit 5.1 Tools Reference Guide*.

The `mibgen` tool only generates the MBean structure to represent the MIB, it is up to the developer to implement the MIB functionality inside the generated classes. Our example gives only a simple implementation of the MIB-II for demonstration purposes. However, this shows you the way to extend the generated classes to provide your own implementation.

The `mibgen` tool handles all three SNMP protocols identically, and the MIBs implemented are entirely protocol neutral. Consequently, code generated by `mibgen` for previous versions works perfectly in the SNMPv3 framework.

16.1.1 Generating MIB MBeans

To run the `mibgen` tool for our example, go to the *examplesDir*`/current/Snmp/Agent` directory and type the following command:

```
$ mibgen -d . mib_II_subset.txt
```

This generates the following files in the current directory:

- The MBean (by inheritance) for the whole MIB: RFC1213_MIB.java
- The MBean and its helper class for the Snmp group: Snmp.java, SnmpMBean.java, SnmpMeta.java
- The MBean and its helper class for the System group: System.java, SystemMBean.java, SystemMeta.java
- The MBean and its helper class for the Interfaces group: Interfaces.java, InterfacesMBean.java, InterfacesMeta.java
- The class representing the Interfaces table, and the MBean representing entries in the table: TableIfTable.java, IfEntry.java, IfEntryMBean.java, IfEntryMeta.java
- Classes representing enumerated types used in these groups: EnumSnmpEnableAuthenTraps.java, EnumIfOperStatus.java, EnumIfAdminStatus.java, EnumIfType.java
- The OID table for SNMP managers wanting to access this MIB: RFC1213_MIBOidTable.java

The MBean with the name of the MIB is a central administrative class for managing the other MBeans that represent the MIB groups and table entries. All of the other MBeans contain the SNMP variables as attributes of their management interface. The mibgen tool generates standard MBeans for the MIB, so attributes are implemented with individual getter and setter methods.

These MBeans are just skeletons, meaning that the attribute implementations only return a default value. You must implement the getters and setters of the attributes to read and write data with the correct semantic meaning of the SNMP variable.

Because SNMP does not support actions in MIBs, the only operations in these MBeans are *checkers* associated with the SNMP “Set” request in writeable variables. Again, these are skeleton methods that you must implement to do the checks that you require before the corresponding “Set” operation. You can add operations and expose them in the MBean interface, but the SNMP manager cannot access them. However, other managers can call these operations if they are connected through another protocol.

16.1.2 Implementing the MIB

Our example only implements a fraction of the attributes, those that are used in this tutorial. The others are simply initialized with a plausible value. Using DEFVAL statements in our MIB, we could force mibgen to generate MBeans with user-defined default values for attributes. As this is not done in our example, mibgen provides a plausible default value according to the variable type.

Our implementations of MIB behavior are contained in the classes with the Impl suffix. These implementation classes extend those that are generated by mibgen so that we can regenerate them without overwriting our customizations.

Here is a summary of the implementation shown in the agent example:

- `InterfacesImpl.java` - adds a notification listener to the `IfTable` object, then creates two table entries with plausible values and adds them to the table; this class is associated with:
  - `TableEntryListenerImpl.java` - the listener for table notifications when entries are added or removed: it prints out the values of a new table entry and prints a message when an entry is removed
  - `IfEntryImpl.java` - implements a table entry and provides an internal method for switching the `OperStatus` variable that triggers a trap (see Example 16–4); this method is not exposed in the `MBean` interface, so it is only available to the code of this agent application
- `SnmpImpl.java` - initializes and implements variables of the SNMP group; many of these are state variables of the SNMP agent, so we call the getter methods of the SNMP adaptor object to return the information
- `SystemImpl.java` - initializes the `System` group variables with realistic values

The `SnmpImpl.java` and `SystemImpl.java` files provide code that you can reuse when you need to implement these common SNMP groups.

16.1.3 Compiling the MBeans and Agents

Compile all the classes in the *examplesDir*`/current/Snmp/Agent` directory. The classpath must contain the current directory (`.`):

```
$ javac -classpath classpath -d . *.java
```

We are now ready to look at the implementation of SNMPv1/v2 and SNMPv3 agents and run the example applications.

16.2 SNMP Protocol Adaptor

Once your MIBs are implemented as MBeans, your agent application needs an SNMP protocol adaptor to function as an SNMP agent. Because the SNMP adaptor is also an MBean, it can be created and started dynamically in your agent by a connected manager, or through the HTML adaptor.

Since Java DMK 5.1, the performance of the SNMP protocol adaptor has been improved by the addition of multithread support.

The following SNMPv1/v2 `Agent` example launches the protocol adaptor for SNMPv1/v2 agents, through the code of the agent application.

EXAMPLE 16–1 SNMPv1/v2 Agent Application

```
public class Agent {

    static SnmpAdaptorServer snmpAdaptor = null;

    private static int nbTraps = -1;

    public static void main(String args[]) {

        final MBeanServer server;
        final ObjectName htmlObjName;
        final ObjectName snmpObjName;
        final ObjectName mibObjName;
        final ObjectName trapGeneratorObjName;
        int htmlPort = 8082;
        int snmpPort = 161;

        [...]
        try {
            server = MBeanServerFactory.createMBeanServer();
            String domain = server.getDefaultDomain();

            // Create and start the HTML adaptor.
            //
            htmlObjName = new ObjectName( domain +
              ":class=HtmlAdaptorServer,protocol=html,port="
                       + htmlPort);
            HtmlAdaptorServer htmlAdaptor = new HtmlAdaptorServer(htmlPort);
            server.registerMBean(htmlAdaptor, htmlObjName);
            htmlAdaptor.start();

            // Create and start the SNMP adaptor.
            //
            snmpObjName = new ObjectName(domain +
                ":class=SnmpAdaptorServer,protocol=snmp,port=" + snmpPort);
            snmpAdaptor = new SnmpAdaptorServer(snmpPort);
            server.registerMBean(snmpAdaptor, snmpObjName);
            snmpAdaptor.start();

            // The rest of the code is specific to our SNMP agent

            // Send a coldStart SNMP Trap (use port = snmpPort+1)
            // Trap communities are defined in the ACL file
            //
            snmpAdaptor.setTrapPort(new Integer(snmpPort+1));
            snmpAdaptor.sendV1Trap(0, 0, null);

            // Create the MIB-II (RFC 1213) and add it to the MBean server.
            //
            mibObjName = new ObjectName("snmp:class=RFC1213_MIB");
            RFC1213_MIB mib2 = new RFC1213_MIB();
            // The MBean will register all group and table entry MBeans
            // during its pre-registration
            server.registerMBean(mib2, mibObjName);
```

EXAMPLE 16–1 SNMPv1/v2 Agent Application *(Continued)*

```
            // Bind the SNMP adaptor to the MIB
            snmpAdaptor.addMib(mib2);

            [...]

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Needed to get a reference on the SNMP adaptor object
    static public SnmpAdaptorServer getSnmpAdaptor() {
        return snmpAdaptor;
    }
}
```

The following SNMPv3 AgentV3 example launches the protocol adaptor for SNMPv3 agents, in the same way as in the SNMPv1/v2 Agent example. This example creates a simple SNMPv3 agent, without encryption. See "Security Mechanisms in the SNMP Toolkit" for details of how to implement SNMPv3 agents with encryption.

EXAMPLE 16–2 SNMPv3 AgentV3 Application

```
public class AgentV3 {

    static SnmpV3AdaptorServer snmpAdaptor = null;

    private static int nbTraps = -1;

    public static void main(String args[]) {

        final MBeanServer server;
        final ObjectName htmlObjName;
        final ObjectName snmpObjName;
        final ObjectName mibObjName;
        final ObjectName trapGeneratorObjName;
        int htmlPort = 8082;
        int snmpPort = 161;
[...]

        try {
            server = MBeanServerFactory.createMBeanServer();
            String domain = server.getDefaultDomain();

            // Create and start the HTML adaptor.
            //
            htmlObjName = new ObjectName(domain +
                 ":class=HtmlAdaptorServer,protocol=html,port="
                 + htmlPort);
            HtmlAdaptorServer htmlAdaptor = new HtmlAdaptorServer(htmlPort);
```

EXAMPLE 16–2 SNMPv3 `AgentV3` Application *(Continued)*

```
        server.registerMBean(htmlAdaptor, htmlObjName);
        htmlAdaptor.start();

        // Create and start the SNMP adaptor.
        //
        snmpObjName = new ObjectName(domain +
            ":class=SnmpAdaptorServer,protocol=snmp,port=" + snmpPort);
        snmpAdaptor = new SnmpV3AdaptorServer(snmpPort);
        server.registerMBean(snmpAdaptor, snmpObjName);
           snmpAdaptor.registerUsmMib(server, null);
        snmpAdaptor.start();

        // Send a coldStart SNMP Trap.
        // Use port = snmpPort+1.
        //
        snmpAdaptor.setTrapPort(new Integer(snmpPort+1));
        snmpAdaptor.snmpV1Trap(0, 0, null);
        println("Done.");

        // Create the MIB II (RFC 1213) and add it to the MBean server.
        //
        mibObjName= new ObjectName("snmp:class=RFC1213_MIB");
        RFC1213_MIB mib2 = new RFC1213_MIB_IMPL();
        server.registerMBean(mib2, mibObjName);

        // Bind the SNMP adaptor to the MIB
            snmpAdaptor.addMib(mib2, "TEST-CONTEXT");

    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static SnmpAdaptorServer getSnmpAdaptor() {
   return snmpAdaptor;
}
```

16.2.1 Starting the SNMP Adaptor

We start the SNMP adaptor in the same way that we start the HTML adaptor. First we create a meaningful object name for its MBean, then we instantiate the class with a constructor enabling us to specify a non-default port, we register the MBean with the MBean server, and we start the adaptor to make it active.

By default, the SNMP protocol adaptor uses the standard SNMP port 161. Because other applications might be using this port on a host, our simple agent, and all the other examples in this section, use port 8085. When we connect to this agent, our SNMP manager must specify this non-standard port number.

Note – On certain platforms, applications also require superuser privileges to assign the default SNMP port 161. If your SNMP adaptor uses this port, its agent application must be started with superuser privileges.

16.2.2 Creating MIB MBeans

Our agent application creates and manages one MIB, our subset of MIB-II. To do so, it instantiates the corresponding `RFC1213_MIB` MBean that was generated by the `mibgen` tool (see "16.1 MIB Development Process" on page 264). We give it a meaningful object name and then we register it in the MBean server.

The registration process enables the MBean to instantiate and register other MBeans that represent the groups of the MIB and the entries of its tables. The set of all these MBeans at the end of registration makes up the MBean representation of the MIB. If an SNMP manager later adds entries to a table, the MIB implementation registers the new entry MBean into the MBean server as well.

If you do not want to expose a MIB through the MBean server, you do not have to register it. However, you still need to create all of its other MBean objects so that the SNMP adaptor can access all of its groups and table entries. The generated code provides the `init` method in the main MBean of the MIB. Calling this method creates all necessary MBean objects without registering them in the MBean server.

16.2.3 Binding the MIB MBeans

The SNMP adaptors do not interact with MBeans in the same way as the other connectors and adaptors. Because the SNMP data model relies on MIBs, only MBeans representing MIBs can be managed through SNMP. The SNMP adaptor does not interact with MBeans of a MIB through the MBean server, they must be explicitly bound to the instance of the SNMP adaptor.

After a MIB is instantiated, you must add the MIB to the SNMP adaptor by calling the `setSnmpAdaptor` method.

In the binding process, the SNMP adaptor obtains the root OID of the MIB. The adaptor uses this OID to determine which variables are implemented in the MIB's corresponding MBeans.

Even though the SNMP adaptors can be registered in the MBean server, the adaptor only makes MIBs visible to SNMP managers. Other MBeans in the agent cannot be accessed or even represented in the SNMP protocol. The SNMP manager is limited by its protocol: it cannot take full advantage of a Java dynamic management agent through the basic MIBs, and it does not have access to any other MBeans. In an advanced management solution, you could write a special MIB and implement it so that operations on its variables actually interact with the MBean server. This is left as an exercise for the reader.

16.2.4 MIB Scoping

In the `SnmpV3AdaptorServer`, it is possible to specify a scope, or context name, when registering a MIB. Conceptually, there is one instance of the global OID tree per scope. If you do not specify a context name when registering a MIB, the MIB is registered in the default scope. You can thus register `SnmpProxy` objects in any default or specific scope. Routing and MIB overlapping is handled in the same way in any scope.

When an incoming request is received, the `SnmpV3AdaptorServer` first determines the scope of that request. Only the SNMP agents that have been registered in that scope are triggered by that request.

If the incoming request is an SNMPv3 request, the scope is identified by the context name specified in the request protocol data unit (PDU). If no context name is specified, the scope of that request is the default scope. If the incoming request is an SNMPv1 or v2 request, the scope of the request is the default scope, unless mapping community strings to context names is enabled in the adaptor.

To register a MIB in a specific context, call the following method:

```
myMib.setSnmpAdaptor(myAdaptor, "myContext")
```

This method only works for adaptors that have been registered with an MBean server.

16.2.5 Accessing a MIB MBean

Once the MBean representing a MIB has been instantiated and bound to one of the two SNMP adaptors, it is accessible through that SNMP adaptor. SNMP managers can send requests to operate on the contents of the MIB. The SNMP adaptor interprets the SNMP management requests, performs the operation on the corresponding MBean and returns the SNMP response to the manager. One SNMP protocol adaptor is compatible with SNMPv1 and SNMPv2c, and the other is compatible with these two protocols as well as SNMPv3.

The advantage of having an SNMP agent “inside” a Java dynamic management agent is that you can use the other communications protocols to interact with MIBs and manage the SNMP adaptor. Because both the registered MIBs and the adaptor are MBeans, they are exposed for management. In our simple agent, the MIB was registered, and you can view its MBeans in a web browser through the HTML protocol adaptor.

If our agent were to include other connectors, management applications could connect to the agent and also manage the MIB and the SNMP adaptor. A non-SNMP manager could instantiate new MIB objects, bind them to the SNMP adaptor and operate on the exposed attributes and operations of the MIB.

Non-SNMP managers can operate on the variables of a MIB, getting and setting values, regardless of any SNMP manager that might also be accessing them through the SNMP adaptor. When dealing with a table, however, they cannot create new table entry MBeans without adding them to the table. For example, in the `InterfacesImpl.java` class, we called the `addEntry` method of the `IfTable` object before registering the entry MBeans with the MBean server. This ensures that the new entries are visible when an SNMP manager accesses the table.

In order for a non-SNMP manager to create a table entry, you must customize the table's group MBean to expose this functionality. Briefly, you would need to write a new method that instantiates and initializes the entry's MBean, adds the MBean to the table object, and registers the entry MBean in the MBean server. Advanced customization such as this is not covered in our examples. In general, the designer of the agent and management applications is responsible for all coherency issues when accessing MIBs concurrently through different protocols and when adding table entries.

16.2.6 Managing the SNMP Adaptors

Non-SNMP managers can also control the SNMP agent through the MBean of the SNMP adaptors. Like the other communications MBeans, the port and other attributes can be modified when the SNMP adaptor is stopped. You can also get information about its state, and stop or restart it to control when it is online. These administrative attributes and operations are defined in the `CommunicatorServerMBean` interface.

The SNMPv1v/2 adaptor server implements the `SnmpAdaptorServerMBean` interface to define its operating information. The SNMPv3 adaptor server implements a similar interface called `SnmpV3AdaptorServerMBean`. The SNMP protocols define certain variables that SNMP agents must expose about their current state. For example, the SNMP adaptor provides methods for `getSnmpInPkts` and `getSnmpOutBadValues`. Non-SNMP managers can read these variables as attributes of the SNMP adaptor MBean.

The SNMP adaptors also expose other operating information that is unavailable to SNMP managers. For example, the `ActiveClientCount` and `ServedClientCount` read-only attributes report on SNMP manager connections to this agent. The read-write `BufferSize` attribute enables you to change the size of the message buffer, but only when the adaptor is not online. The adaptor MBean also exposes operations for sending traps or implementing your own security (see "19.2.1 Enabling User-Based Access Control" on page 360).

▼ To Run the SNMPv1/v2 `Agent` Example

1. **After building the example as described in "16.1 MIB Development Process" on page 264, start the simple SNMPv1/v2 agent with the following command:**

   ```
   $ java -classpath classpath:. Agent nbTraps
   ```

   Set *nbTraps* to zero.

   You should see some initialization messages, including our notification listener giving information about the two table entries that are created. Access this agent's HTML adaptor by pointing a web browser to the following URL: `http://localhost:8082/`.

2. **Through the HTML adaptor, you can see the MBeans representing the MIB:**
   - The `class=RFC1213_MIB` MBean in the `snmp` domain is the MBean representing the MIB; it contains a name and information about the SNMP adaptor to which the MIB is bound
   - The `RFC1213_MIB` domain contains the MBeans for each group; both `name=Snmp` and `name=System` contain variables with values provided by our customizations
   - The `RFC1213_MIB/ifTable` domain contains the entries of the Interfaces table
   - The `trapGenerator` domain contains the class that sends traps periodically, as part of our sample MIB implementation

3. **In any of these MBeans, you can write new values into the text fields of exposed attributes and click the `Apply` button.**

   This sets the corresponding SNMP variable, and thereafter, SNMP managers see the new value. This is an example of managing a MIB through a protocol other than SNMP.

4. **Press `Control C` when you have finished viewing the agent.**

16.2.7 Configuring SNMPv3 Security for Agents

Before you run the SNMPv3 agent examples, you require some information about how SNMPv3 security is configured. Below are brief descriptions of the SNMPv3 security files that provide you with the information you need to run the SNMPv3 examples in this chapter. Full descriptions of the SNMPv3 security mechanisms are given in "19.3 SNMPv3 User-Based Security Model" on page 362.

The SNMPv3 security mechanisms are defined in two text files:

- `jdmk.security`
- `jdmk.uacl`

The files used by the SNMPv3 agent examples are provided in the *examplesDir*/current/Snmp/Agent directory. These files are used by the examples in the subsequent sections of this chapter.

EXAMPLE 16–3 A jdmk.security File for an SNMPv3 Agent

The jdmk.security identifies the SNMP engine, authorized user and the security settings for the SNMPv3 session:

```
#Local engine ID
localEngineID=0x8000002a05819dcb6e00001f95
#Number of boots
localEngineBoots=0

#User and security configuration
userEntry=localEngineID,defaultUser,,usmHMACMD5AuthProtocol,mypasswd
```

The local engine ID and the number of times that engine will boot are read by the agent when it is created.

The authorized users and the security levels for the SNMP session are defined by the userEntry. This particular jdmk.security file defines a user that implements authentication, but not privacy. Consequently, the settings are as follows:

localEngineID	The identifier of the local engine, as specified earlier in the file
defaultUser	The name of the authorized user
usmHMACMD5AuthProtocol	The authentication algorithm; in this case, HMAC MD5
myPasswd	The authentication password

Note – User-based access control is not used by the examples in this chapter, so we do not examine the jdmk.uacl file here. See Chapter 19 to find out how to implement user-based access control.

▼ To Run the SMNPv3 AgentV3 Example

1. **After building the example as described in "16.1 MIB Development Process" on page 264, start the simple SNMPv3 agent with the following command:**

   You have to direct the AgentV3 example to its security file to run it.

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   AgentV3 nbTraps
   ```

   Set *nbTraps* to zero.

You should see some initialization messages, including our notification listener giving information about the two table entries that are created. Access this agent's HTML adaptor by pointing a web browser to the following URL: `http://localhost:8082/`.

2. **Through the HTML adaptor, you can see the MBeans representing the MIB:**

   - The `SNMP_USER_BASED_SM_MIB` domain contains information pertaining to the user-based security model implemented; see "Security Mechanisms in the SNMP Toolkit" for details of how to implement SNMPv3 user-based security.
   - The `class=RFC1213_MIB` MBean in the `snmp` domain is the MBean representing the MIB; it contains a name and information about the SNMP adaptor to which the MIB is bound.
   - The `RFC1213_MIB` domain contains the MBeans for each group; both `name=Snmp` and `name=System` contain variables with values provided by our customizations.
   - The `RFC1213_MIB/ifTable` domain contains the entries of the Interfaces table.
   - The `trapGenerator` domain contains the class that sends traps periodically, as part of our sample MIB implementation.

3. **In any of these MBeans, you can write new values into the text fields of exposed attributes and click the "Apply" button.**

   This sets the corresponding SNMP variable, and thereafter, SNMP managers see the new value. This is an example of managing a MIB through a protocol other than SNMP.

4. **Press `Control C` when you have finished viewing the agent.**

16.3 Sending Traps

Agents can send unsolicited event reports to management applications by using traps. The SNMP protocol adaptor can send v1, v2 and v3 traps, the differences being in the format of the corresponding PDU. Traps in the SNMP specification are not acknowledged by the management application, so agents do not know if traps are received. However, under SNMPv2 and v3, acknowledgements can be sent in the form of informs.

Inform requests are acknowledged event reports, they are sent by entities acting in a manager role, according to RFC 1905. In the Java DMK, both the SNMP adaptor and the classes of the SNMP manager API can send inform requests. Manager-to-manager inform requests are described in "17.3 Inform Requests" on page 311. Agent-to-manager inform requests are demonstrated by the applications in the `current/Snmp/Inform/` example directory located in the main *examplesDir* directory.

In this example, we demonstrate how our simple SNMP agent application can send traps. The class `IfEntryImpl` in the example directory extends the `IfEntry` class generated by `mibgen` to provide a method that switches the `IfOperStatus` variable and sends a trap. Before sending the trap, the example establishes whether the agent is an SNMPv1/v2 agent or SNMPv3, and sends a trap accordingly. This is an example of customization of the generated code: an agent-side entity switches the operation status, the MIB variable is updated and a trap is sent to SNMP managers.

EXAMPLE 16–4 Sending a Trap in the `IfEntryImpl` Class

```
public void switchifOperStatus() {
// Implement the switch and then calls sendTrap indirectly
 [...]
}

// Identify the adaptor created by one of the entry-point
// classes for this example.

private void initialize() {

    boolean encryption=false;
    SnmpAdaptorServer snmpAdaptor = null;

  // Determine wheter we are within the the scope of Agent.java,
  // StandAloneSnmpAgent.java, AgentV3.java, AgentEncryptV3.java

  try {
  snmpAdaptor = Agent.getSnmpAdaptor();
  if (snmpAdaptor != null) return;

  snmpAdaptor = StandAloneSnmpAgent.getSnmpAdaptor();
  if (snmpAdaptor != null) return;

  snmpAdaptor = AgentV3.getSnmpAdaptor();
  if (snmpAdaptor != null) return;

  snmpAdaptor = AgentEncryptV3.getSnmpAdaptor();
  if (snmpAdaptor != null) {
  encryption=true;
  return;
  }

} finally {
  // Set the value of this.snmpAdaptor and
  // this.securityLevel
  //
  this.snmpAdaptor=snmpAdaptor;
  this.securityLevel = SnmpDefinitions.noAuthNoPriv;
```

EXAMPLE 16–4 Sending a Trap in the `IfEntryImpl` Class *(Continued)*

```
    if (snmpAdaptor == null) return;
    if (snmpAdaptor instanceof SnmpV3AdaptorServer) {
    this.securityLevel = (encryption?SnmpDefinitions.authPriv:
                  SnmpDefinitions.authNoPriv);
    }
}
}

    // Send SNMP v1 traps with the generic number of the trap
    // according to the "IfOperStatus" value.
 //
    public void sendTrap(int generic) {

    if (snmpAdaptor == null) {
       java.lang.System.err.println("BUG:
            IfEntryImpl.sendTrap():
            snmpAdaptor is null");
       return;
    }

    String generic_string= null;

    switch (generic) {
    case 3:
        generic_string = "linkUp";
        break;
    case 2:
        generic_string = "linkDown";
        break;
    default:
        java.lang.System.err.println("BUG: IfEntryImpl.sendTrap():
        bad generic: " + generic);
        return;
    }

    SnmpVarBindList varBindList = new SnmpVarBindList();

    SnmpOid oid1 = new SnmpOid("1.3.6.1.2.1.2.2.1.1." + IfIndex);
    SnmpInt value1 = new SnmpInt(IfIndex);
    SnmpVarBind varBind1 = new SnmpVarBind(oid1, (SnmpValue) value1);

    varBindList.addVarBind(varBind1);

    java.lang.System.out.print("NOTE: Sending a " + generic_string +
                               " SNMP trap for the Interface " +
               IfIndex +
                               " to each destination defined" +
               " in the ACL file...");
    try {
        //Send SNMPv3 trap if v3 agent only.
            if(snmpAdaptor instanceof SnmpV3AdaptorServer) {
            final SnmpV3AdaptorServer adaptorV3=
```

EXAMPLE 16–4 Sending a Trap in the `IfEntryImpl` Class *(Continued)*

```
                    (SnmpV3AdaptorServer)snmpAdaptor;
                adaptorV3.snmpV3UsmTrap("defaultUser",
                    securityLevel,
                    "TEST-CONTEXT",
                    new SnmpOid("1.2.3.4.5.6.7.8.9.0"),
                    varBindList);
        }
                // Send v1 trap in every case.
                snmpAdaptor.snmpV1Trap(generic, 0, varBindList);
            } catch (Exception e) {
            e.printStackTrace();
        }
        java.lang.System.out.println("Done.");
    }
}
```

As the `sendTrap` method runs in a different thread, it needs to get a reference to the SNMP adaptor instance. Here we call the static methods of our different possible agent implementations. This code is specific to these agents and is only an example of how to retrieve this information.

As shown in the above example, SNMPv1 traps are always sent, regardless of the version of SNMP of the agent. An SNMPv3 trap is sent only when the agent is an SNMPv3 agent.

To simulate a live operation status, we invent the `LinkTrapGenerator` class that switches the status periodically. It is an MBean that contains a thread that loops endlessly. The interval period between traps and the number of the table entry can be modified through the MBean's attributes.

EXAMPLE 16–5 Thread of the Link Trap Generator

```
public void run() {
    int remainingTraps = nbTraps;
    while ((nbTraps == -1) || (remainingTraps > 0)) {
        try {
            sleep(interval);
        } catch (Exception e) {
            e.printStackTrace();
        }
        triggerTrap();
        remainingTraps--;
    }
}

public void triggerTrap() {
    // get the entry whose status we will switch
    IfEntryImpl ifEntryImpl = InterfacesImpl.find(ifIndex);
    if (ifEntryImpl == null) {
```

EXAMPLE 16–5 Thread of the Link Trap Generator *(Continued)*

```
            errors++;
            return;
        }
        ifEntryImpl.switchifOperStatus();
        successes++;
}
```

To run the trap generator, the example application instantiates and registers a `LinkTrapGenerator` MBean. During its registration, this MBean starts the thread, sending a trap every two seconds by default.

EXAMPLE 16–6 Starting the Trap Generator Example

```
// Create a LinkTrapGenerator (specify the ifIndex in the object name)
//
String trapGeneratorClass = "LinkTrapGenerator";
int ifIndex = 1;
trapGeneratorObjName = new ObjectName("trapGenerator" +
    ":class=LinkTrapGenerator,ifIndex=" + ifIndex);
LinkTrapGenerator trapGenerator = new LinkTrapGenerator(nbTraps);
server.registerMBean(trapGenerator, trapGeneratorObjName);

[...] // Press <Enter> to start sending traps

trapGenerator.start();
```

16.3.1 Specifying the Trap Destination

There are several methods in the SNMP protocol adaptor for sending traps to remote managers. They differ in their method signatures, depending upon whether or not you need to specify the destination host. When no host is specified, the SNMP protocol adaptor relies on the trap group definition in IP-based access control lists (`InetAddressAcl`), as described below.

In all cases, traps are sent to the port specified by the current value of the `TrapPort` attribute on the `SnmpAdaptorServer` or `SnmpV3AdaptorServer` MBean. In our simple agent, we set the trap port to 8086, but this can be changed at any time by a custom MIB implementation or a management application.

Although SNMPv3 implements user-based access control for other types of requests, traps and informs are always sent using `InetAddressAcl`, in all versions of SNMP.

16.3.1.1 Using an `InetAddressAcl` Trap Group

The methods below were used in Example 16–4 to send SNMPv1 and v3 traps. They are presented here with their SNMPv2 equivalent (see the Javadoc API for a description of the parameters).

- `sendV1Trap( int generic, int specific, java.util.Vector varBindList )`
- `sendV2Trap( SnmpOid trapOid, java.util.Vector varBindList )`
- `snmpV3UsmTrap("defaultUser", securityLevel, "TEST-CONTEXT", new SnmpOid("1.2.3.4.5.6.7.8.9.0"), varBindList)`

Using these methods, you must first define the trap group in an `InetAddressAcl`. See “19.1 IP-Based Access Control Lists” on page 353 for a formal definition of the trap group and instructions for defining the `InetAddressAcl` file when starting the agent. By default, these lists are file-based, but you can implement other mechanisms, as described in “19.1.3 Custom Access Control” on page 358.

In this example, we provide the following template file.

EXAMPLE 16–7 Trap Group of the `jdmk.acl` File

```
acl = {
  ...
}

trap = {
  {
  trap-community = public
  hosts = yourmanager
  }
}
```

The trap group lists all the hosts to which the SNMP protocol adaptor sends every trap. A community definition associates a community name with a list of hosts specified either by one of the following identifiers:

Hostname	The name of the host
IP v4 and IPv6 address	For example, `123.456.789.12` for IPv4, and `fe80::a00:20ff:fe9b:ea82` for IPv6
IPv4 and IPv6 netmask prefix notation	For example, `123.456.789.12/24` for IPv4, and `fe80::a00:20ff:fe9b:ea82/64` for IPv6

All hosts in a community definition receive the trap in a PDU identified by the community name.

Note – Because access control and trap recipients share the same file, you must fully define the access control when you want to send traps using the InetAddressAcl mechanism.

Given this definition, traps are sent to a host called *yourmanager*, and the community string of the trap PDU would contain the value public. By adding community definitions to this file, you can specify all hosts that will receive traps along with the community string for each host or group of hosts.

Note – SNMPv3 does not use the community string to identify destinations. Only use the manager's IP address when creating an SNMPv3 trap group, or the contextName to define the scope of the requests sent.

If the InetAddressAcl file is not defined, or if the trap group is empty, the default behavior of these methods is to send a trap only to the local host.

16.3.1.2 Specifying the Hostname Directly

The other methods of the SNMP protocol adaptor, one for each trap version, enable you to send a trap to a specified recipient:

- sendV1Trap(java.net.InetAddress address, java.lang.String cs, …)
- sendV2Trap(java.net.InetAddress address, java.lang.String cs, …)
- snmpV3UsmTrap(java.net.InetAddress address, contextName, …)

In the first two cases, these methods take an address and a community string, in addition to the version-specific trap information. The address is an InetAddress object that is usually instantiated by its static methods getLocalHost or getByName. The second method returns a valid InetAddress object when given a string representing a hostname or IP address.

The cs parameter is the community string, a name that the agent and manager exchange to help identify one another. The string given is used as the community when sending the trap PDU.

The SNMPv3 method also takes an InetAddress, but does not use the community string. Only use the manager's IP address when creating an SNMPv3 trap group, or the contextName to define the scope of the requests sent.

Either one of these methods sends a trap to a single manager using a single community string. The InetAddressAcl trap group mechanism is better suited to sending traps to multiple managers, though it requires you to set up a trap group. Note that even if a trap group is in use, the two methods above only send one trap to the specified host address.

16.3.2 Traps in the Agent and AgentV3 Examples

Before starting the SNMP agent again, edit the jdmk.acl file to replace the occurrences of *yourmanager* with the name of a host running an SNMP manager.

You can start the example SNMPv3 agent by replacing Agent with AgentV3.

Start the agent, specifying the InetAddressAcl file as a property:

```
$ java -classpath classpath
-Djdmk.acl.file=examplesDir/current/Snmp/Agent/jdmk.acl \
       -classpath classpath:. Agent nbTraps
```

In these commands, *nbTraps* is the number of traps that the agent sends. Set it to a small integer to avoid too much output. If you omit this parameter, traps are sent continuously.

If you do not have an SNMP manager or a second host, do not copy the InetAddressAcl file or specify it as a property. In the absence of the trap-community definitions, the traps are addressed to the trap port on the local host. And even if no manager is running, we can still see the agent sending the traps. See "17.1.4 SNMP Trap Handler" on page 299 for details about receiving traps.

▼ To Interact With the Trap Generator

1. **Access this agent's HTML adaptor by pointing a web browser to the following URL: http://localhost:8082/. Click on the class=LinkTrapGenerator,ifIndex=1 MBean in the trapGenerator domain.**

   Through the HTML adaptor, you can see the MBean representing the trap generator object. You can modify its attributes to change the table entry that it operates on and to change the interval between traps.

2. **Change the trap interval to 10000 so that traps are sent every 10 seconds.**

3. **Go back to the agent view and click on the ifEntry.ifIndex=1 MBean in the ifTable domain. Set the reload period to 10, and click the Reload button.**

   You should see the effect of the trap generator is to switch the value of the IfOperStatus variable. It is our implementation of the table entry that sends a trap when this status is changed.

4. **Go back to the agent view and click on the `name=Snmp` MBean in the `RFC1213_MIB` domain. Scroll down to see the `SnmpOutPkts` and `SnmpOutTraps` variables.**

   These variables should be the only nonzero values, if no manager has connected to the SNMP agent. The `Snmp` group shows information about the SNMP adaptor, and we can see how many traps have been sent since the agent was started.

5. **Press `Control-C` when you have finished interacting with the agent.**

The `LinkTrapGenerator` MBean is not manageable through the SNMP adaptor because it is not part of any MIB. It is an example of another MBean providing some control of the SNMP agent, and this control can be exercised by other managers connecting through other protocols. This shows that designing an SNMP agent application involves both the implementation of the MIB functionality and, if desired, the implementation of other dynamic controls afforded by the JMX architecture and the services of Java DMK.

16.4 Standalone SNMP Agents

The design of the SNMP protocol adaptors and of the MBeans generated by `mibgen` give you the option of creating an SNMP agent that is not a Java dynamic management agent.

This standalone agent has no MBean server and thus no possibility of being managed other than through the SNMP protocol. The application must instantiate all MIBs that the SNMP agent needs, as it is impossible to create them through another manager. The advantage of a standalone agent is the reduced size of the application, in terms of memory usage.

This applies to all three versions of SNMP.

EXAMPLE 16–8 `StandAloneSnmpAgent` Example

```
;
import com.sun.management.comm.SnmpAdaptorServer;

public class StandAloneSnmpAgent {

    static SnmpAdaptorServer snmpAdaptor = null;

    private static int nbTraps = -1;

    public static void main(String args[]) {
```

EXAMPLE 16–8 `StandAloneSnmpAgent` Example *(Continued)*

```
        // Parse command line and enable tracing
        [...]

        try {
            // The agent is started on a non standard SNMP port: 8085
            int port = 161;
            port = 8085;
            snmpAdaptor = new SnmpAdaptorServer(port);

            // Start the adaptor
            snmpAdaptor.start();

            // Send a coldStart SNMP Trap
            snmpAdaptor.setTrapPort(new Integer(port+1));
            snmpAdaptor.snmpV1Trap(0, 0, null);

            // Create the MIB you want in the agent (ours is MIB-II subset)
            RFC1213_MIB mib2 = new RFC1213_MIB_IMPL();

            // Initialize the MIB so it creates the associated MBeans
            mib2.init();

            // Bind the MIB to the SNMP adaptor
            snmpAdaptor.addMib(mib2);

            // Optional: create a LinkTrapGenerator
            int ifIndex = 1;
            LinkTrapGenerator trapGenerator =
                 new LinkTrapGenerator(ifIndex, nbTraps);
            trapGenerator.start();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Needed to get a reference on the SNMP adaptor object
    static public SnmpAdaptorServer getSnmpAdaptor() {
        return snmpAdaptor;
    }
}
```

As this example demonstrates, the standalone agent uses exactly the same MIB MBeans, with the same customization, as our other agents. However, instead of registering them in the MBean server, they are only instantiated. And whereas the registration process creates all subordinate MBeans of the MIB, now we must call its `init` method explicitly.

The init method performs the same function as the preRegister method, only it does not register the MBean with the MBean server. Each of the group MBeans then has two constructors, one with and one without a reference to the MBean server. When table entries are added dynamically, the corresponding object only registers the new entry's MBean if the MBean server reference is non-null; that is, only if the MBean is not instantiated in a standalone agent.

The mibgen tool automatically generates both the pre-registration methods and the init methods in the MIB MBeans. Therefore, no special action is necessary to use them in either a regular agent or a standalone agent. If you use a standalone agent for memory considerations, you can remove the registration process from the generated MBean and only customize the init process.

EXAMPLE 16–9 Customizations in the Generated RFC1213_MIB_Impl.java File

```
class RFC1213_MIB_IMPL extends RFC1213_MIB {

    public RFC1213_MIB_IMPL() {
        super();
    }
    /**
     * Passing it a name in order to register the same mib in 2 MBeanServer.
     */
    public RFC1213_MIB_IMPL(String name) {
        super();
        mibName = name;
    }
        protected Object createSnmpMBean(String groupName, String groupOid,
                     ObjectName groupObjname,
                     MBeanServer server)  {

        // Note that when using standard metadata,
        // the returned object must implement the "InterfacesMBean"
        // interface.
        //
        if (server != null)
            return new SnmpImpl(this,server);
        else
            return new SnmpImpl(this);
    }

        protected Object createSystemMBean(String groupName,
                       String groupOid,
                       ObjectName groupObjname,
                       MBeanServer server)  {

        // Note that when using standard metadata,
        // the returned object must implement the "InterfacesMBean"
        // interface.
        //
        if (server != null)
            return new SystemImpl(this,server);
        else
```

EXAMPLE 16–9 Customizations in the Generated `RFC1213_MIB_Impl.java` File *(Continued)*

```
            return new SystemImpl(this);
    }

        protected Object createInterfacesMBean(String groupName,
                       String groupOid,
                       ObjectName groupObjname,
                       MBeanServer server)  {

        // Note that when using standard metadata,
        // the returned object must implement the "InterfacesMBean"
        // interface.
        //
        if (server != null)
            return new InterfacesImpl(this,server);
        else
            return new InterfacesImpl(this);
    }

}
```

After the MIB is initialized, it only needs to be bound to the SNMP adaptor, as in the other agents; except that in the standalone case, we use the `setSnmpAdaptor` method which takes a direct reference to the SNMP adaptor instead of an object name. That is all you need to do when programing a standalone SNMP agent.

You can subclass the MIB and override the group MBean factory method to instantiate your own customized MBean class, replacing for instance `new Interfaces()` with `new InterfacesImpl()`, as shown in the code example.

16.4.1 Running the Standalone Agent Example

▼ To Run the Standalone Agent Example

1. **Start the standalone agent with the following command:**

   ```
   $ java -classpath classpath StandAloneSnmpAgent nbTraps
   ```

2. **If you have not copied the `jdmk.acl` file to the configuration directory, add the following property to your command line:**

   ```
   -Djdmk.acl.file=examplesDir/current/Snmp/Agent/jdmk.acl
   ```

   You should see the same initialization messages as with the simple agent. Then, you should see the agent sending out a trap every two seconds. If you have an

SNMP manager application, you can send requests to the agent and receive the traps. See "Developing an SNMP Manager" for example applications you can use.

The only limitation of a standalone agent is that you cannot access or manage the SNMP adaptor and MIB MBeans in the dynamic management sense. However, the SNMP adaptor still relies on the `InetAddressAcl` file for access control and traps, unless you have customized the `InetAddressAcl` mechanism, and you can implement other security schemes as described in "19.2.1 Enabling User-Based Access Control" on page 360.

3. **Press `Control-C` when you have finished running the standalone agent.**

16.5 Multiple Agents

It is possible to create multiple SNMP agents in the Java virtual machine (JVM). A multiple agent is an agent that instantiates more than one SNMP adaptor server. This enables you to create multiple SNMP MIBs using that agent, with each MIB having its own individual security configuration. The main advantage of this is to reduce the burden on the JVM.

You can create multiple agents using all three versions of SNMP. However, the multiple agent demonstrated in the following example is an SNMPv3 agent that creates and registers two SNMP adaptor servers in the MBean server. Registration in the MBean server enables the MIBs generated to be managed through the HTML adaptor.

EXAMPLE 16–10 `MultipleAgentV3` Example

```
public class MultipleAgentV3 {

    static SnmpV3AdaptorServer snmpAdaptor1 = null;
    static SnmpV3AdaptorServer snmpAdaptor2 = null;

       //Set the number of traps
        //
    private static int nbTraps = -1;

      public static void main(String args[]) {

                MBeanServer server;
               ObjectName htmlObjName;
                ObjectName snmpObjName1;
                   ObjectName snmpObjName2;
                ObjectName mibObjName1;
                   ObjectName mibObjName2;
                ObjectName trapGeneratorObjName;
                int htmlPort = 8082;
```

EXAMPLE 16–10 `MultipleAgentV3` Example *(Continued)*

```
  // Initialize trace property.
  //
  [...]

 // Create and start the HTML adaptor.
//
[...]

// Create and start the first SNMP adaptor.
int snmpPort = 8085;
snmpObjName1 = new ObjectName(domain +
   ":class=SnmpAdaptorServer,protocol=snmp,port="
      + snmpPort);
snmpAdaptor1 = new SnmpV3AdaptorServer(snmpPort);
server.registerMBean(snmpAdaptor1, snmpObjName1);
   snmpAdaptor1.registerUsmMib(server, null);
snmpAdaptor1.start();

// Send a coldStart SNMP Trap.
// Use port = snmpPort+1.
//
print("NOTE: Sending a coldStart SNMP trap"+
   " to each destination defined in the ACL file...");
snmpAdaptor1.setTrapPort(new Integer(snmpPort+1));
snmpAdaptor1.snmpV1Trap(0, 0, null);
   snmpAdaptor1.enableSnmpV1V2SetRequest();
println("Done.");


  // Create and start the second SNMP adaptor.
//
  snmpPort = 8087;
 snmpObjName2 = new ObjectName(domain +
   ":class=SnmpAdaptorServer,protocol=snmp,port=" + snmpPort);
Trace.send(Trace.LEVEL_TRACE, Trace.INFO_MISC, "Agent", "main",
      "Adding SNMP adaptor to MBean server with name \n\t"+
  snmpObjName2);
println("NOTE: SNMP Adaptor is bound on UDP port " + snmpPort);


  // Instantiate second adaptor instantiation.
 //
  SnmpEngineParameters params = new SnmpEngineParameters();
   params.setSecurityFile("jdmk2.security");

  //Create the adaptor passing it the parameters.
snmpAdaptor2 = new SnmpV3AdaptorServer(params,
null,
null,
snmpPort,
null);
```

EXAMPLE 16–10 `MultipleAgentV3` Example *(Continued)*

```
            server.registerMBean(snmpAdaptor2, snmpObjName2);
                snmpAdaptor2.registerUsmMib(server, null);
            snmpAdaptor2.start();

            // Send a coldStart SNMP Trap.
            // Use port = snmpPort+1.
            //
               print("NOTE: Sending a coldStart SNMP trap"+
                 " to each destination defined in the ACL file...");
            snmpAdaptor2.setTrapPort(new Integer(snmpPort+1));
            snmpAdaptor2.snmpV1Trap(0, 0, null);
            println("Done.");


              // Create the first and second instances of MIB II
                  // (RFC 1213) and add them  to the MBean server.
               //
            mibObjName1 = new ObjectName("snmp:class=RFC1213_MIB");
            mibObjName2 = new ObjectName("snmp:class=RFC1213_MIB_2");

            // Create 2 instances of the customized MIB
            //
            RFC1213_MIB mib2 = new RFC1213_MIB_IMPL();

                RFC1213_MIB mib2_2 = new RFC1213_MIB_IMPL("RFC1213_MIB_2");
            server.registerMBean(mib2, mibObjName1);
                server.registerMBean(mib2_2, mibObjName2);

            // Bind the SNMP adaptor to the MIB in order to make the MIB
            //
                mib2.setSnmpAdaptor(snmpAdaptor, "TEST-CONTEXT");
                mib2_2.setSnmpAdaptor(snmpAdaptor2, "TEST-CONTEXT");

        //
        //Multiple agent is ready to answer SNMP requests.
        //

        } catch (Exception e) {
            e.printStackTrace();
        }
    }


}
```

This example creates an MBean server as normal, and creates and registers an HTML adaptor in the MBean server, again in the same way as was done in the simple agent example. However, `MultipleAgentV3` then creates and registers two SNMP adaptor servers in the MBean server. The two SNMP adaptors are bound to non-standard SNMP ports, in this case ports 8085 and 8087.

Each adaptor generates MIBs from its own security file, enabling you to have different, and remotely updatable, security configurations for each SNMP adaptor server. The first SNMP adaptor server, `snmpAdaptor1`, gets its security configuration from the security file `jdmk.security` when the agent is started. The second SNMP adaptor server, `snmpAdaptor2`, gets its security configuration from a second security file, `jdmk2.security`, using the `params.setSecurityFile` method. You can consult `jdmk2.security` in the *examplesDir*`/current/Snmp/Agent` directory. Both these security files implement authentication without privacy.

Caution – The coherency of the `jdmk.security` file is not preserved if you use the same file for more than one adaptor server.

16.5.1 Running the SNMPv3 `MultipleAgentV3` Example

In the same way as for the SNMPv3 agent example, `AgentV3`, you must point the multiple agent application to the security configuration file when you instantiate the multiple agent. This is only the case for the first of the SNMP adaptor servers created by your multiple agent application. All adaptor servers created subsequently by the multiple agent are directed to their corresponding security files in the code of the multiple agent application, when that adaptor server is instantiated.

You must call the `MultipleAgentV3` application inside its directory and make sure that the two security configuration files are present in the same directory.

▼ To Run the SNMPv3 `MultipleAgentV3` Example

1. **Type the following command:**

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   MultipleAgentV3
   ```

   The Java DMK multiple agent is now running on your system.

2. **To manage the agent through a web browser, connect to the following URL:**

   `http://localhost:8082/`

CHAPTER 17

Developing an SNMP Manager

The Java Management Extensions (JMX) specify the SNMP manager API for implementing an SNMP manager application in the Java programming language. This API is covered in the Javadoc API provided with the Java Dynamic Management Kit (Java DMK) (see "Related Books" in the Preface for more information). In this chapter, we explain the example applications that use this API.

The SNMP manager API can be used to access any SNMP agent, not just those developed with the Java DMK. It is compatible with SNMPv1, SNMPv2c and SNMPv3, and it includes mechanisms for handling traps. It lets you program both synchronous managers that block while waiting for responses and multi-threaded asynchronous managers that do not. Managers can also communicate with other managers using inform requests and responses.

The complete source code for these applications is available in the `current/Snmp/Manager` and `current/Snmp/Inform` example directories located in the main *examplesDir* (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "17.1 Synchronous Managers" on page 292 shows the simplest way to program an SNMP manager in the Java programming language.
- "17.2 Asynchronous Managers" on page 307 demonstrates the advantages of a manager that does not block when sending requests.
- "17.3 Inform Requests" on page 311 shows how two managers can exchange management information.

17.1 Synchronous Managers

The synchronous SNMP manager is the simplest to program: the manager sends a request to an agent (peer) and waits for the answer. During the wait, the manager is blocked until either a response is received or the timeout period expires.

The SNMP manager API allows two ways of referring to variables when issuing requests:

- By OID (for example, `1.3.6.1.2.1.11.29`)
- By name (`SnmpOutTraps` in this case)

Referring directly to OIDs requires no configuration but makes code less flexible. The advantages of using variable names are simplified coding and the independence of manager code when custom MIBs are modified. The SNMP manager API supports variable names by storing a description of the MIB it accesses in the `SnmpOid` object.

To refer to variable names, the manager needs to initialize this description with an OID table object. The OID table is instantiated from a subclass of the `SnmpOidTableSupport` class generated by the `mibgen` tool when compiling the MIB. Because this support class is regenerated whenever the MIB is recompiled, the new MIB definition is automatically loaded into the manager when it is started (see the Example 17–1).

The SNMP manager API specifies the `SnmpPeer` object for describing an agent, and the `SnmpParameters` object for describing its read-write communities and its protocol version (SNMPv1 or SNMPv2). The objects `SnmpUsmPeer` and `SnmpUsmParameters` perform these roles under SNMPv3, and handle user-based security parameters. The `SnmpSession` is an object for sending requests. The session instance has an `SnmpOptions` field that we can use to set multiplexing and error fixing behavior.

Note – The objects specified by the SNMP manager API are not MBeans and cannot be registered in an MBean server to create a manager that can be controlled remotely. However, you can write an MBean that uses these classes to retrieve and expose information from SNMP agents.

A manager can contain any number of peers, one for each agent it accesses, and any number of sessions, one for each type of behavior it implements. Once the peers and the sessions are initialized, the manager can build lists of variables and send session requests to operate on them. The session returns a request object, and the manager calls its `waitForCompletion` method with the desired timeout delay.

Finally, the manager analyzes the result of the request, first to see if there were any errors, then to extract the data returned by the request.

17.1.1 Synchronous SNMPv1 and SNMPv2 Managers

Example 17–1 shows the code of the `main` method of the `SyncManager` application for SNMPv1 and SNMPv2. It applies all the steps described in the previous section to execute a very simple management operation.

EXAMPLE 17–1 SNMPv1 and SNMPv2 `SyncManager` Example

The `SyncManager` example is found in the *examplesDir*`/current/Snmp/Manager` directory.

```
// read the command line parameters
final String host = argv[0];
final String port = argv[1];

try {
// Specify the OidTable containing all the MIB II knowledge
// Use the OidTable generated by mibgen when compiling MIB II
//
final SnmpOidTableSupport oidTable = new RFC1213_MIBOidTable();
SnmpOid.setSnmpOidTable(oidTable);

final SnmpPeer agent = new SnmpPeer(host, Integer.parseInt(port));

// When creating the parameter object, you can specify the
// read and write community to be used when querying the agent.
final SnmpParameters params = new SnmpParameters("public", "private");
agent.setSnmpParam(params);

final SnmpSession session = new SnmpSession("SyncManager session");

// When invoking a service provided by the SnmpSession, it
// will use the default peer if none is specified explicitly
session.setDefaultPeer(agent);

// Create a listener and dispatcher for SNMP traps:
// SnmpEventReportDispatcher will run as a thread and
// listens for traps in UDP port = agent port + 1
final SnmpEventReportDispatcher trapAgent =
    new SnmpEventReportDispatcher(Integer.parseInt(port)+1,
                          null,taskServer,null);
// TrapListenerImpl will receive a callback
// when a valid trap PDU is received.
final Thread trapThread = new Thread(trapAgent);
    trapThread.setPriority(Thread.MAX_PRIORITY);
    trapThread.start();
// Build the list of variables you want to query.
```

EXAMPLE 17–1 SNMPv1 and SNMPv2 `SyncManager` Example *(Continued)*

```
// For debugging, you can associate a name to your list.
final SnmpVarbindList list= new SnmpVarbindList(
    "SyncManager varbind list");

// We want to read the "sysDescr" variable.
list.addVarBind("sysDescr.0");

// Make the SNMP get request and wait for the result.
SnmpRequest request = session.snmpGet(null, list);
final boolean completed = request.waitForCompletion(10000);

// Check for a timeout of the request.
if (completed == false) {
    java.lang.System.out.println(
"Request timed out. Check if agent can be reached");
    java.lang.System.exit(0);
}

// Check if the response contains an error.
int errorStatus = request.getErrorStatus();
if (errorStatus != SnmpDefinitions.snmpRspNoError) {
    java.lang.System.out.println("Error status = " +
        SnmpRequest.snmpErrorToString(errorStatus));
    java.lang.System.out.println("Error index = " +
        request.getErrorIndex());
    java.lang.System.exit(0);
}

// Now we can extract the content of the result.
final SnmpVarbindList result = request.getResponseVarBindList();
java.lang.System.out.println("Result: \n" + result);

[...] // Wait for user to type enter. Traps will be handled.

// End the session properly and we’re done
session.destroySession();
java.lang.System.exit(0);
```

In this SNMP manager application, we demonstrate how to implement and enable a trap listener for the traps sent by the agent. First we need to instantiate an `SnmpEventReportDispatcher` object. Then we add our listener implementation through its `addTrapListener` method, and finally we start its thread. Trap listeners can be implemented in any manager using the SNMP manager API, not only synchronous managers.

17.1.2 Configuring SNMPv3 Security for Managers

Before you run the SNMPv3 manager examples, you require some information about how SNMPv3 user-based model (USM) security is configured. Below is a brief description of the SNMPv3 security mechanism that provides you with the information you need to run the SNMPv3 examples in this chapter. Full descriptions of the SNMPv3 security mechanisms are given in "19.3 SNMPv3 User-Based Security Model" on page 362.

An SNMPv3 manager requires a security file, in the same way as an SNMPv3 agent does. The `jdmk.security` file for an SNMPv3 manager differs slightly from that of an SNMPv3 agent, as shown in the following example.

EXAMPLE 17–2 A `jdmk.security` File for an SNMPv3 Manager

```
# User and security configuration
userEntry=0x8000002a05819dcb6e00001f95,defaultUser,,
    usmHMACMD5AuthProtocol,mypasswd
userEntry=0x8000002a05819dcb6e00001f96,defaultUser,,
    usmHMACMD5AuthProtocol,mypasswd

# Number of boots
localEngineBoots=5

# Local engine ID
localEngineID=0x8000002a05000000ec4c49ded9
```

In a manager's security file, there is more emphasis on the engine ID than in an agent's security file. The `userEntry` provides all the security information the manager needs to communicate with a particular authoritative agent, as follows:

`0x8000002a05819dcb6e00001f95`	This is the engine ID of the agent with which the manager will communicate
`defaultUser`	The authorized user for that agent
`usmHMACMD5AuthProtocol`	The authentication algorithm; in this case, HMAC MD5
`mypasswd`	The privacy password

In this example, the information in the `userEntry` corresponds to the security information provided in the `AgentV3` example's `jdmk.security` file, in Example 16–3. Therefore, this manager can communicate with that agent.

The remaining information pertains to the manager itself:

`localEngineBoots`	Sets how many times the local engine will boot
`localEngineID`	Represents the ID of the engine associated to the SNMP session in which the manager is running

17.1.3 Synchronous SNMPv3 Managers

The example synchronous manager application created for SNMPv3 is similar to the SNMPv1/v2 manager, except that it implements SNMPv3 user-based USM mechanisms before making requests.

EXAMPLE 17–3 SNMPv3 `SyncManagerV3` Example

The `SyncManagerV3` example is in the *examplesDir*/`current/Snmp/Manager` directory.

```
    //Read the command line parameters
final String host = argv[0];
final String port = argv[1];

try {

      // Initialize the SNMP Manager API.
    final SnmpOidTableSupport oidTable = new RFC1213_MIBOidTable();
    SnmpOid.setSnmpOidTable(oidTable);

      // Build the session.
    //
        try {
                session= new SnmpSession("SyncManagerV3 session");
            }catch(SnmpStatusException e) {
                println(e.getMessage());
                java.lang.System.exit(0);
        }
        catch(IllegalArgumentException e) {
        // If the engine configuration is faulty
        println(e.getMessage());
        java.lang.System.exit(0);
        }

        // Access the SNMPv3 engine using getEngine
        //
           final SnmpEngine engine = session.getEngine();

        // Create an SnmpUsmPeer object
         //
       final SnmpUsmPeer agent =
        new SnmpUsmPeer(engine, host, Integer.parseInt(port));

        // Create USM parameters
         //
       final SnmpUsmParameters p =
        new SnmpUsmParameters(engine, "defaultUser");

        // Set the security level
        //
         p.setSecurityLevel(SnmpDefinitions.authNoPriv);
```

EXAMPLE 17–3 SNMPv3 SyncManagerV3 Example *(Continued)*

```
     // Contextualize the send request
     //
         p.setContextName("TEST-CONTEXT".getBytes());

     // Set the contextEngineId discovered by the peer upon
     // creation
         p.setContextEngineId(agent.getEngineId().getBytes());

     // Associate the parameter with the agent.
     //
     agent.setParams(p);


     // Discover time of creation and boot
    //
    agent.processUsmTimelinessDiscovery();

     // Associate a default peer (agent) to an SnmpSession.
     //
    session.setDefaultPeer(agent);

     // Create a taskServer for processing traps (optional)
     final DaemonTaskServer taskServer = new DaemonTaskServer();
     taskServer.start(Thread.NORM_PRIORITY);

     // Create a listener and dispatcher for SNMP traps
    //
     final SnmpEventReportDispatcher trapAgent =
     new SnmpEventReportDispatcher(engine,
               Integer.parseInt(port) + 1,
               taskServer, null);

     trapAgent.addTrapListener(new TrapListenerImpl());
    final Thread trapThread = new Thread(trapAgent);
     trapThread.setPriority(Thread.MAX_PRIORITY);
        trapThread.start();

     // Build the list of variables you want to query
     //
     final SnmpVarBindList list =
     new SnmpVarBindList("SyncManagerV3 varbind list");

     // Read the "sysDescr" variable
    //
list.addVarBind("sysDescr.0");

     // Make the SNMP get request and wait for the result
     //
     final SnmpRequest request =
     session.snmpGetRequest(null, list);
        println("SyncManagerV3::main:" +
         "Send get request to SNMP agent on " + host +
```

EXAMPLE 17–3 SNMPv3 `SyncManagerV3` Example *(Continued)*

```
                    " at port " + port);
               final boolean completed = request.waitForCompletion(10000);

                // Check for a timeout
                //
            if (completed == false) {
                println("SyncManagerV3::main:" +
                        " Request timed out. Check if agent
                          can be reached");

                // Print request.
                //
                println("Request: " + request.toString());
                java.lang.System.exit(0);
            }

            // Check the response for errors
                //
            final int errorStatus = request.getErrorStatus();
            if (errorStatus != SnmpDefinitions.snmpRspNoError) {
                println("Error status = " +
                     SnmpRequest.snmpErrorToString(errorStatus));
                println("Error index = " +
                     request.getErrorIndex());
                java.lang.System.exit(0);
            }

            // Display the result.
            //
            final SnmpVarBindList result = request.getResponseVarBindList();
            println("Result: \n" + result);

            [...]

            // End the session
            //
            session.destroySession();

                trapAgent.close();
                 taskServer.terminate();
            java.lang.System.exit(0);

                [...]
}
```

The first instantiated session creates an engine. This engine can be accessed using the getEngine method. To avoid excessive engine creation for each instantiated session, the first engine can be shared between SNMP session objects. While sharing is possible, it should be avoided. It represents an unnecessary increase in overhead and limits the security possibilities because only one security file can be associated with an engine.

The engine is used by all the other classes in the application programming interface (API) to access the USM configuration, contained in the jdmk.security file associated with that session. In Example 17–3, when the peer p is created, it discovers its engineId, and then uses it as the SNMPv3 ContextEngineId. When the request is sent, this engine ID is included as a parameter by setContextEngineId.

In this example, the level of security is set as authentication without privacy. Consequently, this level of security is applied to all the requests between this manager and the peers associated with it, via the security parameters. This level of security must match the level specified in the engine's jdmk.security file.

It is also possible to access MIBs that have been registered in the scope of a context (see "16.2.3 Binding the MIB MBeans" on page 270 for details of contextualized MIBs). In this example, the context TEST-CONTEXT is used, and is set as a parameter in the request by setContextName.

Finally, before sending any requests, if authentication is activated, the timeliness parameters of the request are discovered, using processUsmTimelinessDiscovery.

17.1.4 SNMP Trap Handler

A trap handler for the SNMP manager is an object that implements the SnmpTrapListener interface in the com.sun.management.snmp.manager package. When this object is bound as a listener of an SnmpEventReportDispatcher object, its methods are called to handle trap PDUs.

A trap listener is not a notification listener because the dispatcher is not a notification broadcaster. The listener has callback methods that work in the same manner, but they are given objects that represent traps, not instances of the Notification class.

The interface defines three methods, one for processing SNMPv1 traps, another for SNMPv2 traps and a third for SNMPv3 traps. Trap PDU packets have already been decoded by the dispatcher, and these methods handle an object representation of the trap: SnmpPduTrap objects for SNMPv1, SnmpPduRequest objects for SNMPv2 and SnmpScopedPduRequest for SNMPv3.

EXAMPLE 17–4 SnmpTrapListener Implementation

```
public class TrapListenerImpl implements SnmpTrapListener {

        public void processSnmpTrapV1(SnmpPduTrap trap) {
        println("NOTE: TrapListenerImpl received trap :");
        println("\tGeneric " + trap.genericTrap);
        println("\tSpecific " + trap.specificTrap);
        println("\tTimeStamp " + trap.timeStamp);
        println("\tAgent adress " + trap.agentAddr.stringValue());
    }

    public void processSnmpTrapV2(SnmpPduRequest trap) {
        println("NOTE: Trap V2 not of interest !!!");
    }

    public void processSnmpTrapV3(SnmpScopedPduRequest trap) {
            println("NOTE: TrapListenerImpl received trap V3:");
            println("\tContextEngineId : " +
                SnmpEngineId.createEngineId(trap.contextEngineId));
            println("\tContextName : " + new String(trap.contextName));
            println("\tVarBind list :");
            for(int i = 0; i < trap.varBindList.length; i++)
                println("oid : " + trap.varBindList[i].oid + " val : " +
                    trap.varBindList[i].value);

    }

 }
```

▼ To Run the SyncManager Example

1. **In the *examplesDir*/current/Snmp/Manager directory, generate the OID table description of MIB-II that our manager will access.**

   ```
   $ mibgen -mo mib_II.txt
   ```

2. **Compile the example classes.**

   To set up your environment, see “Directories and Classpath” in the Preface.

   ```
   $ javac -classpath classpath -d . *.java
   ```

3. **Make sure that no other agent is running on port 8085, and start the simple SNMP agent in *examplesDir*/current/Snmp/Agent.**

   See “To Run the SNMPv1/v2 Agent Example” on page 273 if you have not already built and run this example.

   Type the following command to start the Agent example:

   ```
   $ cd examplesDir/current/Snmp/Agent
   $ java -classpath classpath Agent nbTraps
   ```

If you are also running the asynchronous manager example with this agent, omit the *nbTraps* parameter. The agent then sends traps continuously and they can be seen in both managers. Otherwise, specify the number of traps to be sent to the manager. Wait until the manager is started to send the traps.

4. **Start the manager application in another window to connect to this agent.**

   If you want to run the manager on a different host, replace `localhost` with the name of the host where you started the agent.

   ```
   $ cd examplesDir/current/Snmp/Manager
   $ java -classpath classpath SyncManager localhost 8085
   SyncManager::main: Send get request to SNMP agent on localhost at port 8085
   Result:
   [Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
   Value : SunOS sparc 5.7]
   ```

   Here we see the output of the SNMP request, it is the value of the `sysDescr` variable on the agent.

5. **Press `Enter` in the agent's window.**

   You should see the manager receiving the traps it sends. Leave the agent running if you are going on to the next example, otherwise remember to stop it by pressing `Control-C`.

▼ To Run the `SyncManagerV3` Example

1. **In the *examplesDir*`/current/Snmp/Manager` directory, generate the OID table description of MIB-II that our manager will access.**

   ```
   $ mibgen -mo mib_II.txt
   ```

2. **Compile the example classes.**

   To set up your environment, see "Directories and Classpath" in the Preface.

   ```
   $ javac -classpath classpath -d . *.java
   ```

3. **Make sure that no other agent is running on port 8085, and start the simple SNMPv3 agent, `AgentV3`, in *examplesDir*`/current/Snmp/Agent`.**

   See "To Run the SMNPv3 `AgentV3` Example" on page 274 if you have not already built and run this example.

   Type the following commands to start the `AgentV3` example:

   ```
   $ cd examplesDir/current/Snmp/Agent
   $ java -classpath classpath -Djdmk.security.file=jdmk.security AgentV3
   ```

4. **Start the manager application in another window to connect to this agent.**

   If you want to run the manager on a different host, replace `localhost` with the name of the host where you started the agent.

```
$ cd examplesDir/current/Snmp/Manager
$ java -classpath classpath -Djdmk.security.file=jdmk.security
SyncManagerV3 localhost 8085
```

Be sure to run the manager in its directory, otherwise it cannot find its `jdmk.security` file.

5. **Press `Enter` in the agent's window.**

   You should see the manager receiving the traps it sends. Stop the manager by typing `Control-C`.

17.1.5 Synchronous Managers Accessing Several Agents

The SNMP API enables you to configure a synchronous SNMPv3 manager that can access several agents. The `SyncManagerMultiV3` example demonstrates a simple SNMPv3 manager API that makes requests on two SNMPv3 agents. For the sake of simplicity, this manager does not listen for traps.

EXAMPLE 17–5 SNMPv3 `SyncManagerMultiV3` Example

```
//Check host and port arguments
//
    final String host1 = argv[0];
    final String port1 = argv[1];
          final String host2 = argv[2];
    final String port2 = argv[3];

       // Initialize the SNMP Manager API.
        //
        final SnmpOidTableSupport oidTable = new RFC1213_MIBOidTable();
        SnmpOid.setSnmpOidTable(oidTable);

        [...]

            // Build the session.
            //
                try {
            // When instantiating a session, a new SNMP V3 engine is
            // instantiated.
            //
                session= new SnmpSession("SyncManagerMultiV3 session");
            }catch(SnmpStatusException e) {
                println(e.getMessage());
                java.lang.System.exit(0);
           }
            catch(IllegalArgumentException e) {
            // If the engine configuration is faulty
            //
            println(e.getMessage());
```

EXAMPLE 17–5 SNMPv3 `SyncManagerMultiV3` Example *(Continued)*

```
                java.lang.System.exit(0);
        }

                // Get the SnmpEngine.
                //
                final SnmpEngine engine = session.getEngine();

                    // Create a SnmpPeer object for representing the first
                 // entity to communicate with.
                //
                 final SnmpUsmPeer agent1 =
                 new SnmpUsmPeer(engine, host1, Integer.parseInt(port1));

                 //Create the second peer.
                 //
                   final SnmpUsmPeer agent2 =
                new SnmpUsmPeer(engine, host2, Integer.parseInt(port2));

                // Create parameters to associate to the entity to
                // communicate with.
                //
                final SnmpUsmParameters p =
                new SnmpUsmParameters(engine, "defaultUser");

               // Set security level to authNoPriv.
               //
                p.setSecurityLevel(SnmpDefinitions.authNoPriv);

                // Contextualize the send request
               //
                p.setContextName("TEST-CONTEXT".getBytes());

                // Specify a contextEngineId
                   //
                p.setContextEngineId(agent1.getEngineId().getBytes());

                // Associate the newly created parameter to the agent
                //
                agent1.setParams(p);


                // Discover timeliness and boot
                agent1.processUsmTimelinessDiscovery();


                // Build the list of variables you want to query
                //
                final SnmpVarBindList list =
                new SnmpVarBindList("SyncManagerMultiV3 varbind list");

                // Read the "sysDescr" variable
                //
```

EXAMPLE 17–5 SNMPv3 SyncManagerMultiV3 Example *(Continued)*

```
   list.addVarBind("sysDescr.0");

    // Make the SNMP get request on the first agent agent1
    // and wait for the result
    //
    SnmpRequest request =
    session.snmpGetRequest(agent1, null, list);
    println("SyncManagerMultiV3::main:" +
        " Send get request to SNMP agent on " +
        host1 + " at port " + port1);
    boolean completed = request.waitForCompletion(10000);

    // Check for a timeout
    //
if (completed == false) {
    println("SyncManagerMultiV3::main:" +
            " Request timed out. Check if agent can
            be reached");

    // Print request
    //
    println("Request: " + request.toString());
    java.lang.System.exit(0);
}

// Check if the response contains an error
     //
int errorStatus = request.getErrorStatus();
if (errorStatus != SnmpDefinitions.snmpRspNoError) {
    println("Error status = " +
            SnmpRequest.snmpErrorToString(errorStatus));
    println("Error index = " +
            request.getErrorIndex());
    java.lang.System.exit(0);
}

// Display the content of the result
//
SnmpVarBindList result = request.getResponseVarBindList();
println("Result: \n" + result);

println("\n>> Press Enter if you want to send the request" +
" on the second agent.\n");
java.lang.System.in.read();



   // Repeat the process for the second agent agent2.
   //
    SnmpUsmParameters p2 =
    new SnmpUsmParameters(engine, "defaultUser");
```

EXAMPLE 17–5 SNMPv3 `SyncManagerMultiV3` Example *(Continued)*

```
     p2.setSecurityLevel(SnmpDefinitions.authNoPriv);

        p2.setContextName("TEST-CONTEXT".getBytes());

    p2.setContextEngineId(agent2.getEngineId().getBytes());

    // Associate the updated parameters with agent2.
    //
   agent2.setParams(p2);

     // Discover timeliness and boot
    //
     agent2.processUsmTimelinessDiscovery();

       // Make the request with agent2
    //
     request = session.snmpGetRequest(agent2, null, list);
     println("SyncManagerMultiV3::main:" +
             " Send get request to SNMP agent on " +
         host2 + " at port " + port2);
       completed = request.waitForCompletion(10000);

     // Check for a timeout
     //
 if (completed == false) {
     println("SyncManagerMultiV3::main:" +
         " Request timed out. Check  if agent can be
         reached");

     // Print request.
     //
     println("Request: " + request.toString());
     java.lang.System.exit(0);
 }

// Check if the response contains an error
     //
     errorStatus = request.getErrorStatus();
         if (errorStatus != SnmpDefinitions.snmpRspNoError) {
         println("Error status = " +
         SnmpRequest.snmpErrorToString(errorStatus));
         println("Error index = " +
     request.getErrorIndex());
     java.lang.System.exit(0);
 }

 // Display the content of the result
 //
     result = request.getResponseVarBindList();
 println("Result: \n" + result);

 println("\n>> Press Enter if you want to stop " +
```

EXAMPLE 17–5 SNMPv3 `SyncManagerMultiV3` Example *(Continued)*

```
                "this manager.\n");
        java.lang.System.in.read();

        // End the session
        //
        session.destroySession();


}
```

The `SyncManagerMultiV3` example essentially performs the same actions as the `SyncManagerV3` example, except it creates two SNMP USM peers rather than just one. The two peer agents are each created in exactly the same way as in the single peer example.

Both the peer agents are accessed using a common set of parameters, which are kept in the `jdmk.security` file for the session. This `jdmk.security` file contains two rows, one for each SNMP USM peer agent, as shown in the following example.

EXAMPLE 17–6 `jdmk.security` File for the `SyncManagerMultiV3` Example

```
#Authentication only.
userEntry=0x8000002a05819dcb6e00001f95,defaultUser,,usmHMACMD5AuthProtocol,
mypasswd
userEntry=0x8000002a05819dcb6e00001f96,defaultUser,,usmHMACMD5AuthProtocol,
mypasswd

# #####APPENDED PROPERTY####
localEngineBoots=5

# #####APPENDED PROPERTY####
localEngineID=0x8000002a05000000ec4c49ded9
```

If you configure your manager to create a greater number of peers, then its associated `jdmk.security` file must contain a corresponding number of entries, one for each authoritative engine with which the manager will communicate.

In this example, the only difference between the two `userEntry` rows is between the engine ID numbers. These engine IDs correspond to the engines of the two SNMP adaptor servers created by the `MultipleAgentV3` example in "16.5 Multiple Agents" on page 287.

▼ To Run the `SyncManagerMultiV3` Example

1. **Before running the example, you must have run `mibgen` and compiled the Java classes in the *examplesDir*`/current/Snmp/Manager` directory.**

See "To Run the SyncManager Example" on page 300 for instructions if you have not already done this.

2. **Make sure that no other agent is running, and start the multiple SNMPv3 agent, MultipleAgentV3, in *examplesDir*/current/Snmp/Agent.**

   See "16.5.1 Running the SNMPv3 MultipleAgentV3 Example" on page 290 if you have not already built and run this example.

   Type the following commands to start the MultipleAgentV3 example:

   ```
   $ cd examplesDir/current/Snmp/Agent
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   MultipleAgentV3
   ```

   The MultipleAgentV3 example simulates two SNMPv3 agents in one process. We shall make these agents peers of SyncManagerMultiV3.

   Be sure to run the multiple agent in its directory, otherwise it cannot find its jdmk.security and jdmk2.security files.

3. **Start the manager application in another window to connect to these two agents.**

   If you want to run the manager on a different host from the one where the agents are running, replace localhost with the name of the host where you started the agents. Both the agents in the MultipleAgentV3 example run on the same host.

   ```
   $ cd examplesDir/current/Snmp/Manager
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   SyncManagerMultiV3 localhost 8085 localhost 8087
   ```

   Be sure to run the manager in its directory, otherwise it cannot find its jdmk.security file.

4. **You should see sending a request to the first agent. Press Enter to send a request to the second agent.**

   You will now see the manager sending a second request to the agent on port 8087.

5. **Press Enter to stop the manager**

17.2 Asynchronous Managers

The asynchronous SNMP manager lets you handle more requests in the same amount of time because the manager is not blocked waiting for responses. Instead, it creates a request handler object that runs as a separate thread and processes several responses concurrently. Otherwise, the initialization of peers, parameters, sessions, options, and dispatcher is identical to that of a synchronous manager. This applies to all three versions of SNMP.

EXAMPLE 17–7 The AsyncManager Example

```
// read the command line parameters
String host = argv[0];
String port = argv[1];

// Use the OidTable generated by mibgen when compiling MIB-II.
final SnmpOidTableSupport oidTable = new RFC1213_MIBOidTable();

// Sample use of the OidTable.
SnmpOidRecord record = oidTable.resolveVarName("udpLocalPort");
java.lang.System.out.println(
    "AsyncManager::main: variable = " + record.getName() +
        " oid = " + record.getOid() + " type = " + record.getType());

// Initialize the SNMP Manager API.
SnmpOid.setSnmpOidTable(oidTable);

// Create an SnmpPeer object for representing the agent
final SnmpPeer agent = new SnmpPeer(host, Integer.parseInt(port));

// Create parameters for communicating with the agent
final SnmpParameters params = new SnmpParameters("public", "private");
agent.setSnmpParam(params);

// Build the session and assign its default peer
final SnmpSession session = new SnmpSession("AsyncManager session");
session.setDefaultPeer(agent);

    final DaemonTaskServer taskServer = new DaemonTaskServer();
    taskServer.start(Thread.NORM_PRIORITY);

// Same dispatcher and trap listener as in SyncManager example
SnmpEventReportDispatcher trapAgent =
            new SnmpEventReportDispatcher(Integer.parseInt(port)+1,
                          null,taskServer,null);
trapAgent.addTrapListener(new TrapListenerImpl());
final Thread trapThread = new Thread(trapAgent);
    trapThread.setPriority(Thread.MAX_PRIORITY);
    trapThread.start();

// Build the list of variables to query
SnmpVarbindList list = new SnmpVarbindList("AsyncManager varbind list");
list.addVariable("sysDescr.0");

// Create a simple implementation of an SnmpHandler.
AsyncRspHandler handler = new AsyncRspHandler();

// Make the SNMP walk request with our handler
final SnmpRequest request = session.snmpWalkUntil(
    handler, list, new  SnmpOid("sysServices"));

// Here you could do whatever processing you need.
// In the context of the example, we are just going to wait
```

EXAMPLE 17–7 The `AsyncManager` Example *(Continued)*

```
// 4 seconds while the response handler displays the result.
Thread.sleep(4000);

[...] // Wait for user to type enter. Traps will be handled.

// End the session properly and we're done.
//
session.destroySession();
java.lang.System.exit(0);
```

The trap mechanism in this application is identical to the one presented in the `SyncManager` example. The event report dispatcher receives traps and calls the corresponding method of our `SnmpTrapListener` class.

In this example, the manager performs an `snmpWalkUntil` request that gives a response for each variable that it gets. The response handler is called to process each of these responses.

17.2.1 Response Handler

A response handler for an asynchronous manager is an implementation of the `SnmpHandler` interface. When a handler object is associated with a request, its methods are called when the agent returns an answer or fails to return an answer. In these methods, you implement whatever actions you want for processing the responses to a request. Typically, these methods extract the result of each request or the reason for its failure.

The timeout used by the request handler is the one specified by the `SnmpPeer` object representing the agent. The handler is also called to process any errors caused by the request in the session. This ensures that the manager application is never interrupted after issuing a request.

EXAMPLE 17–8 The `SnmpHandler` Implementation

```
public class AsyncRspHandler implements SnmpHandler {

    // Empty constructor
    public AsyncRspHandler() {
    }

    // Called when the agent responds to a request
    public void processSnmpPollData( SnmpRequest request,
        int errStatus, int errIndex, SnmpVarbindList vblist) {

         // Check if a result is available.
        if (request.getRequestStatus() ==
```

EXAMPLE 17–8 The `SnmpHandler` Implementation *(Continued)*

```
       SnmpVarBindList result = request.getResponseVarBindList();
       println("Result = " + result.varBindListToString());

      // Extract the result for display
      SnmpVarbindList result = request.getResponseVbList();
      java.lang.System.out.println(
          "Result = " + result.vbListToString());
    }
  }

  // Called when the agent fails to respond to a request
  public void processSnmpPollTimeout(SnmpRequest request) {

    java.lang.System.out.println(
        "Request timed out: " + request.toString());

    if (request.getRequestStatus() ==
        SnmpRequest.stResultsAvailable) {

        // The result is empty and will display an error message
        SnmpVarbindList result = request.getResponseVbList();
        java.lang.System.out.println(
            "Result = " + result.vbListToString());
    }
  }

  // Called when there is an error in the session
  public void processSnmpInternalError(SnmpRequest request,
      String errmsg) {

    java.lang.System.out.println(
        "Session error: " + request.toString());
    java.lang.System.out.println("Error is: " + errmsg);
  }
}
```

▼ To Run the `AsyncManager` Example

1. **If you have not done so already, start the simple SNMP agent in *examplesDir*`/current/Snmp/Agent`, after making sure that no other agent is running on port 8085.**

   This manager also uses the OID table description (the `SnmpOidTableSupport` class) that we generated from the MIB for the synchronous manager. If you have not already done so, see "To Run the `SyncManager` Example" on page 300 for instructions on how to do this.

2. **If you do not have an SNMP agent still running, make sure that no other agent is running on port 8085 and start one with the following command:**

```
$ cd examplesDir/current/Snmp/Agent
$ java -classpath classpath Agent nbTraps
```

3. **Specify the number of traps to be sent to the manager in the *nbTraps* parameter.**

   Wait until the manager is started to send the traps.

4. **In another terminal window, start the manager application to connect to this agent.**

   If you want to run the manager on a different host, replace `localhost` with the name of the host where you started the agent.

   ```
   $ cd examplesDir/current/Snmp/Manager
   $ java -classpath classpath AsyncManager localhost 8085
   ```

   You should then see the output of the `SnmpWalkUntil` request: the response handler method is called for each variable that is returned.

5. **Press `Enter` in the agent's window to send traps and see the trap reports as they are received in the manager.**

   When you have finished with the agent, do not forget to stop it by typing `Control-C` in its terminal window.

17.3 Inform Requests

The inform request is specified in SNMPv2 and SNMPv3 as a mechanism for sending a report and receiving a response.

Because SNMP managers both send and receive inform requests, the SNMP manager API includes the mechanisms for doing both. Roughly, inform requests are sent in the same way as other requests, and they are received in the same way as traps. Both of these mechanisms are explained in the following sections.

There are simple examples, one for SMNPv2 and one for SNMPv3, in *examplesDir*/`current/Snmp/Inform`. Each example has two manager applications, one of which sends an inform request, and the other which listens for and replies to this request. No SNMP agents are involved in these exchanges.

17.3.1 Sending an Inform Request (SNMPv2)

Like the other types of requests, the manager sends an inform request through a session. The only difference is that the peer object associated with the request should be an SNMP manager able to receive and reply to `InformRequest` PDUs.

You can associate a peer with a session by making it the default peer object. This is how we do it in this example. This means that if we do not specify a peer when sending requests, they are automatically addressed to our manager peer. Because sessions often have agent peers as a default, you can specify the manager peer as a parameter to the `snmpInform` method of the session object.

EXAMPLE 17–9 Sending an SNMPv2 Inform Request in `SimpleManager1`

```
// When calling the program, you must specify the hostname
// of the SNMP manager you want to send the inform to.
//
final String host = argv[0];

// Initialize the port number to send inform PDUs on port 8085.
//
final int port = 8085;

try {
    // Create an SnmpPeer object that represents the entity to
    // communicate with.
    //
    final SnmpPeer peer = new SnmpPeer(host, port);

    // Create parameters to associate to the peer.
    // When creating the parameter object, you can specify the
    // read and write community to be used when sending an
    // inform request.
    //
    final SnmpParameters params = new SnmpParameters(
        "public", "private", "public");

    // The newly created parameter must be associated to the peer.
    //
    peer.setSnmpParam(params);

    // Build the session. A session creates, controls and manages
    // one or more requests.
    //
    final SnmpSession session = new SnmpSession("SimpleManager1
                                               session");
    session.setDefaultPeer(peer);

    // Make the SNMP inform request and wait for the result.
    //
    final SnmpRequest request = session.snmpInform(
        null, new SnmpOid("1.2.3.4"), null);
    java.lang.System.out.println(
        "NOTE: Inform request sent to SNMP manager on " +
        host + " at port " + port);
    boolean completed = request.waitForCompletion(10000);

    // Check for a timeout of the request.
    //
    if (completed == false) {
```

EXAMPLE 17–9 Sending an SNMPv2 Inform Request in `SimpleManager1` *(Continued)*

```
        java.lang.System.out.println(
            "\nSimpleManager1::main: Request timed out. " +
            "Check if agent can be reached");

        // Print request.
        //
        java.lang.System.out.println("Request: " + request.toString());
        java.lang.System.exit(0);
    }

    // Now we have a response. Check if the response contains an error.
    //
    final int errorStatus = request.getErrorStatus();
    if (errorStatus != SnmpDefinitions.snmpRspNoError) {
        java.lang.System.out.println("Error status = " +
            SnmpRequest.snmpErrorToString(errorStatus));
        java.lang.System.out.println("Error index = " +
            request.getErrorIndex());
        java.lang.System.exit(0);
    }

    // Now we shall display the content of the result.
    //
    final SnmpVarbindList result = request.getResponseVbList();
    java.lang.System.out.println("\nNOTE: Response received:\n" + result);

    // Stop the session properly before exiting
    session.destroySession();
    java.lang.System.exit(0);

} catch(Exception e) {
    java.lang.System.err.println(
        "SimpleManager1::main: Exception occurred:" + e );
    e.printStackTrace();
}
```

Before sending the request, the `snmpInform` method automatically adds two variables to the head of the `varbind` list that is passed in as the last parameter. These are `sysUpTime.0` and `snmpTrapOid.0`, in the order they appear in the list. These variables are mandated by RFC 1905 and added systematically so that the calling process does not need to add them.

Like all other requests in a session, inform requests can be handled either synchronously or asynchronously in the sender. In our example, we process the inform request synchronously: the manager blocks the session while waiting for the completion of the request. In an asynchronous manager, you would need to implement a response handler as explained in “17.2.1 Response Handler” on page 309, and then use it to process responses, as shown in Example 17–7.

17.3.2 Sending an Inform Request (SNMPv3)

This example shows how to build an SNMPv3 manager that sends and/or receives SNMPv3 requests. It initializes an SNMPv3 USM peer and an SNMP session, and then sends an inform request to a second SNMP manager.

EXAMPLE 17–10 Sending an SNMPv3 Inform Request in `SimpleManager1V3`

```
            //    When calling the program, specify the hostname of the
            // SNMP manager to send the inform to
            //
            final String host = argv[0];

        // Initialize the port number to send inform PDUs on port 8085.
        //
        final int port = 8085;

            [...]

            // Create an SnmpUSMPeer object for representing the entity
            // to communicate with.
        //
        final SnmpUsmPeer peer = new SnmpUsmPeer(session.getEngine(),
            host,
            port);

        // Create parameters to associate to the entity to
          // communicate with.
       // When creating the parameter object, you specify the necessary
          // security related parameters.
        //
        final  SnmpUsmParameters params =
   new SnmpUsmParameters(session.getEngine());

            // Set the parameters
           //

               // First a principal
            //
           params.setPrincipal("defaultUser");

          // A security level. Authentication is applied.
          //
          params.setSecurityLevel(SnmpDefinitions.authNoPriv);

          // Add a contextEngineId. The context engine Id is the
          // manager's engine ID and not the adaptor's.
          //
          params.setContextEngineId(peer.getEngineId().getBytes());

          // Add a context name.
          //
          params.setContextName("TEST-CONTEXT".getBytes());
```

```
// The newly created parameter must be associated to the peer.
//
peer.setParams(params);


   // The inform is authenticated, so the timeliness
   // discovery must be processed.
   //
   peer.processUsmTimelinessDiscovery();



// A default peer (listener manager) can be associated to an
   // SnmpSession. When invoking a service provided by the
   // SnmpSession, if the peer is not specified, the session
   // will perform the service using the default one as the
   // target of the service
//
session.setDefaultPeer(peer);

println("\nNOTE: SNMP V3 simple manager 1 initialized");

// Make the SNMP inform request and wait for the result.
//
println("\n>> Press Enter to send the inform request on " +
   host + " at port " + port + "...");
 try {
         System.in.read();
     } catch (IOException e) {
         e.printStackTrace();
     }

     final SnmpRequest request =
  session.snmpInformRequest(null, new SnmpOid("1.2.3.4"),
               null);

     println("NOTE: Inform request sent to SNMP manager on " +
     host + " at port " + port);

     final boolean completed = request.waitForCompletion(10000);

  // Check for a timeout of the request.
  //
  if (completed == false) {
         println("\nSimpleManager1::main: Request timed out." +
     " Check reachability of agent");

         // Print request.
         //
         println("Request: " + request.toString());
         java.lang.System.exit(0);
```

EXAMPLE 17–10 Sending an SNMPv3 Inform Request in `SimpleManager1V3` *(Continued)*

```
            }

         // Check if the response contains an error
         //
         final int errorStatus = request.getErrorStatus();
         if (errorStatus != SnmpDefinitions.snmpRspNoError) {
                println("Error status = " +
             SnmpRequest.snmpErrorToString(errorStatus));
                println("Error index = " + request.getErrorIndex());
                java.lang.System.exit(0);
            }

            // Display the content of the result.
            //
            final SnmpVarBindList result = request.getResponseVarBindList();
            println("\nNOTE: Response received:\n" + result);

            // End the session
            //
            println("\nNOTE: SNMP V3 simple manager 1 stopped...");
            session.destroySession();


}
```

As you can see, once the SNMPv3 session has been instantiated and the SNMPv3 USM peer has been created and configured, the inform request is sent in exactly the same way as under SNMPv2. The only difference is the additional SNMPv3 security.

17.3.3 Receiving Inform Requests

Managers receive inform requests as they do traps: they are unsolicited events that must be received by a dispatcher object. Unlike traps, an inform request requires a response PDU that, according to the SNMP specification, must contain the same variable bindings. Therefore, immediately after an inform request is successfully received and decoded, the `SnmpEventReportDispatcher` class automatically constructs and sends the inform response back to the originating host.

The manager application then retrieves the data in the inform request through a listener on the dispatcher. Inform request listeners are registered with the dispatcher object in the same way as trap listeners. The receiving manager in our example is very simple, because its only function is to create the dispatcher and the listener for inform requests. The receiving manager `SimpleManager2` is the same for both the `SimpleManager1` and `SimpleManager1V3` examples.

EXAMPLE 17–11 Receiving Inform Requests in `SimpleManager2`

```
// Initialize the port number to listen for incoming inform PDUs on
// port 8085.
int port = 8085;

try {

    // Create a dispatcher for SNMP event reports
        // (SnmpEventReportDispatcher).
    // SnmpEventReportDispatcher is run as a thread and listens for informs
    // on the specified port.
    // Add our InformListenerImpl class as an SnmpInformListener.
    // InformListenerImpl will receive a callback when a valid trap
    // PDU is received.
    //

        final DaemonTaskServer taskServer = new DaemonTaskServer();
        taskServer.start(Thread.NORM_PRIORITY);

    final SnmpEventReportDispatcher informDispatcher =
        new SnmpEventReportDispatcher(port,taskServer,null,null);
    informDispatcher.addInformListener(new InformListenerImpl());
       final Thread informThread = new Thread(informDispatcher);
       informThread.setPriority(Thread.MAX_PRIORITY);
    informThread.start();
        println("\nNOTE: SNMP simple manager 2 initialized");

    // Note that you can use the same SnmpEventReportDispatcher object
    // for both incoming traps and informs.
    // Just add your trap listener to the same dispatcher, for example:
    //    informDispatcher.addTrapListener(new TrapListenerImpl());

    // Here we are just going to wait for inform PDUs.
    //
    java.lang.System.out.println("\nNOTE: Event report listener
       initialized");
    java.lang.System.out.println(
      "      and listening for incoming inform PDUs on port " + port
        + "...");

} catch(Exception e) {
    java.lang.System.err.println(
        "SimpleManager2::main: Exception occurred:" + e );
    e.printStackTrace();
}
```

The remaining step is to program the behavior we want upon receiving an inform request. To do this, we must write the `InformListenerImpl` class that we registered as an inform request listener in the previous code sample. This class implements the `SnmpInformListener` interface and its `processSnmpInform` and `processSnmpInformV3` methods handle the incoming inform requests.

Because the dispatcher automatically sends the inform response back to the originating host, the SnmpInformListener implementation does not need to do this. Usually this method extracts the variable bindings and takes whatever action is necessary upon receiving an inform request. In our example, we simply print out the source and the contents of the inform request.

EXAMPLE 17–12 The InformListenerImpl Class

```
public class InformListenerImpl implements SnmpInformListener {

    public void processSnmpInform(SnmpPduRequest inform) {

        // Display the received PDU.
        //
        java.lang.System.out.println("\nNOTE: Inform request received:\n");
        java.lang.System.out.println("\tType
             = " + inform.pduTypeToString(inform.type));
        java.lang.System.out.println("\tVersion   = " + inform.version);
        java.lang.System.out.println("\tRequestId = " + inform.requestId);
        java.lang.System.out.println("\tAddress   = " + inform.address);
        java.lang.System.out.println("\tPort      = " + inform.port);
        java.lang.System.out.println("\tCommunity = " +
            new String(inform.community));
        java.lang.System.out.println("\tVB list   = ");

        for (int i = 0; i < inform.varBindList.length; i++) {
            java.lang.System.out.println("\t\t" + inform.varBindList[i]);
        }

    }
            public void processSnmpInformV3(SnmpScopedPduRequest inform) {
                   println("\nNOTE: Inform request V3 received:\n");
                println("\tType      =
                " + inform.pduTypeToString(inform.type));
               println("\tVersion   = " + inform.version);
                  println("\tRequestId = " + inform.requestId);
                  println("\tAddress   = " + inform.address);
                println("\tPort      = " + inform.port);
                println("\tContext = " + new String(inform.contextName));
                println("\tVB list   = ");
                for (int i = 0; i < inform.varBindList.length; i++) {
                    println("\t\t" + inform.varBindList[i]);
        }
    }

     private final static void println(String msg) {
           java.lang.System.out.println(msg);
     }

     private final static void print(String msg) {
           java.lang.System.out.print(msg);
     }
}
```

▼ To Run the SNMPv2 Inform Request Example

The *examplesDir*/current/Snmp/Inform directory contains all of the files for the two manager applications, along with the InformListenerImpl class.

1. **Compile all files in this directory with the javac command.**

   For example, on the Solaris platform with the Korn shell, you would type:

   ```
   $ cd examplesDir/current/Snmp/Inform/
   $ javac -classpath classpath *.java
   ```

2. **To run the example, start the inform request receiver with the following command.**

   ```
   $ java -classpath classpath SimpleManager2
   ```

   You can start the application in another terminal window or on another host.

3. **Wait for this manager to be initialized, then start the other manager with the following command.**

   The *hostname* is the name of the host where you started the receiving manager, or **localhost**.

   ```
   $ java -classpath classpath SimpleManager1 hostname
   ```

4. **When the sender is ready, press Enter to send the inform request.**

   You should see the contents of the request displayed by the receiving manager. Immediately afterwards, the sender receives the inform response containing the same variable bindings and displays them. Both manager applications then exit automatically.

▼ To Run the SNMPv3 Inform Request Example

1. **Before running the example, you must compile the Java classes in the *examplesDir*/current/Snmp/Inform directory.**

   Type the following commands in your working directory:

   ```
   $ javac -d . SimpleManager1V3.java SimpleManager2.java
   InformListenerImpl.java
   ```

2. **Start the receiving manager, SimpleManager2:**

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   SimpleManager2
   ```

   This manager binds to port 8085.

3. **Wait for this manager to be initialized, then start the other manager with the following command, specifying the hostname and port number of the receiving manager.**

In this case, the host name is the localhost

```
$ java -classpath classpath -Djdmk.security.file=sender.security
SimpleManager1V3 localhost
```

4. **When the sender is ready, press `Enter` to send the inform request.**

   You should see the contents of the request displayed by the receiving manager. Immediately afterwards, the sender receives the inform response containing the same variable bindings and displays them. Both manager applications then exit automatically.

CHAPTER 18

Advanced MIB Implementations

This chapter covers the more advanced implementations of the Simple Network Management Protocol (SNMP) that you can create using Java Dynamic Management Kit (Java DMK). These advanced implementations focus on SNMP tables, and are presented in three examples of increasing complexity in the following sections.

- "18.1 Simple SNMP Tables" on page 321 presents the most straightforward implementation of SNMP tables using Java DMK.
- "18.2 SNMP Table Instrumentation" on page 326 shows how to expose and manipulate the content of an MBean server using SNMP tables.
- "18.3 Virtual SNMP Tables" on page 344 shows how to expose and manipulate the content of an MBean server using virtual SNMP tables.

18.1 Simple SNMP Tables

SNMP allows for the structuring of Management Information Base (MIB) variables into logical tables, in which the variables are organized into rows and columns. Java DMK allows for the dynamic addition of rows to the tables that conform to the `RowStatus` convention defined by SNMPv2. Java DMK also allows you to change the values of existing rows. An example of the use of tables in the Java DMK SNMP API is found in the directory *examplesDir*`/current/Snmp/Rowstatus`.

This example shows how to create rows remotely in an SNMP agent from an SNMP manager application, in tables that follow the `RowStatus` convention. It does this by enabling remote entry creation in the tables and by sending the appropriate requests to create or destroy rows in the table. The `RowStatus` example also shows how to replace the generated skeletons for SNMP groups, tables, and entries with your own customized classes, so that the customized entry class is instantiated when a remote SNMP manager requests the creation of a row in a table managed by a `RowStatus` variable.

This example is built around a small SNMP MIB called DEMO-MIB which contains one group, Demo, and one table, TableDemoTable. Before you can proceed with the example, you need to generate this MIB and its associated classes, by using the mibgen compiler supplied with Java DMK. For details of the mibgen compiler, see the *Java Dynamic Management Kit 5.1 Tools Reference Guide*.

▼ To Generate the DEMO-MIB

The example MIB is contained in the configuration file mib_demo.txt, which is found in *examplesDir*/current/Snmp/Rowstatus. You should run the example from within this directory.

1. **Ensure that your PATH environment variable knows where to find mibgen.**

   In a default installation, mibgen is found in *installDir*/bin.

2. **Run the mibgen compiler.**

   ```
   $ mibgen -d . mib_demo.txt
   ```

   You see that mibgen has generated the following Java classes.

   - Demo, which implements the Demo group.
   - DemoEntry, which defines an MBean that provides variables to serve as entries in the table, and specifies the operations that can be performed on this MBean.
   - DemoEntryMBean, which represents the remote management interface for the DemoEntry MBean.
   - DemoEntryMeta, which constructs the SNMP metadata for the DemoEntry MBean.
   - DemoMBean, which represents the remote management interface for the Demo MBean.
   - DemoMeta, which constructs the SNMP metadata for the DemoEntry MBean.
   - DEMO_MIB, which defines the behavior of this SNMP MIB.
   - DEMO_MIBOidTable, which contains the metadata definitions for DEMO-MIB.
   - DemoTableMeta, which constructs the metadata definitions for the SNMP table.
   - EnumDemoTableRowStatus, which provides an enumerated integer value that informs you of the status of a table row.
   - TableDemoTable, which constructs the SNMP table, making each row from DemoEntry objects using SNMP SET operations, and then registers the rows in the Java DMK MBean server.

To be able to modify the classes dynamically, we must take the classes generated by the mibgen compiler and extend them accordingly in a series of custom classes. These customized classes are provided in the *examplesDir*/current/Snmp/Rowstatus directory.

EXAMPLE 18–1 Adding Entries to Tables

```
public class TableDemoTableImpl extends TableDemoTable {
    public TableDemoTableImpl(SnmpMib myMib, DemoImpl myGroup) {
        super(myMib);
           this.myGroup = myGroup;
    }
    public TableDemoTableImpl(SnmpMib myMib, MBeanServer server,
                  DemoImpl myGroup) {
       super(myMib,server);
           this.myGroup = myGroup;
    }

    public void removeEntryCb(int pos, SnmpOid row, ObjectName name,
                Object entry, SnmpMibTable meta)
            throws SnmpStatusException {
       super.removeEntryCb(pos,row,name,entry,meta);
          myGroup.removeRowCb();
    }
    public void addEntryCb(int pos, SnmpOid row, ObjectName name,
                     Object entry, SnmpMibTable meta)
              throws SnmpStatusException {
          super.addEntryCb(pos,row,name,entry,meta);
          myGroup.addRowCb();
    }
    public Object createDemoEntryMBean(SnmpMibSubRequest req,
                SnmpOid rowOid, int depth, ObjectName entryObjName,
                SnmpMibTable meta, Integer  aDemoTableIndex)
            throws SnmpStatusException  {

        DemoEntryImpl entry = new DemoEntryImpl(theMib);
        entry.DemoTableIndex = aDemoTableIndex;
        return entry;
    }
    private DemoImpl myGroup;

}
```

The `TableDemoTableImpl` class shown in Example 18–1 subclasses the `TableDemoTable` class generated by `mibgen` from the `mib_demo.txt` file. It customizes the `SnmpTableSupport` `addEntryCb` and `removeEntryCb` methods, adding new functionality not originally provided by `TableDemoTable`. The `addEntryCb` and `removeEntryCb` callback methods are called from the SNMP runtime when a row is added to or removed from the table. The `removedRowCb` and `addRowCb` methods are callback methods that maintain a local row counter, `DemoInteger`. However, the fact that they maintain a row counter is merely for the sake of this example. The `addEntryCb` and `removeEntryCb` callback methods could perform any operation the developer chooses to implement, such as emitting notifications, for example.

The most important method of the TableDemoTableImpl class is createDemoEntryMBean. The createDemoEntryMBean method inTableDemoTableImpl overrides the one defined by TableDemoTable to implement DemoEntryImpl instead of the skeletal DemoEntry.

The DemoImpl group, shown in Example 18–2, overrides the constructors defined by the Demo group, to allow remote creation of rows in a DemoTable.

EXAMPLE 18–2 Permitting Remote Creation of Rows

```
public class DemoImpl extends Demo {
    public DemoImpl(SnmpMib myMib) {
           super(myMib);
            DemoTable = new TableDemoTableImpl(myMib, this);
       DemoTable.setCreationEnabled(true);
    }
    public DemoImpl(SnmpMib myMib, MBeanServer server) {
            super(myMib,server);
        DemoTable = new TableDemoTableImpl(myMib, server, this);
            DemoTable.setCreationEnabled(true);
    }
    public void addRowCb() {
            DemoInteger = new Integer(DemoInteger.intValue() + 1);
    }
    public void removeRowCb() {
            DemoInteger =  new Integer(DemoInteger.intValue() - 1);
    }
}
```

As you can see, as well as setting the SnmpTableSupport method setCreationEnabled value to true, DemoImpl defines the addRowCb and removeRowCb callback methods called by TableDemoTableImpl. For each row added or deleted, addRowCb and removeRowCb update the value of the DemoInteger counter accordingly. DemoInteger is defined by the Demo class, and counts the number of entries in the demoTable.

EXAMPLE 18–3 Adding and Deleting Rows from a Remote SNMP Manager

```
public class Manager {

[...]
        try {
            SnmpOidTableSupport oidTable = new DEMO_MIBOidTable();
            SnmpOid.setSnmpOidTable(oidTable);
            [..]
            EnumRowStatus   createAndGo = new EnumRowStatus("createAndGo");
            EnumRowStatus   destroy     = new EnumRowStatus("destroy");

            SnmpVarBindList setList1 =
                new SnmpVarBindList("Manager SET varbind list");
            SnmpVarBind stringRow1 =
                new SnmpVarBind(new SnmpOid("demoTableString.1"),
                                new SnmpString("This is row 1"));
            SnmpVarBind statusRow1 =
                new SnmpVarBind(new SnmpOid("demoTableRowStatus.1"),
                                createAndGo.toSnmpValue());

            setList1.addVarBind(stringRow1);
            setList1.addVarBind(statusRow1);

            request = session.snmpSetRequest(null, setList1);
            completed = request.waitForCompletion(10000);
            [...]
            result = request.getResponseVarBindList();

            SnmpVarBindList getList =
                new SnmpVarBindList("Manager GET varbind list");
            getList.addVarBind("demoInteger.0");
            getList.addVarBind("demoTableString.1");
            getList.addVarBind("demoTableRowStatus.1");
            request = session.snmpGetRequest(null, getList);
            completed = request.waitForCompletion(10000);
            [...]
            result = request.getResponseVarBindList();

            SnmpVarBindList setList2 =
                new SnmpVarBindList("Manager SET varbind list");
            SnmpVarBind statusRow2 =
                new SnmpVarBind(new SnmpOid("demoTableRowStatus.1"),
                                destroy.toSnmpValue());

            setList2.addVarBind(statusRow2);
            request = session.snmpSetRequest(null, setList2);
            completed = request.waitForCompletion(10000);
            [...]
            result = request.getResponseVarBindList();
            request = session.snmpGetRequest(null, getList);
            completed = request.waitForCompletion(10000);
            result = request.getResponseVarBindList();
            [...]
            session.destroySession();
            java.lang.System.exit(0);
        }
        [...]
    }
```

The `Manager` begins by loading the `DEMO_MIBOidTable` table created by `mibgen`. After setting up the SNMP peer, session and parameters, the `Manager` defines the creation and deletion of rows. The table in this example supports the `RowStatus` variable. The values for columns can be changed by sending SNMP `set` requests to the index of the row that is to be added or removed. Setting the value of the column that contains the `RowStatus` variable to `createAndGo` creates and activates a row for which a value is provided by the set request. Setting the value of the `RowStatus` column of an existing row you want to delete to `destroy`, deletes that row.

▼ To Run the SNMP Table RowStatus Example

As stated in "To Generate the DEMO-MIB" on page 322, you must have run mibgen on the mib_demo.txt file before proceeding with this example. Run the example inside the *examplesDir*/current/Snmp/Rowstatus directory.

1. **Compile the Java classes.**

   ```
   $ javac -d . *.java
   ```

2. **Start the Agent.**

   Make sure that no agents are already running before you start the Agent.

   ```
   $ java Agent
   ```

   You see confirmation of the creation of the HTML adaptor on port 8082 and the SNMP adaptor on port 8085, and the addition of the DEMO-MIB to the MBean server. You can perform management operations on the Agent using the HTML adaptor, by loading the URL http://localhost:8082/ in a browser.

3. **In another terminal, start the Manager.**

   ```
   $ java Manager agent-hostname 8085
   ```

   Where *agent-hostname* is the name of the machine where the Agent is running. 8085 represents the port number on which the SNMP adaptor is running.

   When you start the Manager, you will be prompted to press Enter to perform the get and set operations to add and delete rows from the demoTable.

18.2 SNMP Table Instrumentation

The example shown in "18.1 Simple SNMP Tables" on page 321 showed simply how to use Java DMK to add and remove rows from an SNMP table. The example presented in this section goes further, and demonstrates how to add instrumentation to an SNMP table.

This example is based on the classes in the *examplesDir*/current/Snmp/MBeanTable directory. It defines a MIB called JMX-MBEAN-SERVER-MIB that exposes the content of an MBean server through SNMP. To achieve this, JMX-MBEAN-SERVER-MIB defines two SNMP tables:

- jmxMBeanTable, which has read-create access permission, and contains one row per MBean registered in the remote MBean server.
- jmxMBeanAttrTable, which has read-only access permission, and extends the jmxMBeanTable. For each MBean mirrored in the jmxMBeanTable, it contains one additional row per attribute exposed by this MBean.

18.2.1 Classes Generated by mibgen

Before you can proceed with this example, you need to generate the JMX-MBEAN-SERVER-MIB and it's associated classes, by using the mibgen compiler supplied with Java DMK. For details of the mibgen compiler, see the *Java Dynamic Management Kit 5.1 Tools Reference Guide*.

▼ To Generate the JMX-MBEAN-SERVER-MIB

The example MIB is contained in the configuration file JMX-MBEAN-SERVER-MIB.mib, which is found in *examplesDir*/current/Snmp/MBeanTable. You should run the example from within this directory.

1. **Ensure that your PATH environment variable knows where to find mibgen.**

   In a default installation, mibgen is found in *installDir*/bin.

2. **Create a new directory, generated, inside *examplesDir*/current/Snmp/MBeanTable.**

   ```
   $ mkdir generated
   ```

   This directory is where mibgen generates the classes associated with JMX-MBEAN-SERVER-MIB.mib.

3. **Run the mibgen compiler.**

   ```
   $ mibgen -X:use-display-hint -d generated JMX-MBEAN-SERVER-MIB.mib
   ```

   The advanced mibgen option use-display-hint instructs mibgen to generate an attribute of type String for any object using a textual convention whose DISPLAY-HINT is 255a. This option is used because JMX-MBEAN-SERVER-MIB defines textual conventions (for example, JmxMBeanObjectNameTC) which must be translated into java.lang.String attributes, rather than the default Byte[] attributes.

   The -d generated option sends the output of mibgen to the newly created generated directory

Within the generated directory, you see that mibgen has generated the following Java classes.

- EnumJmxMBeanRowStatus, an enumerated type generated for the jmxMBeanRowStatus columnar variable defined in the MIB.
- JmxMBeanAttrEntry, which implements the JmxMBeanAttrEntryMBean interface and is generated from the jmxMBeanAttrEntry conceptual row defined in the MIB. The JmxMBeanAttrEntry is an empty skeleton class that can be subclassed to implement a row in the jmxMBeanAttrTable defined in the MIB. This example shows how this generated class can be subclassed to provide an instrumented implementation, JmxMBeanAttrEntryImpl, that is linked to the actual resource exposed.

- JmxMBeanAttrEntryMBean, which represents the public interface for the jmxMBeanAttrEntry conceptual row. An object implementing the JmxMBeanAttrEntryMBean interface represents a row in the jmxMBeanAttrTable, that is defined in the MIB.
- JmxMBeanAttrEntryMeta, which constructs the SNMP metadata for the JmxMBeanAttrEntry conceptual row, that is defined in the MIB.
- JmxMBeanAttrTableMeta, which constructs the SNMP metadata for the jmxMBeanAttrTable table, that is defined in the MIB.
- JmxMBeanEntry, which implements the JmxMBeanEntryMBean interface and is generated from the jmxMBeanEntry conceptual row defined in the MIB. The JmxMBeanEntry is an empty skeleton class that can be subclassed to implement a row in the jmxMBeanTable defined in the MIB. This example shows how this generated class can be subclassed in order to provide an instrumented implementation, JmxMBeanEntryImpl. that is linked to the actual resource exposed.
- JmxMBeanEntryMBean, which represents the public interface for the jmxMBeanEntry conceptual row. An object implementing the JmxMBeanEntryMBean interface represents a row in the jmxMBeanTable, that is defined in the MIB.
- JmxMBeanEntryMeta, which constructs the SNMP metadata for the jmxMBeanEntry conceptual row defined in the MIB.
- JmxMBeanServer, which represents the jmxMBeanServer group that is defined in the MIB. The JmxMBeanServer is an empty skeleton class that can be subclassed to implement the jmxMBeanServer group. This default implementation of the jmxMBeanServer group instantiates the generated default implementation for the two SNMP tables defined in that group:

  - TableJmxMBeanAttrTable for the jmxMBeanAttrTable.
  - TableJmxMBeanTable for the jmxMBeanTable.

  This example shows how the generated class JmxMBeanServer, can be subclassed by JmxMBeanServerImpl, to instantiate specialized generated table classes that instantiate JmxMBeanAttrEntryImpl and JmxMBeanEntryImpl objects instead of the generated JmxMBeanAttrEntry and JmxMBeanEntry skeletons.
- JmxMBeanServerMBean, which represents the public interface for the object that represents the jmxMBeanServer group.
- JmxMBeanServerMeta, which constructs the SNMP metadata for the jmxMBeanServer group, that is defined in the MIB.
- JMX_MBEAN_SERVER_MIB, which instantiates the metadata and instrumentation classes for all the objects defined in the JMX-MBEAN-SERVER-MIB module. This example shows how JMX_MBEAN_SERVER_MIB_Impl subclasses the generated JMX_MBEAN_SERVER_MIB, to instantiate the customized class JmxMBeanServerImpl instead of the generated skeletons.

- JMX_MBEAN_SERVER_MIBOidTable, which is a convenience class generated by mibgen to make it possible to map symbolic OID definitions from the JMX-MBEAN-SERVER-MIB. For example, jmxMBeanAttrName is mapped to the corresponding OID value, 1.3.6.1.4.1.42.2.145.1.3.1.1.1.1.4.1.2.
- JmxMBeanTableMeta, which constructs the metadata definitions for the jmxMBeanTable table defined in the MIB.
- TableJmxMBeanAttrTable, which is the default implementation of the jmxMBeanAttrTable table defined in the MIB. By default, the table is empty.
- TableJmxMBeanTable, which is the default implementation of the jmxMBeanTable table defined in the MIB. By default, the table is empty. If remote creation of rows through SNMP is enabled, the default implementation creates instances of the generated JmxMBeanEntry empty skeleton. The JmxMBeanServerImpl class provided in this example shows how to instantiate a specialized version of this class that creates customized JmxMBeanEntryImpl objects instead of the default implementation.

18.2.2 Customized Classes

As was the case for the simple SNMP table example, to be able to tie the code generated by mibgen to its proper instrumentation, you must extend the classes generated by mibgen in a series of custom classes. The customized classes are those provided in the *examplesDir*/current/Snmp/MBeanTable directory.

The customized extensions to the generated classes add the following functionality:

- JMX_MBEAN_SERVER_MIB_Impl extends the generated JMX_MBEAN_SERVER_MIB, instantiating JmxMBeanServerImpl instead of the generated JmxMBeanServer skeleton. In addition, JMX_MBEAN_SERVER_MIB_Impl adds a new method, start(), that when called starts mirroring the content of the MBean server into the MIB.

  The start() method instructs the JmxMBeanServerImpl group to populate the jmxMBeanTable and jmxMBeanAttrTable according to the initial content of the MBean server mirrored by this MIB. The start() method also instructs the JmxMBeanServerImpl group to start listening for MBeanServerNotifications coming from the MBean server.
- JmxMBeanServerImpl extends the JmxMBeanServer skeleton, and instantiates a subclass of TableJmxMBeanTable, in which the MBeans registered in the MBean server are mirrored. The TableJmxMBeanTable object is subclassed to support remote creation and deletion of entries in the jmxMBeanTable through SNMP.
  - An SNMP SET request that creates a new row in the jmxMBeanTable is transformed into a createMBean() call to the MBean server.
  - An SNMP SET request deleting an existing row in the jmxMBeanTable is transformed in an unregisterMBean() call to the MBean server.

The JmxMBeanServerImpl also listens for notifications of the type MBeanServerNotification to keep the jmxMBeanTable updated, and creates and deletes rows in the jmxMBeanAttrTable accordingly.

- JmxMBeanAttrEntryImpl extends the generated JmxMBeanAttrEntry, and is used to mirror an attribute of an MBean that is itself mirrored in the jmxMBeanTable. A JmxMBeanAttrEntryImpl object adds instrumentation to a conceptual row in the jmxMBeanAttrTable. When a new row is added to jmxMBeanTable, the corresponding JmxMBeanEntryImpl updates the jmxMBeanAttrTable by adding one row per attribute exposed by the mirrored MBean. When a row is deleted from the jmxMBeanTable, the JmxMBeanEntryImpl deletes the corresponding rows from the jmxMBeanAttrTable. The jmxMBeanAttrTable is thus managed entirely internally by the agent. Its instances cannot be created or deleted by a remote SNMP manager, except as a side effect of the creation or deletion of a row in the jmxMBeanTable.
- JmxMBeanEntryImpl extends the generated JmxMBeanEntry skeleton. JmxMBeanEntryImpl is used to mirror MBeans that are registered in the MBean server that is made accessible through SNMP by the JMX_MBEAN_SERVER_MIB_Impl MIB. A JmxMBeanEntryImpl object adds instrumentation to a conceptual row in the jmxMBeanTable. JmxMBeanEntryImpl objects can be created or deleted as a result of SNMP SET requests, or as a result of an MBean being locally registered or deregistered from the MBean server mirrored by the MIB. The JmxMBeanServerImpl object keeps the jmxMBeanTable updated, and thus manages the life cycle of the JmxMBeanEntryImpl objects. The JmxMBeanEntryImpl objects keep the jmxMBeanAttrTable updated, and thus manage the life cycle of the JmxMBeanAttrEntryImpl objects.

Aspects of the JmxMBeanServerImpl and JmxMBeanEntryImpl classes are examined in more detail in the following sections.

18.2.2.1 JmxMBeanServerImpl

As stated previously, JmxMBeanServerImpl extends the JmxMBeanServer empty skeleton class. JmxMBeanServerImpl also instantiates a subclass of TableJmxMBeanTable, to mirror the MBeans registered in an MBean server.

EXAMPLE 18–4 Subclassing TableJmxMBeanTable

```
[...]
    private final void inittables() {

       JmxMBeanTable = new TableJmxMBeanTable(myMib,null) {
             public Object createJmxMBeanEntryMBean(SnmpMibSubRequest req,
                                                    SnmpOid rowOid,
                                                    int depth,
                                                    ObjectName entryObjName,
                                                    SnmpMibTable meta,
```

EXAMPLE 18–4 Subclassing `TableJmxMBeanTable` *(Continued)*

```
                                                    Integer  aJmxMBeanIndex)
                 throws SnmpStatusException  {

                 return JmxMBeanServerImpl.this.
                    createJmxMBeanEntryMBean(req,rowOid,depth,entryObjName,
                                                 meta,aJmxMBeanIndex);
                }

                public void removeEntryCb(int pos, SnmpOid row,
                                          ObjectName name,
                                          Object entry, SnmpMibTable meta)
                    throws SnmpStatusException {
                    super.removeEntryCb(pos,row,name,entry,meta);
                    final JmxMBeanEntryMBean e =
                        (JmxMBeanEntryMBean) entry;
                    final EnumJmxMBeanRowStatus destroy =
                         new EnumJmxMBeanRowStatus(EnumRowStatus.destroy);
                    e.setJmxMBeanRowStatus(destroy);
                }
            };

        JmxMBeanTable.setCreationEnabled(true);
    }

[...]
```

In Example 18–4, `JmxMBeanServerImpl` defines a method, `inittables`, to instantiate and initialize its table objects. The `inittables` method creates a new instance of `TableJmxMBeanTable`, and overrides it so that its `createJmxMBeanEntryMBean` method returns a `JmxMBeanEntryImpl` object instead of the default `JmxMBeanEntry` object. This is done by defining a new `createJmxMBeanEntryMBean` method in the customized `JmxMBeanEntryImpl` class (see Example 18–5), and calling it on the parent `JmxMBeanServerImpl` object from the overridden `TableJmxMBeanTable` object, as shown in Example 18–4.

The `removeEntryCb` method is overridden, so that table entries representing MBeans also delete their associated entries in the `jmxMBeanAtrrTable` by calling `setJmxMBeanRowStatus(destroy)` when they are removed from the `jmxMBeanTable`.

EXAMPLE 18–5 `createJmxMBeanEntryMBean` Method

```
[...]
    private Object createJmxMBeanEntryMBean(SnmpMibSubRequest req,
                                                SnmpOid rowOid,
                                                int depth,
                                                ObjectName entryObjName,
                                                SnmpMibTable meta,
                                                Integer  aJmxMBeanIndex)
             throws SnmpStatusException  {
```

EXAMPLE 18–5 `createJmxMBeanEntryMBean` Method *(Continued)*

```
              SnmpVarBind rowStatusVb = req.getRowStatusVarBind();

             if (rowStatusVb == null)
                 throw new SnmpStatusException(
                            SnmpStatusException.snmpRspNoCreation);

              if (! isAvailable(aJmxMBeanIndex.intValue())) {
                 if (Boolean.getBoolean("info"))
                 System.out.println("Index is not suitable: "
                     + aJmxMBeanIndex);
                 throw new SnmpStatusException(
                            SnmpStatusException.snmpRspInconsistentValue);
             }

              JmxMBeanEntryImpl entry =
                 new JmxMBeanEntryImpl(myMib,this);
             entry.JmxMBeanIndex = aJmxMBeanIndex;

              entry.createFromRemote(req,rowStatusVb);
             return entry;
    }

[...]
```

As shown in Example 18–4, the `createJmxMBeanEntryMBean` method is called when a remote SNMP Manager creates a new row in a `jmxMBeanTable`. Example 18–5 shows how the `createJmxMBeanEntryMBean` method is overridden by the `JmxMBeanServerImpl` class, to add new entries in the `jmxMBeanAttrTable` as the rows are created in the table.

Of the parameters the customized version of `createJmxMBeanEntryMBean` takes when it is started, the following are the most significant..

- `req`, the `SnmpMibSubRequest` containing the `VarBind` list pertaining to this new row.
- `rowOid`, the index of this new row's OID.
- `meta`, the `SnmpMibTable` metadata object of the `jmxMBeanTable`.
- `aJmxMBeanIndex`, the value of the new entry's `JmxMBeanIndex` index variable.

The `createJmxMBeanEntryMBean` method checks first of all whether the `VarBind` list found when `req` calls `getRowStatusVarBind()` is valid. It then checks whether the row's OID index is available. Once it has established that both the VarBind and the index are viable, it proceeds to create a `JmxMBeanEntryImpl` instance, `entry`. Calling `createFromRemote()` at this point ensures that `createMBean()` will be called when the new entry is eventually activated.

As mentioned previously, `JmxMBeanServerImpl` is also configured to listen for notifications of the type `MBeanServerNotifications` coming from the MBean server.

EXAMPLE 18–6 Listening for Notifications

```
[...]
public synchronized void start() {

      started = true;
      try {
             final ObjectName delegate =
               new ObjectName("JMImplementation:type=MBeanServerDelegate");
           myMib.getMibServer().
               addNotificationListener(delegate,mbsListener,null,null);

           try {
                   Set mbeans = myMib.getMibServer().queryNames(null,null);
            for (final Iterator it = mbeans.iterator();
                it.hasNext();) {
                     final ObjectName name = (ObjectName) it.next();
                     if (mustShow(name)) showMBean("start",name);
                 }
            } catch (Exception x) {
                 try {
                     myMib.getMibServer().
                          removeNotificationListener(delegate,mbsListener);
                 } catch (Exception e) { /* OK */ }
                 throw x;
            }
        } catch (Exception x) {
            started = false;
            System.err.println("Failed to start MBean Table: " + x);
            if (Boolean.getBoolean("debug")) x.printStackTrace();
        }
}

[...]
```

The `start()` method shown in Example 18–5 starts a notification listener in the MIB server that listens out for notifications from the `MBeanServerDelegate`. It also populates the `jmxMBeanTable` and `jmxMBeanAttrTable` with the initial content of the MBean server. These tables are then updated as and when notifications are received, as shown in Example 18–7.

EXAMPLE 18–7 Handling Notifications

```
[...]

private void handleMBSNotification(Notification notif, Object handback) {

      synchronized(this) { if (!started) return; }
```

EXAMPLE 18–7 Handling Notifications *(Continued)*

```
        if (notif instanceof MBeanServerNotification) {
            final MBeanServerNotification n =
                  (MBeanServerNotification) notif;
            final String nt = n.getType();
            final ObjectName mbeanName = n.getMBeanName();

            if (MBeanServerNotification.REGISTRATION_NOTIFICATION.
                 equals(nt)) {

             synchronized (this) {
                 if (mustShow(mbeanName)) {
                      showMBean(nt,mbeanName);
                 }
             }
         } else if (MBeanServerNotification.UNREGISTRATION_NOTIFICATION.
                  equals(nt)) {

             synchronized (this) {
                if (mustHide(mbeanName)) {
                     hideMBean(nt,mbeanName);
                }
            }
         }
      }
}

[...]
```

The `handleMBSNotification` method that `JmxMBeanServerImpl` defines begins by checking that the correct type of notification has been received. If the notification is of the type `MBeanServerNotification.REGISTRATION_NOTIFICATION`, then a method named `showMBean()` is called. Otherwise, if `MBeanServerNotification.UNREGISTRATION_NOTIFICATION` is received, then the MBean is hidden with a call to `hideMBean`. The `showMBean()` method is shown in Example 18–8.

EXAMPLE 18–8 Exposing MBeans through the SNMP Tables

```
[...]

private void showMBean(String operation,ObjectName name,int index) {
       try {
           JmxMBeanEntry entry = new JmxMBeanEntryImpl(myMib,this);
           entry.JmxMBeanIndex = new Integer(index);
           entry.setJmxMBeanObjectName(name.toString());
           JmxMBeanTable.addEntry(entry);
           names.put(name, entry);
           entry.setJmxMBeanRowStatus(
                   new EnumJmxMBeanRowStatus(EnumRowStatus.active));
           if (Boolean.getBoolean("info"))
                System.out.println("ADDED: JmxMBeanTable["+index+"]="+name);
```

EXAMPLE 18–8 Exposing MBeans through the SNMP Tables *(Continued)*

```
       } catch (Exception x) {
           System.err.println("Failed to add MBean entry: " + name);
           if (Boolean.getBoolean("debug")) x.printStackTrace();
       }
}

[...]
```

The `showMBean` method adds the MBeans that `handleMBSNotification` learns about to the SNMP table `JmxMBeanTable` as `JmxMBeanEntryImpl` row entries. The status of the row is set to `active`. The MBean discovered by the notification listener is thus entered in the SNMP table as a row entry.

18.2.2.2 `JmxMBeanEntryImpl`

As mentioned previously, the `JmxMBeanEntryImpl` objects keep the `jmxMBeanAttrTable` updated. Example 18–9 and Example 18–10 show how `JmxMBeanEntryImpl` changes the status of rows, activates rows, and destroys them.

EXAMPLE 18–9 Changing the Row Status

```
[...]

public synchronized void setJmxMBeanRowStatus(EnumJmxMBeanRowStatus x)
       throws SnmpStatusException {
       switch (x.intValue()) {
       case EnumRowStatus.active:
           if (! (JmxMBeanRowStatus.intValue() == EnumRowStatus.active))
                activate();
           break;
       case EnumRowStatus.notReady:
           break;
       case EnumRowStatus.notInService:
           super.setJmxMBeanRowStatus(x);
           break;
       case EnumRowStatus.destroy:
           destroy();
           break;
       default:
           throw new SnmpStatusException(SnmpStatusException.
                                               snmpRspInconsistentValue);
       }
}

[...]
```

The `setJmxMBeanRowStatus` method shown above is called by the SNMP runtime when the row is created remotely. It is also called explicitly by `JmxMBeanEntryImpl` when the row is created after receiving an MBean server notification.

EXAMPLE 18–10 Activating and Destroying Rows

```
private void activate() throws SnmpStatusException {
       if ((remoteCreation) && (mbean == null || name == null))
           throw new SnmpStatusException(SnmpStatusException.snmpRspGenErr);
       try {

            if (remoteCreation) {
                 myGroup.registerMBean(mbean,name,this);
                 initJmxMBean(name);
                 remoteCreation = false;
                 mbean = null;
            }

            exposedAttrCount = myGroup.addAttributes(name,this);
            if (Boolean.getBoolean("info"))
                 System.out.println(name.toString()+ ": added " +
                                      exposedAttrCount
                                      + " attribute(s).");
            JmxMBeanRowStatus =
                 new EnumJmxMBeanRowStatus(EnumRowStatus.active);
       } catch (Exception x) {
            SnmpStatusException sn =
                 new SnmpStatusException(SnmpStatusException.snmpRspGenErr);
            sn.initCause(x);
            throw sn;
       }
}

private void destroy() throws SnmpStatusException {
      try {
           JmxMBeanRowStatus =
                new EnumJmxMBeanRowStatus(EnumRowStatus.notInService);
           if (name != null && remoteDeletion)
                myGroup.unregisterMBean(name,this);
           remoteCreation = false;
           mbean = null;
           if (name != null)
                myGroup.removeAttributes(name,this);
           attrList = null;
       } catch (Exception x) {
           SnmpStatusException sn =
                new SnmpStatusException(SnmpStatusException.snmpRspGenErr);
           sn.initCause(x);
           throw sn;
       }
}
```

Example 18–10 shows the `activate` and `destroy` methods defined by `JmxMBeanEntryImpl`. The `activate` method verifies first of all whether the row was created remotely or locally, and acts differently depending on the answer. If the row was created remotely by calling the `createFromRemote` method, the row entry registers the requested MBean in the MBean. If the row is created locally, this means that the request to create the row was made by an existing registered MBean, so there is no need to register it again in the MBean server.

The `destroy` method is created when the row is removed, either locally by `handleMBeanServerNotification` or remotely. The `destroy` method is called by `removeEntryCb`, which itself is called when the row is removed from the table, whether remotely or locally.

18.2.3 Point of Entry into the SNMP Table Instrumentation Example

In this example, the table instrumentation operations shown in the preceding sections are all demonstrated by a single class, the `Agent`. The fact that this `Agent` class creates MBeans, and registers them in an MBean server so that they can be mirrored by the SNMP MIB, is irrelevant to this example. The real purpose of the `Agent` is to demonstrate *how* to add instrumentation to the SNMP tables using Java DMK technology, not to demonstrate what the example actually *does*.

EXAMPLE 18–11 Making an MBean Server Accessible by an SNMP Manager

```
public class Agent {
    public interface SimpleMBean {
           public String getName();
           public int    getCount();
           public void   reset();
    }

    public static class Simple implements SimpleMBean {
           public Simple() {
               this(null);
           }
    // Define MBean operations
    [...]
    }

    public static MBeanServer getPlatformMBeanServer() {
            final MBeanServer test = MBeanServerFactory.createMBeanServer();
            final MBeanServer first = (MBeanServer)
               MBeanServerFactory.findMBeanServer(null).get(0);
            if (test != first) MBeanServerFactory.releaseMBeanServer(test);
            return first;
    }

    public int populate(MBeanServer server, int mbeanCount) {
          int count = 0;
          for (int i=0; i<mbeanCount; i++) {
              try {
                    final SimpleMBean simple = new Simple();
                    final ObjectName name =
```

EXAMPLE 18–11 Making an MBean Server Accessible by an SNMP Manager *(Continued)*

```
                       new ObjectName("Example:type=Simple,name="+
                                         simple.getName());
                    server.registerMBean(simple,name);
                    count ++;
                    System.out.println("Registered MBean: " + name);
              } catch (Exception x) {
                    System.err.println("Failed to register MBean: " + x);
                    debug(x);
              }
          }
          return count;
    }

    public int depopulate(MBeanServer server, int mbeanCount) {
          int count = 0;
          while (true) {
              try {
                   final ObjectName pattern =
                      new ObjectName("Example:type=Simple,*");
                   Set mbeans = server.queryNames(pattern,null);
                   if (mbeans.isEmpty()) break;
                   for (Iterator it=mbeans.iterator(); it.hasNext() ; ) {
                        if (count == mbeanCount) return count;
                        final ObjectName next = (ObjectName) it.next();
                        try {
                              server.unregisterMBean(next);
                              count++;
                        System.out.println("Deregistered MBean: "
                                             + next);
                        } catch (InstanceNotFoundException x) {
                           continue;
                        }
                   }
                   if (count >= mbeanCount) break;
              } catch (Exception x) {
                   System.err.println("Unexpected exception: " + x);
                   debug(x);
                   break;
              }
          }
          return count;
   }
[...]
```

The `Agent` class shown in Example 18–11 defines a basic MBean called `SimpleMBean`, that performs very basic MBean operations. The `getPlatformMBeanServer` method is used to obtain the first MBean server that has been created in this session by the `MBeanServerFactory`.

EXAMPLE 18–12 Creating an SNMPv3 Adaptor Server and `JMX-MBEAN-SERVER-MIB`

```
[...]

    public void startmib(MBeanServer server,int port) throws IOException {
          final SnmpV3AdaptorServer adaptor =
              new SnmpV3AdaptorServer((InetAddressAcl)null,port);
          adaptor.enableSnmpV1V2SetRequest();
          adaptor.start();
          do {
              adaptor.waitState(CommunicatorServer.ONLINE,1000);
          } while (adaptor.getState() == CommunicatorServer.STARTING);

          final int state = adaptor.getState();
          if (state != CommunicatorServer.ONLINE) {
              try { adaptor.stop(); } catch (Exception x) { /* OK */ }
              throw new IOException("Can't start adaptor: " +
                                        adaptor.getStateString());
          }

       JMX_MBEAN_SERVER_MIB_Impl mib =
              new JMX_MBEAN_SERVER_MIB_Impl(server);

          try {
               mib.init();
               mib.setSnmpAdaptor(adaptor);
               server.registerMBean(adaptor,null);
               mib.start();
          } catch (Exception x) {
               System.err.println("Failed to register SnmpAdaptor: " + x);
               try { adaptor.stop(); } catch (Exception e) { /* OK */ }
               final IOException io =
                    new IOException("Failed to register SnmpAdaptor");
               io.initCause(x);
               throw io;
          }
       System.out.println("SnmpAdaptor ready at port: "
                          + adaptor.getPort());
    }
[...]
```

The `Agent` class then defines a method, `startmib`, to instantiate an SNMPv3 adaptor server named `adaptor`, and creates an instance of the `JMX_MBEAN_SERVER_MIB_Impl` MIB, named `mib`. The `mib` is constructed around an MBean server named `server`, the content of which is mirrored in this MIB. The MIB is then initialized by a call to the `init()` method defined by `JMX_MBEAN_SERVER_MIB_Impl`. The reference to the SNMP protocol adaptor through which the MIB is accessible, in this case `adaptor`, is set by calling the `SnmpMibAgent` method `setSnmpAdaptor`, and this adaptor is then registered as an MBean in the MBean server `server`. The `start()` method defined by `JMX_MBEAN_SERVER_MIB_Impl` is then called. The `start()` method starts the mbeanServerGroup, which itself is an instance of JmxMBeanServerImpl, and which activates the mirroring of the MBean server in the MIB. See the JMX_MBEAN_SERVER_MIB_Impl.java file in the generated directory for the full implementation of the start() method.

EXAMPLE 18–13 Starting the Agent

```
[...]
      public void start(int port, String[] args) throws IOException {
           final MBeanServer server =  getPlatformMBeanServer();
           final JMXServiceURL[] urls = new JMXServiceURL[args.length];
           for (int i=0;i<args.length;i++) {
                try {
                      urls[i] = new JMXServiceURL(args[i]);
                } catch (MalformedURLException x) {
                           throw x;
                }
           }
           for (int i=0;i<urls.length;i++) {
               try {
                    final JMXConnectorServer s =
                            JMXConnectorServerFactory.
                       newJMXConnectorServer(urls[i],
                                  null,null);
                    final ObjectName name =
                  new ObjectName("Connector:type=
                                 "+s.getClass().getName()+
                                        ",instance="+i);
                    server.registerMBean(s,name);
                    s.start();
                   } catch (Exception x) {
                    final String msg = "Failed to start connector:
                                  " + args[i];
                    System.err.println(msg);
                    final IOException io = new IOException(msg);
                    io.initCause(x);
                    throw io;
                  }
           }

           populate(server,5);

           startmib(server,port);

           final Timer timer = new Timer();
           final long  period= 2000;
           final long  now=0;
           final int   max=4;
           for (int i=0;i<max;i++) {
               final TimerTask taskAdd = new TimerTask() {
                        public void run() {
                       final MBeanServer server =
                                getPlatformMBeanServer();
                             populate(server,1);
                        }
```

EXAMPLE 18–13 Starting the Agent *(Continued)*

```
                };
            final TimerTask taskRem = new TimerTask() {
                    public void run() {
                   final MBeanServer server =
                           getPlatformMBeanServer();
                         depopulate(server,1);
                    }
                };
            timer.schedule(taskAdd,now+period*i,2*period*max);
            timer.schedule(taskRem,now+period*max+period*i,2*period*max);
        }
    }

    public static void initOidTable() {
        final SnmpOidTable orig = SnmpOid.getSnmpOidTable();
        SnmpOidDatabaseSupport mibs = new SnmpOidDatabaseSupport();
        mibs.add(new JMX_MBEAN_SERVER_MIBOidTable());
        if (orig != null) mibs.add(orig);
        SnmpOid.setSnmpOidTable(mibs);
    }
[...]
```

Agent now defines a `start()` method of its own. The `start` method firstly obtains an MBean server instance, `server`, by calling the `getPlatformMBeanServer` method defined earlier in the class. It also retrieves any JMX service URLs that might be passed to `Agent` as arguments when it is launched. If any JMX service URLs are provided at start-time, the `Agent.start()` method uses them to create `JMXConnectorServer` instances that it registers in the MBean server `server` as MBeans.

The MBean server instance `server` is populated with dummy MBeans by calling the `populate()` method. These MBeans are mirrored in the MIB, when the mirroring is activated.

The SNMP adaptor and the MIB are started by calling `startmib`, that was defined in Example 18–12.

Once the `startmib` method has been called, timer tasks are defined that periodically add and remove MBeans from the MBean server. These periodic tasks serve merely to test and demonstrate that the `jmxMBeanTable` and `jmxMBeanAttrTable` are updated accordingly in the MIB.

`JMX_MBEAN_SERVER_MIBOidTable` metadata definitions for the MIBs are loaded into the global SNMP OID table, to enable the use of symbolic OID names.

EXAMPLE 18–14 Agent `main()` Method

```
[...]
    public static void main(String[] args) {
```

EXAMPLE 18–14 Agent `main()` Method *(Continued)*

```
        Agent agent = new Agent();
        try {
             initOidTable();
          final String portStr = System.getProperty("snmp.port",
                                                     "16161");
             final int port;
             try {
                   port = Integer.parseInt(portStr);
             } catch (Exception x) {
              System.err.println("Invalid value specified for
                                  snmp.port: "
                                        + portStr);
                 System.err.println("Error is: " + x);
                 throw x;
             }
             System.out.println("Using SNMP port " + port);

             agent.start(port,args);

             // Wait forever...
             Thread.sleep(Long.MAX_VALUE);
        } catch (Exception x) {
             System.err.println("Failed to start agent: " + x);
             debug(x);
             System.exit(1);
        }

    }

}
```

Finally, an instance of `Agent`, `agent`, is started on the appropriate SNMP port by calling the `start` method defined in Example 18–13.

18.2.4 Running the SNMP Table Instrumentation Example

After you have run `mibgen` to generate `JMX_MBEAN_SERVER_MIB` and its associated classes, as explained in "18.2.1 Classes Generated by `mibgen`" on page 327, you can run the example.

▼ To Run the SNMP Table Instrumentation Example

Run the example from the *examplesDir*`/current/Snmp/MBeanTable` directory.

1. **Compile the Java classes in the `generated` directory.**

```
$ cd generated
$ javac -d .. *.java
```

This creates the compiled `*.class` files into the *examplesDir*/`current/Snmp/MBeanTable` directory, rather than in `generated`, so that they are accessible to the other classes used in the example.

2. **Compile the Java classes in the *examplesDir*`/current/Snmp/MBeanTable` directory.**

```
$ cd ..
$ javac -d . *.java
```

3. **Start the `Agent`.**

   You have several options when starting the `Agent` class.

   - To start the `Agent` on the default SNMP port, 16161:

   ```
   $ java  Agent service:jmx:jmxmp://
   ```

   - To start the `Agent` on a port of your choice:

   ```
   $ java  -Dsnmp.port=port_number Agent service:jmx:jmxmp://
   ```

   - To start the `Agent` with a connector of your choosing, you can specify a different service URL and connector port. For example:

   ```
   $ java  Agent JMX_service_URL
   ```

   In any of the above cases, you see confirmation of the SNMP port used, confirmation of the creation of the connector and then the alternating registration and deregistration of sets of five MBeans.

4. **You can now perform management operations on the `Agent`.**

   Use a JMX management console of your choice to examine the content of the MBean server through a JMXMP connector.

   Use the SNMP management application, `easymanager`, that is supplied with Java DMK to examine the `JMX-MBEAN-SERVER-MIB` through SNMP. You can find the `easymanager` application in the *installDir*/`contributions/easymanager` directory.

▼ To Examine `JMX-MBEAN-SERVER-MIB` Using `easymanager`

1. **Add the `JMX-MBEAN-SERVER-MIBOidTable.class` to your classpath.**

   Type the following command if you are using a UNIX platform:

   ```
   $ export CLASSPATH=$CLASSPATH:examplesDir/current/Snmp/MBeanTable/
   ```

   Type the following command if you are using a Windows platform:

   ```
   set classpath=%classpath%;examplesDir/current/Snmp/MBeanTable/
   ```

2. **Open *installDir*`/contributions/easymanager/bin`**

3. **Start easymanager.**

   If the SNMP agent has been started with the default example port, type the following command.

   ```
   $ easymanager.sh -port 16161
   ```

   You can now use easymanager to perform the following operations.

   - Pop up a MIB discovery window.

     ```
     discovermib -p JDMK_STD_V2_profile
     ```

   - Create a new MBean.

     ```
     set:2 -w private :<<jmxMBeanObjectName.998,4,Test:name=test1>,
     <jmxMBeanClassName.998,4,Agent$Simple>,
     <jmxMBeanRowStatus.998,2,4>>
     ```

   - Click on `resynch` in the MIB discovery window to observer the changes in the MIB.
   - Destroy the MBean you created.

     ```
     set:2 -w private :<<jmxMBeanRowStatus.998,2,6>>
     ```

18.3 Virtual SNMP Tables

This example shows how to implement virtual SNMP tables when instrumenting a MIB in Java DMK. It is based on the same MIB as the one used in the MBean Table Instrumentation example, so you should make sure you have run that example before running this one. However, the fundamental difference between this example and the SNMP Table Instrumentation example is that the tables implemented remain entirely virtual. The tables are calculated whenever they are needed by an SNMP request, rather than residing permanently in the SNMP session. The virtual table is also created with a limited period of validity. Once this period has expired, if the request for which the table was calculated has been executed, the table can be garbage-collected. There are thus two main advantages to operating with virtual tables rather than with tables that are permanently present.

- *Coherency*: A table snapshot is calculated when a request is received, unless there is already a valid table snapshot present. Every snapshot is an immutable object. Consequently, the table snapshot will remain coherent with the context of the request for the entire duration of the execution of that request.
- *Resource usage*: Unused and expired tables can be garbage-collected. Consequently, as long as there are no requests arriving, there is no redundant table consuming resources unnecessarily, and no need to continue updating the table when no managers are making requests.

The virtual SNMP table example is based on the classes in the *examplesDir*/current/Snmp/MBeanVirtualTable directory. It defines the MIB called JMX-MBEAN-SERVER-MIB that exposes the content of an MBean server through SNMP. This MIB is the same as the one defined in the previous example. To achieve this, JMX-MBEAN-SERVER-MIB defines the same two SNMP tables as before:

- jmxMBeanTable, which has read-create access permission, and contains one row per MBean registered in the remote MBean server.
- jmxMBeanAttrTable, which has read-only access permission, and extends the jmxMBeanTable. For each MBean mirrored in the jmxMBeanTable, it contains one additional row per attribute exposed by this MBean.

18.3.1 Classes Generated by mibgen

Before you can proceed with this example, you need to generate the JMX-MBEAN-SERVER-MIB and it's associated classes, by using the mibgen compiler supplied with Java DMK. For details of the mibgen compiler, see the *Java Dynamic Management Kit 5.1 Tools Reference Guide*.

▼ To Generate the JMX-MBEAN-SERVER-MIB

The example MIB is contained in the configuration file JMX-MBEAN-SERVER-MIB.mib, which is found in *examplesDir*/current/Snmp/MBeanVirtualTable. You should run the example from within this directory.

1. **Ensure that your PATH environment variable knows where to find mibgen.**

   In a default installation, mibgen is found in *installDir*/bin.

2. **Create a new directory, generated, inside *examplesDir*/current/Snmp/MBeanVirtualTable.**

   ```
   $ mkdir generated
   ```

   This directory is where mibgen generates the classes associated with JMX-MBEAN-SERVER-MIB.mib.

3. **Run the mibgen compiler.**

   ```
   $ mibgen -X:use-display-hint -X:no-table-access -X:abstract-mib \
               -d generated JMX-MBEAN-SERVER-MIB.mib
   ```

   The advanced mibgen option use-display-hint instructs mibgen to generate an attribute of type String for any object using a textual convention whose DISPLAY-HINT is 255a. This option is used because JMX-MBEAN-SERVER-MIB defines textual conventions (for example, JmxMBeanObjectNameTC) which must be translated into java.lang.String attributes, rather than the default Byte[] attributes.

   The -X:no-table-access option instructs mibgen not to generate a table accessor in the group MBean interfaces. The effect of this is that the accessors used to return the TableJmxMBeanTable and TableJmxMBeanAttrTable table objects are not created.

The -X:abstract-mib option, as the name suggests, generates an abstract MIB. In this case, the MIB class is an abstract class, in which the MBean factory methods are also abstract.

The -d generated option sends the output of mibgen to the newly created generated directory

Using the first two options makes it possible to get rid of any references to the default JmxMBeanServer class, as well as to its associated Table* objects. These classes are still generated, but our customer implementation no longer references them.

The MIB used in this example is the same as the one used in the SNMP table instrumentation example. Consequently, mibgen generates the same files in the generated directory as were generated in the table instrumentation example. The differences between this example and the instrumentation example are found in the customized classes, that are used to extend the generated classes.

18.3.2 Customized Classes

The customized classes extend the functionality of the classes of the generated MIB. As was the case in the previous example, the JMX_MBEAN_SERVER_MIB_Impl extends the JMX-MBEAN-SERVER-MIB generated by mibgen, to instantiate the instrumented JmxMBeanServerImpl class. However, the extended behavior that the customized classes bring to the MIB in this example differs from that provided by the SNMP table instrumentation example. Most significantly, whereas the JmxMBeanServerImpl class was the principal author of the customized behavior of the MIB in the previous example, in this example the bulk of the work is done by JmxMBeanTableMetaImpl.

The following sections examine each of the customized classes, describing their purpose, and where necessary, highlighting their differences with the corresponding classes in the SNMP table instrumentation example.

18.3.2.1 JmxMBeanServerImpl

JmxMBeanServerImpl instruments the jmxMBeanServer group defined in the MIB. It implements the JmxMBeanServerMBean interface that is generated by mibgen, but unlike the previous example, it does not extend the generated JmxMBeanServer skeleton. As a result of not extending the JmxMBeanServer object, and also because the MIB was generated with the -X:no-table-access option, the generated TableJmxMBeanTable and TableJmxMBeanAttrTable objects are not instantiated. Although they are still generated by mibgen, these classes are not used in this example.

The jmxMBeanTable and jmxMBeanAttrTable tables are implemented as virtual tables. They are computed on the fly when an SNMP request needs to access them. Furthermore, the jmxMBeanTable and jmxMBeanAttrTable tables are managed by their respective customized JmxMBeanTableMetaImpl and JmxMBeanAttrTableMetaImpl meta objects. In the table instrumentation example, the tables were managed by the JmxMBeanServerImpl object.

18.3.2.2 JmxMBeanServerMetaImpl

JmxMBeanServerMetaImpl extends the generated JmxMBeanServerMeta class, that is used to represent SNMP metadata for the JmxMBeanServer group.

JmxMBeanServerMetaImpl overrides the table-meta factory methods in order to instantiate customized implementations of the JmxMBeanTableMeta and JmxMBeanAttrTableMeta meta classes. These meta classes are customized in the JmxMBeanTableMetaImpl and JmxMBeanAttrTableMetaImpl meta classes, that are shown in “18.3.2.3 JmxMBeanTableMetaImpl” on page 347 and “18.3.2.4 JmxMBeanAttrTableMetaImpl” on page 348 respectively.

18.3.2.3 JmxMBeanTableMetaImpl

JmxMBeanTableMetaImpl extends the generated JmxMBeanTableMeta class, and overrides the following JmxMBeanTableMeta methods.

- getEntry(SnmpOid rowOid), so that it returns a JmxMBeanEntryImpl object for the indicated row instead of a JmxMBeanEntry table entry.
- contains(SnmpOid oid, Object userData), so that it tells whether the indicated row is present in the table.
- getNextOid(Object userData), so that it returns the row index, the rowOid, of the first row in the table.
- getNextOid(SnmpOid rowOid,Object userData), so that it returns the index of the row that immediately follows the indicated row in the table.
- createNewEntry(SnmpMibSubRequest req, SnmpOid rowOid, int depth), so that it makes it possible to create new rows from SNMP SET requests. Overriding this method is not necessary when the table is read-only.
- removeTableRow(SnmpMibSubRequest req, SnmpOid rowOid, int depth), so that it makes it possible to delete rows from SNMP SET requests. Overriding this method is not necessary when the table is read-only.

In addition to overriding the above methods, it is the JmxMBeanTableMetaImpl class that implements the caching mechanism. An SnmpUserData factory is used to create a transient request-contextual cache, that will remain active for the duration of the incoming request. After the request has completed, the contextual-cache is garbage-collected.

JmxMBeanTableMetaImpl defines the following subclasses, that implement the caching mechanism.

- The JmxMBeanTableData is a snapshot of the table index. It is computed on demand, and cached in a JmxMBeanTableCache object which holds it through a WeakReference. The JmxMBeanTableData includes a time-stamp which is used to determine whether the cached data is still valid, or if it needs to be recomputed.
- The JmxMBeanTableCache is a weak cache used to avoid constant recomputation of the JmxMBeanTableData. It holds the last computed JmxMBeanTableData in a weak reference, so that the garbage collector might reclaim it at any time, provided that no request is currently working on that most-recently computed copy. When an SNMP request that needs to access the jmxMBeanTable comes in, the JmxMBeanTableData is looked up from the JmxMBeanTableCache, and recomputed if needed. The JmxMBeanTableData object is then put in the request-contextual cache where it remains until the request is completed. The request will always consult that particular copy, thus ensuring the consistency of the returned data. If in the meantime a concurrent request comes in, it might obtain the same JmxMBeanTableData copy as the previous request, or a newly computed object, if the previous one was found to be obsolete. Consequently, the two requests might work on the same, or on different snapshots of the table. The snapshot obtained by a given request never changes.
- The JmxMBeanTableDataItem is a simple class that wraps an ObjectName and a JmxMBeanEntry. The JmxMBeanEntry is lazy-evaluated. The JmxMBeanTableData contains an array of JmxMBeanTableDataItem objects.

18.3.2.4 JmxMBeanAttrTableMetaImpl

JmxMBeanAttrTableMetaImpl extends the generated JmxMBeanAttrTableMeta class, and overrides the following methods.

- getEntry(SnmpOid rowOid), so that it returns an JmxMBeanAttrEntryImpl object for the indicated row.
- contains(SnmpOid oid, Object userData), so that it tells whether the indicated row is present int the table.
- getNextOid(Object userData), so that it returns the row index, the rowOid, of the first row in the table.
- getNextOid(SnmpOid rowOid,Object userData), so that it returns the index of the row that immediately follows the indicated row in the table.

The methods createNewEntry(SnmpMibSubRequest req, SnmpOid rowOid, int depth), and removeTableRow(SnmpMibSubRequest req, SnmpOid rowOid, int depth) are not overriden. The table is read-only, so these methods will never be called.

This class does not use any specific caching mechanism. Since the jmxMBeanAttrTable is an extension of the jmxMBeanTable it simply relies on the cache established for the jmxMBeanTable. The list of attributes pertaining to a specific MBean is cached, if necessary, in the corresponding JmxMBeanEntryImpl object, which can be reclaimed at the end of the request. Note that attribute values are not cached, but rather they are obtained when they are needed.

18.3.2.5 JmxMBeanContextFactory

JmxMBeanContextFactory implements the com.sun.management.snmp.agent.SnmpUserDataFactory interface. The userData allocated by this factory is a java.util.Map that serves as a request-contextual cache. When the MIB instrumentation needs to access a piece of data, it proceeds as follows.

- Firstly, it looks for it in the request-contextual cache, namely, in the userData.
- If the required data is not found,the MIB instrumentation computes its value, puts it in the request-contextual cache, and then returns it.

This mechanism ensures the coherency of the data returned in the response. Once a piece of data has been loaded into the request-contextual cache, it is consistently reused for the whole duration of the request. This makes sure that the same snapshot is always used during the request, even if the underlying managed object constantly changes its value.

18.3.2.6 JmxMBeanEntryImpl

JmxMBeanEntryImpl extends theJmxMBeanEntry skeleton that is generated by mibgen, and adds instrumentation to the entries of the jmxMBeanTable table defined in the MIB. Each JmxMBeanEntry represents an MBean registered in the exposed MBean server. The JmxMBeanEntry objects are created on the fly when needed. They are temporary objects which are created when an SNMP request needs access to the instrumented MBean. Once created, JmxMBeanEntry objects are put in the request-contextual cache, where they remain until the request completes.

18.3.2.7 JmxMBeanAttrEntryImpl

JmxMBeanAttrEntryImpl extends theJmxMBeanAttrEntry skeleton that is generated by mibgen, and adds instrumentation to the entries of the jmxMBeanAttrTable table defined in the MIB. Like JmxMBeanEntry objects, JmxMBeanAttrEntry objects are created on the fly when an SNMP request needs access to the instrumented MBean attributed.

JmxMBeanAttrEntry objects are stored in their corresponding JmxMBeanEntryImpl objects, which are themselves cached in the request-contextual cache and in the JmxMBeanTableCache.

18.3.3 Running the SNMP Virtual Tables Example

After you have run mibgen to generate JMX_MBEAN_SERVER_MIB and its associated classes, as explained in "18.2.1 Classes Generated by mibgen" on page 327, you can run the example.

▼ To Run the SNMP Virtual Tables Example

Run the example from the *examplesDir*/current/Snmp/MBeanVirtualTable directory.

1. **Compile the Java classes in the generated directory.**

   ```
   $ cd generated
   $ javac -d .. *.java
   ```

   This creates the compiled *.class files into the *examplesDir*/current/Snmp/MBeanVirtualTable directory, rather than in generated, so that they are accessible to the other classes used in the example.

2. **Compile the Java classes in the *examplesDir*/current/Snmp/MBeanVirtualTable directory.**

   ```
   $ cd ..
   $ javac -d . *.java
   ```

3. **Start the Agent.**

   You have several options when starting the Agent class.

   - To start the Agent on the default SNMP port, 16161:

     ```
     $ java  Agent service:jmx:jmxmp://
     ```

   - To start the Agent on a port of your choice:

     ```
     $ java  -Dsnmp.port=port_number Agent service:jmx:jmxmp://
     ```

   - To start the Agent with a connector of your choosing, you can specify a different service URL and connector port. For example:

     ```
     $ java  Agent JMX_service_URL
     ```

   In any of the above cases, you see confirmation of the SNMP port used, confirmation of the creation of the connector and then the alternating registration and deregistration of sets of five MBeans.

4. **You can now perform management operations on the Agent.**

   Use a JMX management console of your choice to examine the content of the MBean server through a JMXMP connector.

   Use the SNMP management application, easymanager, that is supplied with Java DMK to examine the JMX-MBEAN-SERVER-MIB through SNMP. You can find the easymanager application in the *installDir*/contributions/easymanager directory.

▼ To Examine JMX-MBEAN-SERVER-MIB Using easymanager

1. **Add the `JMX-MBEAN-SERVER-MIBOidTable.class` to your classpath.**

   Type the following command if you are using a UNIX platform:

   ```
   $ export CLASSPATH=$CLASSPATH:examplesDir/current/Snmp/MBeanVirtualTable/
   ```

   Type the following command if you are using a Windows platform:

   ```
   set classpath=%classpath%;examplesDir/current/Snmp/MBeanVirtualTable/
   ```

2. **Open *installDir*`/contributions/easymanager/bin`**

3. **Start easymanager.**

   If the SNMP agent has been started with the default example port, type the following command.

   ```
   $ easymanager.sh -port 16161
   ```

   You can now use easymanager to perform the following operations.

   - Pop up a MIB discovery window.

     ```
     discovermib -p JDMK_STD_V2_profile
     ```

   - Create a new MBean.

     ```
     set:2 -w private :<<jmxMBeanObjectName.998,4,Test:name=test1>,
     <jmxMBeanClassName.998,4,Agent$Simple>,
     <jmxMBeanRowStatus.998,2,4>>
     ```

   - Click on `resynch` in the MIB discovery window to observer the changes in the MIB.
   - Destroy the MBean you created.

     ```
     set:2 -w private :<<jmxMBeanRowStatus.998,2,6>>
     ```

CHAPTER 19

Security Mechanisms in the SNMP Toolkit

Both the simple network management protocol (SNMP) protocol adaptor and the SNMP manager API provide mechanisms for ensuring the security of management operations. Under SNMPv1 and SNMPv2, agents act as information servers, and IP-based access control is used to protect this information from unauthorized access. The SNMPv3 protocol provides much more sophisticated security mechanisms, implementing a user-based security model (USM), allowing both authentication and encryption of the requests sent between agents and their managers, as well as user-based access control.

The complete source code for these examples is available in subdirectories of the *examplesDir*/`current/Snmp/` directory (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "19.1 IP-Based Access Control Lists" on page 353 shows how to specify host communities to control manager access and send traps
- "19.2 SNMPv3 User-Based Access Control" on page 359 demonstrates the SNMPv3 user-based access control mechanism
- "19.3 SNMPv3 User-Based Security Model" on page 362 illustrates the SNMPv3 user-based security mechanisms
- "19.4 Legacy SNMP Security" on page 381 describes the legacy SNMPv1 and v2 security mechanisms

19.1 IP-Based Access Control Lists

For the SNMP adaptor, the Java Dynamic Management Kit (Java DMK) provides access control based on the IP address and community of the manager's host machine. Information on the access rights for communities and host machines is stored in access control lists (InetAddressAcl). The default implementation provided with the product uses an InetAddressAcl file, but you can provide your own implementation as described in "19.1.3 Custom Access Control" on page 358.

The InetAddressAcl mechanism can also be used to define the communities and managers to which the agent will send traps. When you rely on the InetAddressAcl trap group, the agent will send traps to all hosts listed in the InetAddressAcl file. See "16.3.1 Specifying the Trap Destination" on page 279 for the different ways that an agent application can send traps.

The following code example gives the contents of the *examplesDir*/current/Snmp/Agent/jdmk.acl file used when running the SNMP example applications. When using it in the security examples, you should replace *yourmanager* with the complete IP address or hostname of the host running your SNMP manager application.

EXAMPLE 19–1 jdmk.acl File

```
acl = {
 {
 communities = public
 access = read-only
 managers = yourmanager
 }
 {
 communities = private
 access = read-write
 managers = yourmanager
 }
}

trap = {
  {
  trap-community = public
  hosts = yourmanager
  }
}
```

19.1.1 InetAddressAcl File Format

An InetAddressAcl file contains an acl group defining community and manager access rights and a trap group defining the community and hosts for sending traps.

19.1.1.1 Format of the acl Group

The acl group contains one or more access configurations.

```
acl = {
   access1
   access2
```

```
    ...
   accessN
}
```

Each access configuration has the following format:

```
{
   communities = communityList
   access = accessRights
   managers = hostList
}
```

The *communityList* is a list of SNMP community names to which this access control applies. The community names in this list are separated by commas.

The *accessRights* specifies the rights to be granted to all managers connecting from the hosts specified in the *hostList*. There are two possible values: either `read-write` or `read-only`.

The *hostList* item gives the hosts of the managers to be granted the access rights. The *hostList* is a comma-separated list of hosts, each of which can be expressed as any one of the following:

- A host name
- An IP address
- A subnet mask, in the format `A.B.C.D/24`

The set of all access configurations defines the access policy of the SNMP agent. A manager whose host is specified in a *hostList* and that identifies itself in one of the communities of the same configuration will be granted the permissions defined by the corresponding *accessRights*. A manager's host can appear in several access configurations provided it is associated with a different community list. This will define different access communities with different rights from the same manager.

A manager whose host-community identification pair does not appear in any of the access configurations will be denied all access. This means that protocol data units (PDU) from this manager will be dropped without being processed.

19.1.1.2 Format of the Trap Group

The trap group specifies the hosts to which the agent will send traps if the `InetAddressAcl` mechanism is used. This group contains one or more trap community definitions.

```
trap = {
   community1
   community2
   ...
   communityN
}
```

Each community definition defines the association between a set of hosts and the SNMP community string in the traps to be sent to them. Each trap definition has the following format:

```
{
   trap-community = trapCommunityName
   hosts = trapHostList
}
```

The *trapCommunityName* item specifies a single SNMP community string. It will be included in the traps sent to the hosts specified in the `hosts` item. SNMPv3 does use the community string, so use IP addresses or the context name instead.

The *trapHostList* item specifies a comma-separated list of hosts. Each host must be identified by its name or complete IP address.

When the SNMP protocol adaptor is instructed to send a trap using the `InetAddressAcl` mechanism, it will send a trap to every host listed in the trap community definitions. If a host is present in more than one list, it will receive more than one trap, each one identified by its corresponding trap community.

19.1.1.3 Format of the Inform Group

The inform group specifies the hosts to which your agent or manager will send informs if the `InetAddressAcl` mechanism is used. This group contains one or more inform community definitions.

```
inform = {
   community1
   community2
   ...
   communityN
}
```

Each community definition defines the association between a set of hosts and the SNMP community string in the informs to be sent to them. Each inform definition has the following format:

```
{
   inform-community = informCommunityName
   hosts = informHostList
}
```

The *informCommunityName* item specifies a single SNMP community string. It will be included in the informs sent to the hosts specified in the `hosts` item. SNMPv3 does use the community string, so use IP addresses or the context name instead.

The *informHostList* item specifies a comma-separated list of hosts. Each host must be identified by its name or complete IP address.

When the SNMP protocol adaptor is instructed to send an inform using the InetAddressAcl mechanism, it will send an inform to every host listed in the inform community definitions. If a host is present in more than one list, it will receive more than one inform, each one identified by its corresponding inform community.

19.1.2 Enabling InetAddressAcl

The default InetAddressAcl mechanism provided with the Java DMK relies on an InetAddressAcl file to define the access rights and trap recipients. To enable access control with this mechanism, you must first write an InetAddressAcl file to reflect the access and trap policy of your SNMP agent. Then, there are two ways to enable file-based access control, one way to modify the file in use and one way to disable access control.

The simplest way of enabling access control and traps is to ensure that an InetAddressAcl file exists when the SNMP protocol adaptor MBean is instantiated. To be automatically detected, the InetAddressAcl file must be named jdmk.acl and must be located in the configuration directory of the Java DMK installation. On UNIX systems with a standard installation of the product, the configuration directory is owned by root and requires superuser privileges to write or modify the InetAddressAcl file.

Operating Environment	Configuration Directory
Solaris/Linux/Windows	*installDir*/SUNWjdmk/5.1/etc/conf/

The other way of enabling file-based access control is to specify a different file through the jdmk.acl.file system property. The filename associated with the property will override any InetAddressAcl file in the configuration directory. This property can be set programmatically, but it is usually done on the command line when starting your agent. For example, if the full pathname of your InetAddressAcl file is *MyAclFile*, use this command to start the agent with SNMP access control enabled:

```
$ java -classpath classpath -Djdmk.acl.file=MyAclFile MyAgent
```

If an InetAddressAcl file exists, the access rights it defines apply to all management applications that access the agent through its SNMP adaptor. This includes managers on the agent's local host: the InetAddressAcl groups must explicitly give permissions to localhost or the host's name or IP address for such managers. If the InetAddressAcl file does not exist when the SNMP adaptor is instantiated, either in the configuration directory or defined as a property, all SNMP requests will be processed, and traps will be sent only to the local host.

The InetAddressAcl file-based mechanism relies on the JdmkAcl class to provide the access control functionality. This is the class that is initialized with the contents of the InetAddressAcl file. This class provides the rereadTheFile method to reset the access control and trap lists with the contents of the InetAddressAcl file. This method will reload the same file that was used originally, regardless of any new property definitions. After you have updated the InetAddressAcl file, call the following methods to update the access control lists:

```
// assuming mySnmpAdaptor is declared as an SnmpAdaptorServer object
JdmkAcl theAcl = (JdmkAcl)(mySnmpAdaptor.getInetAddressAcl());
theAcl.rereadTheFile();
```

The JdmkAcl class that is used by default might not be suitable for all environments. For example, it relies on the java.security.acl package that is not available in the PersonalJava™ runtime environment. Therefore, one constructor of the SnmpAdaptorServer class lets you override this default, forcing the adaptor not to use access control, regardless of any existing InetAddressAcl file. If you specify false for the useAcl parameter of this constructor, the SNMP adaptor will not even search for an InetAddressAcl file. In this case, no access control is performed, as if there were no InetAddressAcl file: all SNMP requests will be processed, and traps will be sent only to the local host. For security considerations, the use of access control cannot be toggled once the SNMP adaptor has been instantiated.

19.1.3 Custom Access Control

The JdmkAcl class that relies on an ACL file is the default access control mechanism in the SNMP adaptor. For greater adaptability, the SnmpAdaptorServer class has constructors that let you specify your own access control mechanism. For example, if your agent runs on a device with no file system, you could implement a mechanism that does not rely on the jdmk.acl file.

To instantiate an SNMP adaptor with your own access control, use one of the constructors that takes an acl parameter of the type InetAddressAcl. Note that if this parameter's value is null, or if you use a constructor that does not specify an acl parameter, the SNMP adaptor will use the JdmkAcl class by default. If you want to instantiate an SNMP adaptor without access control, call the constructor with the useAcl parameter set to false.

Your access control mechanism must be a class that implements the InetAddressAcl interface. This interface specifies the methods that the SNMP adaptor uses to check permissions when processing a request. If you instantiate the SNMP adaptor with your access control class, the adaptor will call your implementation of the access control methods. Again, for security reasons, the InetAddressAcl implementation in use cannot be changed once the SNMP adaptor has been instantiated.

The JdmkAcl class implements the default access mechanism that uses the jdmk.acl file. It is also an implementation of the InetAddressAcl interface, and it provides a few other methods, such as rereadTheFile, to control the InetAddressAcl mechanism.

19.2 SNMPv3 User-Based Access Control

The user-based access control implemented by SNMPv3 is based on contexts and user names, rather than on IP addresses and community strings. It is a partial implementation of the view-based access control model (VACM) defined in SNMP RFC 2575.

The users, contexts and associated security information controlling access to the agents in an SNMP session are defined in the `jdmk.uacl` file, as shown in the following example, taken from the *examplesDir*`/current/Snmp/Agent` directory.

EXAMPLE 19–2 `jdmk.uacl` File

```
acl = {
{
context-names = TEST-CONTEXT
access = read-write
security-level=authNoPriv
users = defaultUser
}

}
```

In the `jdmk.uacl` file, you define the following:

- A list of context names, separated by commas; this example uses `TEST-CONTEXT` defined in "16.2.3 Binding the MIB MBeans" on page 270. You can define a null context by declaring `context-names = NULL`
- The access level, which can be either `read-write` or `read-only`
- The security level, as follows:

`noAuthNoPriv`	No security mechanisms activated
`authNoPriv`	Authentication activated, with no privacy
`authPriv`	Both authentication and privacy activated

- A list of authorized users, separated by commas; an asterisk (`*`) opens access to all users

The user ACL file allows any request from `defaultUser`, in the scope of `TEST-CONTEXT`, with a minimum of `authNoPrivacy` to be authorized. Any other request will be rejected.

19.2.1 Enabling User-Based Access Control

To enable user-based access control, you must create a UserAcl file. You must then direct the agent applying the access control to look in this file.

The simplest way of enabling access control and traps is to ensure that a user-based access control file UserAcl file exists when the SNMP protocol adaptor MBean is instantiated. To be automatically detected, the UserAcl file must be named jdmk.uacl and must be located in the configuration directory of the Java DMK installation. On UNIX systems with a standard installation of the product, the configuration directory is owned by root and requires superuser privileges to write or modify the UserAcl file.

Operating Environment	Configuration Directory
Solaris/Linux/Windows	*installDir*/SUNWjdmk/5.1/etc/conf/

The other way of enabling file-based access control is to specify a different file through the jdmk.uacl.file system property. The filename associated with the property will override any UserAcl file in the configuration directory. This property can be set programmatically, but it is usually done on the command line when starting your agent. For example, if the full pathname of your UserAcl file is *MyUaclFile*, use this command to start the agent with SNMP access control enabled:

```
$ java -classpath classpath -Djdmk.uacl.file=MyUaclFile MyAgent
```

If a UserAcl file exists, the access rights it defines apply to all management applications that access the agent through its SNMP adaptor. If the UserAcl file does not exist when the SNMP adaptor is instantiated, either in the configuration directory or defined as a property, all SNMP requests will be accepted.

The UserAcl file-based mechanism relies on the SnmpEngineParameters class to provide the access control functionality. This is the class that is initialized with the contents of the UserAcl file. This class provides the rereadTheFile method to reset the access control and trap lists with the contents of the UserAcl file. This method will reload the same file that was used originally, regardless of any new property definitions. After you have updated the UserAcl file, call the following method to update the access control lists:

```
Uacl.rereadTheFile
```

The following procedure demonstrates how to enable Uacl, using the example of the simple synchronous manager we saw in "17.1.3 Synchronous SNMPv3 Managers" on page 296 and the simple SNMPv3 agent from Example 16–2.

▼ To Run a Simple Manager with Access Control

1. **If you have not already done so, build and compile the `AgentV3` example in *examplesDir*`/current/Snmp/Agent`.**

   Type the following commands:

   ```
   $ mibgen -mo -d . mib_II.txt
   $ javac -classpath classpath -d . *.java
   ```

2. **Start the `AgentV3` example in its `Agent` directory, this time pointing it to its associated `jdmk.uacl` file, as well as to its `jdmk.security` file.**

   ```
   $ java -classpath classpath
   -Djdmk.security.file=jdmk.security -Djdmk.uacl.file=jdmk.uacl AgentV3
   ```

3. **If you have not already done so, in a separate window, compile the `SyncManagerV3` example in *examplesDir*`/current/Snmp/Manager`.**

   ```
   $ javac -classpath classpath -d . *.java
   ```

4. **Start the `SyncManagerV3` SNMP manager in its `Manager` directory, specifying the agent's host and port.**

   This is the manager we configured to communicate with `AgentV3` in "To Run the `SyncManagerV3` Example" on page 301. As before, we set the *host* to `localhost` and the *port* to `8085`.

   ```
   $ java -classpath classpath
   -Djdmk.security.file=jdmk.security SyncManagerV3 localhost 8085
   ```

   You should see the following error message:

   ```
   SyncManagerV3::main:Send get request to SNMP agent
   on localhost at port 8085
   Error status = authorizationError
   Error index = -1
   ```

   The agent refuses the manager's request because the level of security for this agent in the manager's `jdmk.security` file does not match the level of security set in the agent's `jdmk.uacl` file.

5. **Press `Control-C` to stop the manager.**

6. **Start the `SyncManagerV3` SNMP manager again**

   ```
   $ java -classpath classpath
   -Djdmk.security.file=jdmk.security SyncManagerV3 localhost 8085
   ```

   You should now see the manager sending requests to the agent.

   ```
   $ SyncManagerV3::main:Send get request to SNMP agent
   on localhost at port 8085
   Result:
   [Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
   Value : SunOS sparc 5.8]
   >> Press Enter if you want to stop this SNMP manager.
   ```

19.3 SNMPv3 User-Based Security Model

As stated in the earlier SNMP chapters in this manual, SNMPv3 implements a much more sophisticated set of security mechanisms than the previous versions of SNMP. The SNMPv3 USM enables you to implement authentication and privacy in the communication that takes place between your SNMP agents and managers. SNMPv3 also introduces the concept of the authoritative SNMP engine and enables you to create authorized users for specific SNMPv3 agents.

Chapter 16 and Chapter 17 provided just enough information about configuring SNMPv3 security to enable you to run the examples in those chapters. The following sections provide a more complete description of SNMPv3 security.

19.3.1 SNMPv3 Engines

SNMPv3 introduces the notion of the *authoritative* SNMP engine. The concept of authoritative is defined as follows:

- Authoritative engines receive acknowledged requests, namely `get`, `set`, `getnext`, `getbulk` and informs. Authoritative engines also send traps (unacknowledged)
- Non-authoritative engines send the above acknowledged requests and receive unacknowledged traps

Being authoritative means that entities have the ability to accept or deny requests from other entities, depending upon whether or not both sides of the exchange have been appropriately configured to communicate with each other, and whether the request itself arrives in a timely fashion. To check the timeliness of a request, the authoritative engine checks the time of sending included in the request against its own internal clock. If the difference between the time of sending and the time of receipt recorded by the authoritative engine exceeds 150 seconds, the request is not considered timely and is rejected.

The authoritative engine also checks timeliness by reading the `localEngineBoots` value recorded in the request, and comparing it to the number of reboots that the sending engine has undergone. It checks this by calling the `SnmpEngine.getEngineBoots`. If the value recorded in the request and the value returned by `SnmpEngine.getEngineBoots` do not correspond, the request is rejected.

In general, agents are authoritative, and managers are non-authoritative. However, when receiving informs, managers are authoritative, and can accept or deny the informs according to their timeliness.

Java DMK 5.1 associates an SNMP engine with every SnmpV3AdaptorServer that is instantiated. Engines can be shared between several SNMP sessions. SNMP engines are identified by their engine ID.

19.3.2 Generating SNMPv3 Engine IDs

SNMPv3 engine ID objects are generated in accordance with SNMP RFC 2571. An engine discovers its ID in one of the following ways, each of which is tried in the order shown:

1. It is found in the engine's associated jdmk.security file
2. It is found using SnmpEngineParameters
3. It is computed by the SNMP adaptor, based on the host and port information, or it is computed by the SNMP session, based on time

In Java DMK 5.1, you can create new SNMP engine IDs either manually using a Java command line tool called EngineIdGenerator, or automatically in the code of your applications using the SnmpEngineId class. You can see the EngineIdGenerator tool in the *examplesDir*/current/Snmp/EngineId directory. You must provide either of EngineIdGenerator or SnmpEngineId with any of the following information:

- An engine ID you have computed previously
- An IP address or DNS host name
- A UDP port
- An Internet Assigned Numbers Authority (IANA) number

The EngineIdGenerator uses the SnmpEngineId class when generating engine IDs (see the Javadoc entry for SnmpEngineId for details). The SnmpEngineId class simplifies the configuration of SNMP entities by enabling you to identify engines using information that is easily comprehensible to humans, such as host names and port numbers, which it then converts into system-oriented hexadecimal engine IDs for you. The SnmpEngineId class also generates SNMP object identifiers (OIDs) from the engine IDs it creates. This is particularly useful when computing USM MIB user table indexes, as seen in "19.3.6 Creating Users for SNMPv3 USM MIBs" on page 377.

▼ To Run the SNMP EngineIdGenerator Example

1. **Compile the EngineIdGenerator class in** ***examplesDir*****/current/Snmp/EngineId.**

   ```
   $ javac -classpath classpath EngineIdGenerator.java
   ```

2. **Start the EngineIdGenerator tool.**

   ```
   $ java -classpath classpath EngineIdGenerator
   Start making your engine Id construct:(h for help)
   #:
   ```

Typing h will provide examples of information you can provide.

3. **Provide the relevant information to create the engine ID.**

   Declare your information, using the appropriate separators, as follows:

   address:*port*:*IANA number*
   All three inputs are used

   address:*port*
   The address and port you specify are used; the IANA number defaults to 42 (SUN Microsystems)

   address
   The address you specify is used; the port defaults to 161 and the IANA number defaults to 42 (SUN Microsystems)

   :*port*
   The port you specify is used; the host defaults to localhost and the IANA number defaults to 42 (SUN Microsystems)

   ::*IANA number*
   The IANA number you specify is used; the host defaults to localhost and the IANA number defaults to 42 (SUN Microsystems)

   :*port*:*IANA number*
   The port and IANA number you specify are used; the host defaults to localhost

   address::*IANA number*
   The address and IANA number you specify are used; the port defaults to 161

   ::
   The port defaults to 161, the address defaults to localhost and the IANA number defaults to 42 (SUN Microsystems)

   For example, to specify all three of the address, port and IANA number, when prompted you might type:

```
Start making your engine Id construct:(h for help)
#:localhost:8087:42
Generated engine Id ******** [0x8000002a05819dcb6200001f97] ********
#:
```

4. **Press `Control-C` when you have finished generating engine IDs**

19.3.3 SNMPv3 USM Configuration

The SNMPv3 USM is configured in a Java DMK text file, called `jdmk.security`. Every SNMP engine has an associated security file.

In a traditional agent and manager SNMP architecture, you will have one security file associated with the agent and one associated with the manager. Both files will have a very similar configuration.

The authoritative agent's security file contains all the security information users need when requests are received from the manager. The non-authoritative manager's security file contains all the security information users need when making requests of authoritative agents.

The following examples show typical security files for an agent and a manager.

EXAMPLE 19–3 A Typical Agent `jdmk.security` File

```
localEngineID=myHost:8085
localEngineBoots=7

#Typical authenticated entry. Accepts requests from a user called
#aSecureUser
userEntry=localEngineID,aSecureUser,aSecureUser,
usmHMACMD5AuthProtocol, mypasswd

#Typical authenticated and encrypted entry. Accepts requests from
#aSecureUser
#userEntry=localEngineID,aSecureUser,aSecureUser,
#usmHMACMD5AuthProtocol, #mypasswd,usmDESPrivProtocol,mypasswd
```

The example agent `jdmk.security` file identifies the agent's associated SNMP engine using its host name and port number, records the number of times that engine has rebooted, and sets two possible security configurations for a user called `aSecureUser`. One possible configuration applies authentication to requests from `aSecureUser`. The second configuration, which is currently commented out and is therefore inactive, applies both authentication and privacy to requests from the same user.

EXAMPLE 19–4 A Typical Manager `jdmk.security` File

```
#Typical authenticated entry. Makes requests to authoritative engine
#myHost:8085 with some parameters.
userEntry=myHost:8085,aSecureUser,aSecureUser,usmHMACMD5AuthProtocol,
mypasswd

#Typical authenticated and encrypted entry. Makes requests to authoritative
#engine myHost:8085 with some parameters.
#userEntry=myHost:8085,aSecureUser,aSecureUser,usmHMACMD5AuthProtocol,
#mypasswd,
#usmDESPrivProtocol,mypasswd

# #####APPENDED PROPERTY####
localEngineBoots=5

# #####APPENDED PROPERTY####
localEngineID=myOtherHost:8087
```

The example manager `jdmk.security` file sets two possible configurations to send requests from the user `aSecureUser` to the above agent. Again, the first configuration applies authentication to requests from `aSecureUser`, and the second configuration, which is currently commented out and is therefore inactive, applies both authentication and privacy.

Note – The `localEngineID` for each of the manager and the agent must be different. If two entities that communicate with each other have the same local engine ID, behavior is unpredictable.

19.3.3.1 Adding Users to the Security Files

As you can see in Example 19–3 and Example 19–4, every user that has access to an agent is represented by a `userEntry` row in each of the agent's and the manager's security files. The example manager `jdmk.security` file is configured to send requests from `aSecureUser` to the agent, either with authentication only, or with privacy activated. The agent is configured to receive those requests.

You configure `userEntry` as follows, with the parameters separated commas:

`userEntry=`*engine ID*`,` *user name*`,` *security name*`,` *authentication algorithm*`,` *authentication key*`,` *privacy algorithm*`,` *privacy key*`,` *storage type*`,` *template*

The only mandatory parameters are the engine ID and the user name. All the other parameters are optional.

The possible values for the parameters are as follows:

Engine ID — A local or remote SNMP engine, defined in one of the following ways:

- The string `localEngineID`, to denote the local engine
- A hexadecimal string, as generated by `EngineIdGenerator`; for example, `0x8000002a05819dcb6e00001f95`
- A human readable string used to generate an engine ID, providing any or all of the host name, port and IANA number, as shown in "19.3.2 Generating SNMPv3 Engine IDs" on page 363

User name — Any human-readable string

Security name — Any human-readable string

Authentication algorithm — The following algorithms are permitted:

- `usmHMACMD5AuthProtocol`

- `usmHMACSHAAuthProtocol`
- `usmNoAuthProtocol`

Authentication key

Any text password or any hexadecimal key starting with `0x`; for example, `0x0098768905AB67EFAA855A453B665B12`, of size:

- 0 to 32 inclusive for HMACMD5
- 0 to 40 inclusive for HMACSHA

Privacy algorithm

The following algorithms are permitted:

- `usmDESPrivProtocol`
- `usmNoPrivProtocol` If no algorithm is specified, the default is `usmNoPrivProtocol`.

Any text password or any hexadecimal key starting with `0x`; for example, `0x0098768905AB67EFAA855A453B665B12`, of size 0 to 32 inclusive

If a hexadecimal string is provided, it must be a localized key

Storage type

A value of 3 denotes *non-volatile*, meaning that the user entry is flushed in the security file; any other value than 3 will be rejected, throwing an `IllegalArgumentException`

template

Can be either `true` or `false`:

If `true`, the row is a template, not seen from USM MIB. This kind of user is used when cloning users.

The default is `false`.

19.3.4 Enabling Privacy in SNMPv3 Agents

As shown in the example security files given in “19.3.3 SNMPv3 USM Configuration” on page 364, you can protect the communication between your SNMPv3 entities by enabling encryption, otherwise known as privacy.

The privacy algorithms used by SNMPv3 are the data encryption standard (DES) protocol from the Java Cryptography Extension (JCE) from the Java 2 Platform, Standard Edition (J2SE) 1.4, as well as the secure hash algorithm (SHA) and message digest 5 (MD5) encryption protocols provided since J2SE 1.2.

To run an SNMP entity with privacy enabled, you must configure both the entity itself and its corresponding security file. The following example shows the code for an SNMPv3 agent with privacy enabled, called AgentEncryptV3. This example is found in the *examplesDir*/current/Snmp/Agent directory.

EXAMPLE 19–5 AgentEncryptV3 Agent with Privacy Enabled

```
public class AgentEncryptV3 {

    static SnmpV3AdaptorServer snmpAdaptor = null;

    private static int nbTraps = -1;

    public static void main(String args[]) {

        final MBeanServer server;
        final ObjectName htmlObjName;
        final ObjectName snmpObjName;
        final ObjectName mibObjName;
        final ObjectName trapGeneratorObjName;
        int htmlPort = 8082;
        int snmpPort = 161;

        // Parse the number of traps to be sent.

            [...]

      // SNMP specific code:

      [...]

            // Set up encryption

           //First create parameters.
            SnmpEngineParameters parameters = new SnmpEngineParameters();

            //Then activate encryption
            parameters.activateEncryption();

            //Create the SNMPv3 adaptor and pass it the parameters.
            snmpAdaptor = new SnmpV3AdaptorServer(parameters,
           null,
           null,
           snmpPort,
           null);

            // Register the SNMP Adaptor in the MBean Server
           //
            server.registerMBean(snmpAdaptor, snmpObjName);

            // Register the USM MIB
           snmpAdaptor.registerUsmMib(server, null);

            // Start the adaptor.
```

EXAMPLE 19–5 `AgentEncryptV3` Agent with Privacy Enabled *(Continued)*

```
        snmpAdaptor.start();

        // Send a coldStart SNMP Trap.
        // Use port = snmpPort+1.
        //
        print("NOTE: Sending a coldStart SNMP trap to each " +
              "destination defined in the ACL file...");

        snmpAdaptor.setTrapPort(new Integer(snmpPort+1));
        snmpAdaptor.snmpV1Trap(0, 0, null);
        println("Done.");

        // Create the MIB II (RFC 1213) and add it to the MBean server.
        //
        mibObjName= new ObjectName("snmp:class=RFC1213_MIB");
        Trace.send(Trace.LEVEL_TRACE, Trace.INFO_MISC, "Agent", "main",
                   "Adding RFC1213-MIB to MBean server with name \n\t" +
                   mibObjName);

        // Create an instance of the customized MIB
        //
        RFC1213_MIB mib2 = new RFC1213_MIB_IMPL();
        server.registerMBean(mib2, mibObjName);

        // Bind the SNMP adaptor to the MIB to make the MIB
        // accessible through the SNMP protocol adaptor.
        //
            snmpAdaptor.addMib(mib2, "TEST-CONTEXT");

        // Create a LinkTrapGenerator.
        // Specify the ifIndex to use in the object name.
        //
        String trapGeneratorClass = "LinkTrapGenerator";
        int ifIndex = 1;
        trapGeneratorObjName = new ObjectName("trapGenerator" +
                        ":class=LinkTrapGenerator,ifIndex=" + ifIndex);
        Trace.send(Trace.LEVEL_TRACE, Trace.INFO_MISC, "Agent", "main",
              "Adding LinkTrapGenerator to MBean server with name \n\t"+
              trapGeneratorObjName);
        LinkTrapGenerator trapGenerator =
                new LinkTrapGenerator(nbTraps);
        server.registerMBean(trapGenerator, trapGeneratorObjName);

        println("\n>> Press Enter if you want to start sending traps."+
        " SNMP V1 and SNMP V3 traps will be sent.");
        println("   -or-");
        println(">> Press Ctrl-C if you want to stop this agent.");
        java.lang.System.in.read();

        trapGenerator.start();

    } catch (Exception e) {
```

EXAMPLE 19–5 `AgentEncryptV3` Agent with Privacy Enabled *(Continued)*

```
            e.printStackTrace();
        }
    }

}
```

By default, a Java DMK 5.1 agent handles requests that are authenticated, but not encrypted. To activate encryption, you need to set certain parameters when you instantiate the SNMP engine. As shown in Example 19–5, these parameters are passed to the engine using the `SnmpEngineParameters` class, as follows:

- Firstly, the application creates new SNMP engine parameters, called `parameters` in this example, by calling `SnmpEngineParameters`:

  `SnmpEngineParameters parameters = new SnmpEngineParameters();`
- Then it activates encryption by making `parameters` call the `activateEncryption` method:

  `parameters.activateEncryption();`
- Finally, it then passes the parameters to the newly created SNMPv3 adaptor server:

  `snmpAdaptor = new SnmpV3AdaptorServer(parameters, null, null, snmpPort, null)`

The `AgentEncryptV3` application then continues with the registration of the SNMP adaptor server in the MBean server, binding the MIBs and calling `LinkTrapGenerator` in the same way as any other agent.

As well as the agent itself, you must also configure the security file associated with that agent. Example 19–6 shows the security file associated with `AgentEncryptV3`.

EXAMPLE 19–6 Agent `jdmkencrypt.security` File

```
#Local engine Id.
localEngineID=0x8000002a05819dcb6e00001f95
#Number of boots.
localEngineBoots=0

#defaultUser configuration.
userEntry=localEngineID,defaultUser,null,usmHMACMD5AuthProtocol,mypasswd,
usmDESPrivProtocol,mypasswd,3,
```

In this file, you can see that the DES privacy protocol is specified.

▼ To Run the AgentEncryptV3 Example

1. **If you have not already done so, build and compile the `AgentEncryptV3` example in *examplesDir*`/current/Snmp/Agent`.**

   Type the following commands:

   ```
   $ mibgen -d . mib_II.txt
   $ javac -classpath classpath -d . *.java
   ```

2. **Start the `AgentEncryptV3` agent, passing it its associated security file, `jdmkencrypt.security`.**

   ```
   $ java -classpath classpath
   -Djdmk.security.file=jdmkencrypt.security AgentEncryptV3 [nb_traps]
   ```

   In the command above, *nb_traps* represents the number of traps that you want to send.

3. **Press `Enter` to start sending traps**

   ```
   NOTE: Sending a linkDown SNMP trap for the Interface 1 to each
   destination defined in the ACL file...Done.
   NOTE: Sending a linkDown SNMP trap for the Interface 1 to each
   destination defined in the ACL file...Done.
   ```

4. **Press `Control-C` to stop the agent**

19.3.5 Enabling Privacy in SNMPv3 Managers

If you enable privacy in your SNMPv3 agents, then you must also enable privacy in the corresponding manager. The following example shows the code for an SNMPv3 agent with privacy enabled, called `SyncManagerEncryptV3`. This example is found in the *examplesDir*`/current/Snmp/Manager` directory.

EXAMPLE 19–7 `SyncManagerEncryptV3` Manager with Privacy Enabled

```
/**
 public class SyncManagerEncryptV3 {

        public static void main(String argv[]) {
        SnmpSession session = null;

        if (argv.length != 2) {
            usage();
            java.lang.System.exit(1);
        }

            //Check arguments first
            //host and port.
        final String host = argv[0];
        final String port = argv[1];
```

```
  // Initialize the SNMP Manager API.
//
    [...]


  // Activate the encryption

   //

       // First create parameters.
    //
    final SnmpEngineParameters parameters =
    new SnmpEngineParameters();

    // Then activate encryption
    parameters.activateEncryption();

    // Finaly create the session passing it the parameters.
    try {
    // When instantiating a session, a new SNMP V3 engine is
    // instantiated.
    session= new SnmpSession(parameters,
 null,
 "SyncV3Manager session",
 null);
    }catch(SnmpStatusException e) {
    println(e.getMessage());
    java.lang.System.exit(0);
}
    catch(IllegalArgumentException e) {
    //If the engine configuration is faulty
    println(e.getMessage());
    java.lang.System.exit(0);
}

       final SnmpEngine engine = session.getEngine();

    // Create a SnmpPeer object
   //
    final SnmpUsmPeer agent =
    new SnmpUsmPeer(engine, host, Integer.parseInt(port));

    // Create parameters to associate to the entity to
    // communicate with.
      //
    final SnmpUsmParameters p =
    new SnmpUsmParameters(engine, "defaultUser");

    // Set Security level
    //
     p.setSecurityLevel(SnmpDefinitions.authPriv);
```

EXAMPLE 19–7 `SyncManagerEncryptV3` Manager with Privacy Enabled *(Continued)*

```
// Register MIBS under the scope of a context.
//
p.setContextName("TEST-CONTEXT".getBytes());

// Specify a contextEngineId. This is
//
p.setContextEngineId(agent.getEngineId().getBytes());

// The newly created parameter must be associated to the agent.
//
agent.setParams(p);


// Discovery timeliness
//
agent.processUsmTimelinessDiscovery();

// A default peer (agent) can be associated to a SnmpSession.
  //
   session.setDefaultPeer(agent);

  // Create a listener and dispatcher for SNMP traps
final SnmpEventReportDispatcher trapAgent =
new SnmpEventReportDispatcher(engine,
              Integer.parseInt(port) + 1,
              taskServer, null);
trapAgent.addTrapListener(new TrapListenerImpl());
final Thread trapThread = new Thread(trapAgent);
trapThread.setPriority(Thread.MAX_PRIORITY);
trapThread.start();


// Build the list of variables you want to query.
// For debug purposes, you can associate a name to your list.
//
final SnmpVarBindList list =
new SnmpVarBindList("SyncManagerEncryptV3 varbind list");

// We want to read the "sysDescr" variable.
//
// We will thus query "sysDescr.0", as sysDescr is a scalar
    // variable (see RFC 1157, section 3.2.6.3.  Identification
   // of Object Instances, or RFC 2578, section 7.  Mapping of
    // the OBJECT-TYPE macro).
    //
    list.addVarBind("sysDescr.0");

// Make the SNMP get request and wait for the result.
//
final SnmpRequest request = session.snmpGetRequest(null, list);
println("SyncManagerEncryptV3::main:" +
```

EXAMPLE 19–7 `SyncManagerEncryptV3` Manager with Privacy Enabled *(Continued)*

```
     " Send get request to SNMP agent on " +
     host + " at port " + port);
final boolean completed = request.waitForCompletion(10000);

// Check for a timeout of the request.
//
if (completed == false) {
    println("SyncManagerEncryptV3::main:" +
" Request timed out. Check reachability of agent");

    // Print request.
    //
    println("Request: " + request.toString());
    java.lang.System.exit(0);
}

// Check if the response contains an
    // error.
//
final int errorStatus = request.getErrorStatus();
if (errorStatus != SnmpDefinitions.snmpRspNoError) {
    println("Error status = " +
    SnmpRequest.snmpErrorToString(errorStatus));
    println("Error index = " +
    request.getErrorIndex());
    java.lang.System.exit(0);
}

// Display the content of the result.
//
final SnmpVarBindList result = request.getResponseVarBindList();
println("Result: \n" + result);

println("\n>> Press Enter if you want to stop" +
" this SNMP manager.\n");
java.lang.System.in.read();

// Nicely stop the session
//
session.destroySession();

    // End the SnmpEventReportDispatcher.
    //
    trapAgent.close();
    taskServer.terminate();

//
// That's all !
//
java.lang.System.exit(0);

    } catch(Exception e) {
```

EXAMPLE 19–7 `SyncManagerEncryptV3` Manager with Privacy Enabled *(Continued)*

```
            java.lang.System.err.println("SyncManagerEncryptV3::main:" +
                     " Exception occurred:" + e );
            e.printStackTrace();
        }
    }


}
```

By default, a Java DMK 5.1 manager handles requests that are authenticated, but not encrypted. To activate encryption, you need to set certain parameters when you instantiate the SNMP session. As shown in Example 19–7, these parameters are passed to the engine using the `SnmpEngineParameters` class, as follows:

- Firstly, the application creates new SNMP engine parameters, called `parameters` in this example, by calling `SnmpEngineParameters`:

  `SnmpEngineParameters parameters = new SnmpEngineParameters();`
- Then it activates encryption by making `parameters` call the `activateEncryption` method:

  `parameters.activateEncryption();`
- Finally, it then passes the parameters to the newly created SNMPv3 session:

  `session= new SnmpSession(parameters, null, "SyncV3Manager session", null)`

The `SyncManagerEncryptV3` manager application then continues with the generation of a USM peer, defining the context and setting trap listeners in the same way as any other manager. Note, however, that in this manager, the security level is set to `authPriv`.

As well as the manager itself, you must also configure the security file associated with that manager. Example 19–8 shows the security file associated with `SyncManagerEncryptV3`.

EXAMPLE 19–8 Manager `jdmkencrypt.security` File

```
#Authentication and encryption.
userEntry=0x8000002a05819dcb6e00001f95,defaultUser,,
usmHMACMD5AuthProtocol,mypasswd,usmDESPrivProtocol,mypasswd

# #####APPENDED PROPERTY####
localEngineBoots=2

# #####APPENDED PROPERTY####
localEngineID=0x8000002a05000000ebffd342ca
```

As was the case for the `AgentEncryptV3` agent, in this file, you can see that the DES privacy protocol is specified.

▼ To Run the `SyncManagerEncryptV3` Example

1. **If you have not already done so, build and compile the `AgentEncryptV3` example in *examplesDir*`/current/Snmp/Agent`.**

   Type the following commands:

   ```
   $ mibgen -d . mib_II.txt
   $ javac -classpath classpath -d . *.java
   ```

2. **Start the `AgentEncryptV3` agent, passing it its associated security file, `jdmkencrypt.security`.**

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmkencrypt.security
   AgentEncryptV3
   ```

   Press `Enter` to start sending traps.

3. **Press `Enter` to start sending traps.**

4. **In another window, if you have not already done so, build and compile the `SyncManagerEncryptV3` example in *examplesDir*`/current/Snmp/Manager`.**

   Type the following commands:

   ```
   $ mibgen -mo -d . mib_II.txt
   $ javac -classpath classpath -d . *.java
   ```

5. **Start the `SyncManagerEncryptV3` manager, passing it its associated security file, `jdmkencrypt.security`, and specifying the host name and port number of the agent it is to communicate with.**

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmkencrypt.security
   SyncManagerEncryptV3 localhost 8085
   ```

   You should see the manager start to receive encrypted traps from the agent.

   ```
   SyncManagerEncryptV3::main:
   Send get request to SNMP agent on localhost at port 8085
   Result:
   [Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
   Value : SunOS sparc 5.8]

   >> Press Enter if you want to stop this SNMP manager.

   NOTE: TrapListenerImpl received trap V3:
           ContextEngineId : 0x8000002a05819dcb6e00001f95
           ContextName : TEST-CONTEXT
           VarBind list :
   oid : 1.3.6.1.2.1.1.3.0 val : 0:0:40
   oid : 1.3.6.1.6.3.1.1.4.1.0 val : 1.2.3.4.5.6.7.8.9.0
   ```

```
oid : 1.3.6.1.2.1.2.2.1.1.1 val : 1
```

6. **Press `Control-C` in each window to stop both the agent and the manager**

19.3.6 Creating Users for SNMPv3 USM MIBs

The SNMPv3 USM implemented in Java DMK 5.1 enables you to create users remotely in an SNMPv3 agent by accessing a MIB that has been registered in the SNMPv3 adaptor server. By default, the USM MIB is not registered in the adaptor server. You can register a MIB in the adaptor server by calling `registerUsmMib`.

```
$ snmpV3AdaptorServer.registerUsmMib()
```

Caution – You can use `registerUsmMib` to register your MIB in the MBean server, making it available via the HTML server. This can be useful for debugging purposes, but this can also represent a security breach.

The `CreateUsmMibUser` example in the *examplesDir*/`current/Snmp/UsmMib` directory is a tool that uses the SNMPv3 manager API to instantiate a new user in an agent USM MIB. `CreateUsmMibUser` performs authenticated and encrypted communication with the agent `Agent`, which is found in the same directory.

The complete code for the `CreateUsmMibUser` example is too long to show here, but the process that it goes through to create new users remotely can be summarized as follows:

- The `CreateUsmMibUser` class uses a user template, called `defaultUser`, to bootstrap the remote configuration of the USM MIB. Because `defaultUser` is a template, it does not appear in the USM MIB, and therefore it is not visible to remote managers. `CreateUsmMibUser` knows of `defaultUser` because it is present in its configuration file, `manager.security`, as shown in the following example.

  EXAMPLE 19–9 `manager.security` File for the `CreateUsmMibUser` Example

  ```
  localEngineID=0x8000002a05000000ec6c315f54
  localEngineBoots=0

  # User to create remotely in the agent.
  userEntry=0x000000000000000000000002,myNewUser,
  myNewUser,usmHMACMD5AuthProtocol,newsyrup,
  usmDESPrivProtocol,newsyrup

  # Template user to be used by the manager
  userEntry=0x000000000000000000000002,defaultUser,,
  usmHMACMD5AuthProtocol,maplesyrup,usmDESPrivProtocol,
  ```

EXAMPLE 19–9 `manager.security` File for the `CreateUsmMibUser` Example *(Continued)*

```
maplesyrup
```

`CreateUsmMibUser` uses the `defaultUser` template to send an initial secure configuration request to create a new user, called `myNewUser` in this example. The newly created user will be a normal user, and will thus appear in the USM MIB.

- `CreateUsmMibUser` implements the key-change mechanism defined in SNMP RFC 2574. This enables you to allocate new keys to the newly created users. Calling `getUsmKeyHandler` enables you to compute key localization and delta generation. An instance of `SnmpUsmKeyHandler` is associated to each SNMP engine object. When one of the standard authentication algorithms is used when computing the key. Because the `CreateUsmMibUser` example operates with authentication activated, the new peer agent has to perform timeliness checks on the incoming requests for the creation of new users. If proven timely, the request for a new user is granted and the creation process proceeds. Otherwise, the request is rejected.
- Once the request has been accepted and the data provided by the user has been parsed and processed, the `CreateUsmMibUser` clones a new `userEntry` in the agent's `jdmk.security` file. The cloned entry is based on the `defaultUser` template entry, with the new information provided by the user added using the `cloneFromUser` variables.
- In the `CreateUsmMibUser` example, the new user `myNewUser` is remotely granted access to the agent `Agent`. The agent's initial security configuration is set in the `jdmk.security` file for `Agent`, which is found in the *examplesDir*`/current/Snmp/UsmMib` directory, as shown below.

EXAMPLE 19–10 `jdmk.security`File for `Agent` in the `CreateUsmMibUser` Example

```
localEngineID=0x000000000000000000000002
localEngineBoots=0

userEntry=localEngineID,defaultUser,,usmHMACMD5AuthProtocol,
maplesyrup,usmDESPrivProtocol,maplesyrup,3,true
```

As you can see, the `jdmk.security` file currently only allows the `defaultUser` template to access the agent. The `CreateUsmMibUser` class will remotely add an extra row to this file to allow `myNewUser` to access the agent too.

▼ To Run the `CreateUsmMibUser` Example

1. **If you have not already done so, build and compile the examples in** *examplesDir*`/current/Snmp/UsmMib`**.**

   Type the following commands:

```
$ javac -classpath classpath -d . *.java
```

2. **Make sure that no other agents are running in *examplesDir*/current/Snmp/UsmMib, and start Agent.**

```
$ java -classpath classpath -Djdmk.security.file=jdmk.security Agent
```

Note – The jdmk.security file must be writable if CreateUsmMibUser is to be able to add new user entries.

---

3. **In another window, start the CreateUsmMibUser example.**

   When starting CreateUsmMibUser, you must point it to the manager.security configuration file, and specify the user name, the security level, the agent's host name and the port on which the agent is running. In this example, the security level is authentication and privacy enabled, and the agent is running on the local host.

```
$ java -classpath classpath -Djdmk.security.file=manager.security
CreateUsmMibUser defaultUser noAuthNoPriv localhost 8085
```

   You will see the following output:

```
Initializing creator.
Ready for new user inputs.
```

4. **When prompted, provide the configuration information for your new user.**

   The information you provide must correspond to users that you have already configured into your manager's security file. In this example, we are remotely adding the user myNewUser that is defined in manager.security to the agent Agent. You therefore provide the following information, all of which is found in the manager.security file. You can enter any value for the auth key random and the priv key random.

```
Type the engine Id :0x000000000000000000000002
Type the new user name :myNewUser
Type the clone from user name :defaultUser
Type the security level :authPriv
Type the old priv password :maplesyrup
Type the new priv password :newsyrup
Type the priv key random :00000000000000000000000000000000
Type the auth protocol :usmHMACMD5AuthProtocol
Type the old auth password :maplesyrup
Type the new auth password :newsyrup
Type the auth key random :00000000000000000000000000000000
```

   You will see the following output:

```
********** Input summary ************

        * Engine Id : 0x000000000000000000000002
        * New user name : myNewUser
        * Clone from : defaultUser
        * Security level : authPriv
        * Old priv password : maplesyrup
        * New priv password : newsyrup
        * Priv key random : 00000000000000000000000000000000
        * Auth protocol : usmHMACMD5AuthProtocol
        * Old auth password : maplesyrup
        * New auth password : newsyrup
        * Auth key random : 00000000000000000000000000000000
Do you agree (yes, no) [yes]:
```

5. **Press `Enter` to confirm your inputs.**

   You should see the following confirmation:

```
***** New user [myNewUser] created.
***** Doing Priv key change
***** Priv key change DONE.
***** Doing Auth key change
***** Auth key change DONE.
***** Setting row status to active.
***** Setting row status to active DONE.

***** SUCCESSFULLY CREATED NEW ROW IN AGENT FOR USER : [myNewUser]*****


Send sanity check? Your manager.security file MUST contain the currently
created user (press return to do it, "no" to skip):
```

6. **Press `Enter` to perform the sanity check.**

   You should see the following confirmation:

```
SANITY CHECK SUCCESSFUL, SPIN LOCK VALUE IS: 5
Ready for new user inputs.

Type the engine Id (return to accept) [0x000000000000000000000002]:
```

   You are then invited to provide configuration information for any other users you want to allow to access `Agent`.

7. **Check that the new user has been granted access to the agent by looking at the agent's `jdmk.security` file.**

   You should see a new `userEntry` for the new user in the `jdmk.security` file.

EXAMPLE 19–11 `jdmk.security` for `Agent` File after Running `CreateUsmMibUser`

```
localEngineID=0x000000000000000000000002
localEngineBoots=7

userEntry=0x000000000000000000000002,myNewUser,myNewUser,
usmHMACMD5AuthProtocol,0x87021D7BD9D101BA05EA6E3BF9D9BD4A,
usmDESPrivProtocol,0x87021D7BD9D101BA05EA6E3BF9D9BD4A,3,

userEntry=localEngineID,defaultUser,,usmHMACMD5AuthProtocol,maplesyrup,
```

EXAMPLE 19–11 `jdmk.security` for `Agent` File after Running `CreateUsmMibUser` *(Continued)*

```
usmDESPrivProtocol,maplesyrup,3,true
```

8. **When you have added all your new users, press `Control C` in both windows to stop both `Agent` and `CreateUsmMibUser`**

19.4 Legacy SNMP Security

Because Java DMK 5.1 implements an SNMPv3 adaptor, and all SNMPv3 security aspects are handled completely by this adaptor, MIB instrumentation does not depend on the version of SNMP via which it is accessed. MIBs that were developed under previous releases of Java DMK can thus be directly registered into the new `SnmpV3AdaptorServer`, and benefit from all the SNMPv3 security mechanisms.

However, earlier versions of Java DMK provided a hook via the `SnmpPduFactory`, that enabled the implementation of authentication and encryption on top of the SNMPv1 and v2 protocol adaptor. This can be used to implement proprietary security over the regular SNMPv1 and v2 PDUs. This hook has been preserved in Java DMK 5.1, for reasons of backwards compatibility.

Note – Although the SNMPv1 and v2 community-based security mechanism is still available in Java DMK 5.1, you should migrate applications that require better security to SNMPv3. When migrating your applications to SNMPv3, applications which have implemented their own PDU factory must be revised before they can be imported into the `SnmpV3AdaptorServer`, as the `SnmpPduFactory` class developed for SNMPv1/v2 PDUs is not compatible with SNMPv3 PDUs.

19.4.1 Decoding and Encoding SNMP Packets

The `SnmpPduFactory` hook provided by Java DMK 5.1 involves the following Java classes:

`com.sun.management.snmp.SnmpPduFactory`
Defines the interface of the object in charge of encoding and decoding SNMP packets.

`com.sun.jdmk.snmp.SnmpPduFactoryBER`
The default implementation of the `SnmpPduFactory` interface.

`com.sun.management.snmp.SnmpPdu`
The fully decoded representation of an SNMP packet.

`com.sun.management.snmp.SnmpMsg`
A partially decoded representation of an SNMP packet, containing the information stored in any SNMPv1, SNMPv2 or SNMPv3 message.

After receiving an SNMP packet, Java DMK 5.1 performs the following steps:

1. The received bytes are translated into an `SnmpMsg` object by the message processing subsystem. If the SNMP protocol version of the original request was either v1 or v2, this step simply involves the BER decoding of the ASN.1 Message sequence as defined in RFC 1901. If the SNMP protocol version of the original request was v3, the message processing subsystem will in addition invoke the security subsystem to authenticate and decrypt the message.
2. The `SnmpMsg` object is then translated into an `SnmpPdu` object.
3. The `SnmpPdu` is analyzed and the corresponding operation is performed.

Before sending an SNMP packet, Java DMK 5.1 performs the following steps:

1. An `SnmpPdu` object is initialized according to the requested operation. This could be either an `SnmpPduPacket`for SNMPv1or v2, or an `SnmpScopedPduPacket` for SNMPv3.
2. The `SnmpPdu` object is translated into an `SnmpMsg`.
3. The `SnmpMsg` is then passed to the message processing subsystem, and translated into bytes. If SNMPv1 or SNMPv2 is being used, this step simply involves the BER encoding of the ASN.1 message sequence as defined in RFC 1901. If SNMPv3 is being used, the message processing subsystem also invokes the security subsystem to sign and encrypt the message.

The `SnmpPdu` object is the fully decoded description of the SNMP request. In particular, it includes the operation type (`get`, `set`, and so on), the list of variables to be operated upon, the request identifier, and the protocol version, as shown in Example 19–12.

EXAMPLE 19–12 Using the `SnmpPdu` Class

```
abstract class SnmpPdu {
             ...
             public int version ;
             public int type ;
             public int requestId ;
             public SnmpVarBind[] varBindList ;
             ...
     }
```

The use of the `SnmpMsg` class is shown in Example 19–13. The `SnmpMsg` class is a partially decoded representation of the SNMP request. Only the protocol version and security parameters are decoded. All the other parameters remain encoded.

The SnmpMsg class is the base class derived from the message syntax from RFC 1157 and RFC 1901, and SNMPv3Message from RFC 2572. The SnmpMessage class that was present in releases of Java DMK before 5.0 derives from SnmpMsg and represents an SNMPv1 or SNMPv2 message. Because SNMPv3 introduces additional security parameters, the SnmpMessage class can only be used for SNMPv1 or SNMPv2 messages. SnmpPduFactory implementations that make direct use of SnmpMessage will therefore need to be updated if they are to be imported into a Java DMK 5.1 SNMPv3 protocol adaptor. However, they do not need to be changed as long if the old SnmpAdaptorServer is used instead of SnmpV3AdaptorServer.

EXAMPLE 19–13 Using the SnmpMsg Class

```
abstract class SnmpMsg {
             ...
             public int version ;
             ...
     }

     class SnmpMessage extends SnmpMsg {
             ...
             public byte[] community ;
             public byte[] data ;
             ...
     }
```

19.4.2 SnmpPduFactory Interface

When Java DMK 5.1 needs to translate an SnmpMsg object into an SnmpPdu object, it delegates this task to an object which implements SnmpPduFactory, as shown in Example 19–14.

EXAMPLE 19–14 Using the SnmpPduFactory Interface

```
interface SnmpPduFactory {

       // Makes an SnmpPdu from an SnmpMsg
       public SnmpPdu decodeSnmpPdu(SnmpMsg msg)
       throws SnmpStatusException ;

       // Makes an SnmpMsg from an SnmpPdu
       public SnmpMsg encodeSnmpPdu(SnmpPdu pdu, int maxPktSize)
       throws SnmpStatusException, SnmpTooBigException ;

     }
```

Note – SnmpPduFactory has two additional methods inherited from Java DMK 4.2, decodePdu and encodePdu, that are now deprecated but are kept for backwards compatibility.

Java DMK 5.1 provides a default implementation of theSnmpPduFactory, called SnmpPduFactoryBER. SnmpPduFactoryBER is used automatically unless stated otherwise. The SnmpPduFactoryBER methods control every incoming or outgoing SNMP PDU.

- If decodeSnmpPdu() returns null, Java DMK will assume that the SnmpMsg is unsafe and will refuse it
- If encodeSnmpPdu() returns null, Java DMK will assume that it cannot send the SnmpPdu safely and will abort the current request

Therefore, it is possible to implement a security policy using an SnmpPduFactory class. However, it is recommended to rely rather on the standard SNMPv3 policy. Using the SnmpPduFactory to implement additional levels of security only makes sense on an SNMPv1 or SNMPv2 framework, when SNMPv3 is not an option.

19.4.3 Implementing a New SnmpPduFactory Class

Java DMK expects decodeSnmpPdu to behave as follows:

- Decode the SnmpMsg object and return a fully initialized SnmpPdu.
- Return null if it judges the SnmpMsg object to be unsafe; in this case, Java DMK will refuse the SnmpMsg object.
- Throw an SnmpStatusException if the decoding failed or if the PDU contains illegal values; in this case, Java DMK will refuse the SnmpMsg object.

Java DMK expects encodeSnmpPdu to behave as follows:

- Encode the SnmpPdu object and return a fully initialized SnmpMsg object.
- Throw an SnmpStatusException if the SnmpPdu object contains illegal values.
- Throw an SnmpTooBigException if the SnmpPdu object does not fit into the internal buffer used by Java DMK.
- Return null if it fails to secure the SnmpPdu object; in this case, Java DMK aborts the current request and reports an error. This probably means that the agent or the manager contains a bug.

Because SnmpPdu and SnmpMsg are abstract classes, you should delegate their creation and initialization to an instance of SnmpPduFactoryBER and work on the result returned.

You can change the SnmpPduFactory object used by the SNMP adaptor by using the setPduFactory method, shown in Example 19–15.

EXAMPLE 19–15 Changing the SnmpPduFactory object Using setPduFactory

```
            ...
        myAdaptor.setPduFactory(new MyFireWallPduFactory()) ;
        ...
```

In Java DMK 4.2, the SnmpPduFactory was attached to the SnmpPeer object. In Java DMK 5.1, the SnmpPduFactory is attached to the SnmpSession. Factories set via the deprecated SnmpPeer API are reused in Java DMK 5.0. They can be changed using the setPduFactory method, as shown in Example 19–16.

EXAMPLE 19–16 Updating Deprecated SnmpPeer Factories Using setPduFactory

```
...
SnmpSession mySession = new SnmpSession() ;
mySession.setPduFactory(new MyFireWallPduFactory()) ;
mySession.snmpGet(myPeer, this, myVarBindList) ;
...
```

Caution – Setting two different factories in the peer and in the session can lead to unpredictable behavior. Use the same factory at both levels.

CHAPTER 20

SNMP Master Agent

It is easier to manage a large number of simple network management protocol (SNMP) agents when they have a hierarchical structure of master agents and subagents. Master agents concentrate and relay the information in their subagents and can provide their own specific information as well. Managers only communicate with the master agents and access the subagents transparently, as if the information actually resided in the master agent.

This chapter covers the following topics:

- "20.1 SNMP Master Agent and the SNMPv3 Proxy Forwarder" on page 388 explains the differences between the Java Dynamic Management Kit (Java DMK) 5.1 SNMP master agent and the SNMPv3 proxy forwarder
- "20.2 Overview of the SNMP Master Agent" on page 389 provides a description of the different Java DMK 5.1 functions that make up the SNMP master agent
- "20.3 Routing Overlapping MIBs" on page 391 explains the concept of overlapping MIBs
- "20.4 MIB Scoping in Master Agents" on page 393 details how scoped MIBs are used by the master agent
- "20.5 Trap Forwarding" on page 393 describes how master agents listen for traps and forward them to managers
- "20.6 Protocol Translation" on page 395 describes how the master agent translates requests between different versions of the SNMP protocol
- "20.7 SNMP Master Agent Examples" on page 402 explains how to run the SNMP master agent example provided with Java DMK 5.1

20.1 SNMP Master Agent and the SNMPv3 Proxy Forwarder

The Java DMK 5.1 SNMP master agent is clearly distinct from the SNMPv3 proxy forwarder application as described in RFC 2573 *SNMPv3 Applications*, and must not be confused with it. The distinction is as follows:

- An SNMP master agent makes it possible to route parts of an SNMP request to a number of subagents. The subagents are completely hidden from the manager. The manager sees a single SNMP entity, the master agent. In an SNMPv3 framework, the manager knows only one SNMP engine ID, namely that of the master agent. A single SNMP request might trigger the retrieval of information from several subagents. How the information is dispatched and retrieved from the subagents is completely controlled by the master agent and is hidden from the manager.
- On the contrary, in a proxy forwarder application, the manager needs to be aware of all the proxied agents. It needs to build a request targeted to a single specific subagent, and it needs to include in the request some information that enables the proxy forwarder application to determine which subagent is the target of the request. In SNMPv3, this is the subagent's SNMP engine ID. In SNMPv1 and SNMPv2, it is a specific community string, for example `public@bridge2`, where `bridge2` identifies the subagent to which the request is forwarded.

The SNMP master agent functionality is thus particularly dedicated to the integration of legacy SNMP agents, in cases where the manager must see only a single entity. An example of such an SNMP master agent is `snmpdx` on the Solaris operating environment. The proxy forwarder application is more generally used as a firewall, to route information from a public node to agents running in a private network, or to add an additional security layer (for example, SNMPv3 security) to unsecured agents.

Both applications, master agent and proxy forwarder, share a common set of functions, for example, transferring information from one version of the SNMP protocol to another. However, their scope of application is different, and the problems they address are distinct. Java DMK 5.1 supports only the master agent, and does not support the proxy forwarder.

20.2 Overview of the SNMP Master Agent

The SNMP master agent function is available on the SnmpV3AdaptorServer. As the SnmpV3AdaptorServer is trilingual, a master agent can answer requests in SNMPv1, SNMPv2 or SNMPv3. The master agent function is based on the functional blocks of the Java DMK 5.1 SNMP toolkit, particularly:

- SnmpV3AdaptorServer
- SnmpProxy
- SnmpUsmProxy
- SnmpTrapForwarder

20.2.1 SnmpV3AdaptorServer

The SnmpV3AdaptorServer makes it possible to register an SnmpMibAgent that handles all varbinds whose OIDs are contained below a given node of the global OID tree. For each SNMP request, all varbinds that are relevant to the part of the OID tree handled by the SnmpMibAgent are grouped into an SnmpMibRequest and passed to the SnmpMibAgent. The concrete SnmpMibAgent implementation can be either a local MIB generated by the mibgen tool, or a remote MIB, configured with an SnmpProxy object.

The SnmpV3AdaptorServer enables you to register overlapping MIBs. The actual value returned to the manager is the value implemented by the deepest MIB registered for that value. This makes it possible to register, for example, a default SnmpMibAgent to which all unhandled OIDs are forwarded. An example of such a default subagent can be the Solaris subagent snmpdx, all the OIDs of which are neither handled locally nor handled by known subagents. This makes it possible to forward them to an SnmpProxy object that forwards to snmpdx and is registered at the root of the OID tree. This only requires registering such an SnmpProxy object at the root of the OID tree.

The SnmpV3AdaptorServer supports MIB scoping, namely, context names, and makes it possible to register a MIB within a given scope, that is, for a given context name. The SnmpV3AdaptorServer also makes it possible to extract context names from an SNMPv1 or SNMPv2 community string. The master agent function can thus be spawned over several context names. A specific context name can be dedicated to a specific subagent, enabling you to register a single SnmpProxy object at the root of the OID tree for a given context name. The context name can also be shared between several subagents, and possibly local MIBs, as described above.

20.2.2 SnmpProxy

The SnmpProxy object makes it possible to get the actual values requested by the manager from a MIB implemented in a remote SNMP subagent. The SnmpProxy objects are able to get values from SNMP agents implementing either of the SNMPv1 or SNMPv2 protocols. This makes it possible to get values from legacy SNMP agents in a way that is transparent to the manager.

The SnmpProxy object extends the SnmpMibAgent abstract class, and as such can be registered in the SnmpV3AdaptorServer just like a local MIB generated by the mibgen tool. In a master agent application, one SnmpProxy must be created per subagent. When creating the SnmpProxy object, you must provide it with the following parameters to enable it to access the underlying subagent:

- Subagent's IP address
- Subagent's port number
- Security parameters, namely community strings, used for communicating with the subagent
- Version of SNMP that must be used to access the subagent, namely either SNMPv1 or SNMPv2

Then you must register that proxy with the SnmpV3AdaptorServer, once for each MIB implemented by the subagent that you want to mirror in the master agent. This is demonstrated in Example 20–1.

EXAMPLE 20–1 Registering SnmpProxy in the SnmpV3AdaptorServer

A subagent implements the following three MIBs:

- mib#1 (root OID=x.y.z.1)
- mib#2 (root OID=x.y.z.2)
- mib#3 (root OID=x.y.z.3)

If you want mib#1 and mib#3 to be accessible from the master agent managers, then you need to register your SnmpProxy object for both mib#1 and mib#3:

```
SnmpV3AdaptorServer snmpAdaptor = ....;
SnmpProxy mySubAgentProxy = new SnmpProxy(....);
SnmpOid oids[] = new SnmpOid(2);
oids[0] = new SnmpOid("x.y.z.1");
oids[1] = new SnmpOid("x.y.z.3");
snmpAdaptor.addMib(mySubAgentProxy,oids);
```

Once you have done this, all varbinds contained in an SNMP request whose OIDs appear to be below x.y.z.1 or x.y.z.3 are passed to the registered mySubAgentProxy. The mySubAgentProxy proxy gets the actual values from the remote agent it is proxying. OIDs that appear to be contained below x.y.z.2 are not forwarded, and where applicable, are handled locally, because the SnmpProxy is not registered for those OIDs.

20.2.3 SnmpUsmProxy

The SnmpUsmProxy object performs the same function as the SnmpProxy, except that it gets values from SNMP agents implementing the SNMPv3 protocol.

The SnmpUsmProxy object extends the SnmpUsmMibAgent abstract class, and as such can be registered in the SnmpV3AdaptorServer just like a local MIB generated by the mibgen tool. In a master agent application, one SnmpUsmProxy must be created per subagent. When creating the SnmpUsmProxy object, you must provide it with the following parameters to enable it to access the underlying subagent:

- Subagent's IP address
- Subagent's port number
- Security parameters, namely context names, the user name and the security level, used for communicating with the subagent
- Request context, namely the context engine ID and the context name

The SnmpUsmProxy receives requests in any version of SNMP, and always forwards them as SNMPv3 requests.

20.2.4 SnmpTrapForwarder

An SnmpTrapForwarder object makes it possible to listen to traps emitted by a subagent, convert them into the master agent's manager's protocol or protocols, and forward them to the manager or managers as if they originated from the master agent itself. One or more SnmpTrapForwarder objects must be instantiated, and the subagents must be configured to send their traps to these objects. This is the responsibility of the main application code. This is explained in detail in "20.5 Trap Forwarding" on page 393.

20.3 Routing Overlapping MIBs

In a master agent application, routing is performed depending on the OIDs requested. SNMP MIB agents are registered for branches of the global OID tree, and varbinds are dispatched to subagents according to the requested OIDs. The SnmpV3AdaptorServer supports registration of overlapping MIBs. Overlapping MIBs are MIBs that are registered below the OID of a previously registered MIB. The actual value returned to the manager is the value implemented by the deepest MIB registered for that value. When a get or set request is received, for each varbind in the request, the SnmpV3AdaptorServer identifies the deepest MIB responsible for handling the OID contained in the varbind. All varbinds targeted to that MIB are grouped in an SnmpMibRequest, and that request is passed to the identified SnmpMibAgent.

20.3.1 Shadowing Overlapping MIBs

MIB overlapping enables you to shadow MIBs that are implemented in subagents. For instance, you might want to implement the system group and snmp from MIB-II locally in your master agent, and delegate all other requests concerning MIB-II to a legacy SNMP agent implementing MIB-II. You do this as described in "To Shadow Overlapping MIBs" on page 392.

▼ To Shadow Overlapping MIBs

1. **Run the mibgen tool to generate an implementation of MIB-II**

2. **Instrument the system group and snmp group as shown in the basic SNMP agent example in "Creating an SNMP Agent"**

3. **Register your generated MIB with the OIDs of both the system group, 1.3.6.1.2.1, and snmp group, 1.3.6.1.2.1.1.1**

4. **Create an SnmpProxy object proxying your legacy MIB-II agent, and register that proxy for the root OID of MIB-II**

20.3.2 Delegation and Precedence of Overlapping MIBs

MIB overlapping also enables you to delegate all unhandled OIDs to a legacy master agent, such as the Solaris agent snmpdx. If you register an SnmpProxy at the root of the OID tree, then all varbinds for whose OIDs no SnmpMibAgent was registered are passed to that SnmpProxy.

In addition, MIB overlapping enables you to give precedence to a specific agent for a part of a MIB, by registering it deeper in the OID tree.

Note – For a `get-next` or `get-bulk` request, the incoming OID does not make it possible to identify directly the subagent that will contain the next OID. Indeed, the next OID might not be a sub-OID of the requested OID, but might be in a different part of the global OID tree. The `get-next` and `get-bulk` requests are thus forwarded to all subagents, and sorted a posteriori to identify the deepest matching subagent.

20.4 MIB Scoping in Master Agents

MIB scoping is a special case in which the SNMP proxy forwarder application can be emulated by a master agent application. If a subagent is dedicated completely to handling a specific MIB scope, you register a single `SnmpProxy` for that subagent at the root of the OID tree for that scope. Because there can be only one context per request, then all or none of the varbinds in the incoming request are passed to the `SnmpProxy`. In this case, the master agent application behaves similarly to a proxy forwarder application. The only difference is that it identifies the target agent with the context name contained in the request.

20.5 Trap Forwarding

Another function of an SNMP master agent is to listen to SNMP traps and notifications emitted by its subagents, and to forward them to its managers. The Java DMK 5.1 SNMP toolkit provides an `SnmpTrapForwarder` for that purpose.

How to configure the trap forwarding mechanism is entirely the responsibility of your main application code. The following are three possibilities for ways to configure your trap forwarding mechanism, but the list is by no means exhaustive.

- Create a single `SnmpTrapForwarder` and make all subagents to send traps to that forwarder
- Create one forwarder per subagent, each listening to a different port for each subagent
- Create one `SnmpTrapForwarder` per manager, each one forwarding traps to a specific manager

The `SnmpTrapForwarder` handles protocol conversion. It is able to convert any of an SNMPv1 trap, or an SNMPv2 or SNMPv3 notification into any of an SNMPv1 trap or an SNMPv2 or SNMPv3 notification following the protocol conversion rules specified in RFC 2576.

20.5.1 Configuration of Trap Targets

You can pass specific manager addresses when forwarding for a specific protocol. By default the `InetAddressAcl` is also parsed. So the set of actual manager addresses is the trap blocks located in the `acl` file and the set of added targets. You can disable the `InetAddressAcl` parsing by calling the method `setInetAddressAclUsed (false)`.

To enable trap forwarding you must start the trap forwarder.

⚠ **Caution –** If you have manager addresses in your set of targets that are also present in `InetAddressAcl`, or you have no `InetAddressAcl` activated but are targeting the local host, your manager will receive the trap twice. To prevent this, configure the `SnmpTrapForwarder` carefully. You can, for example, disable the `setInetAddressAcl` parsing by calling `isInetAddressAclUsed`(`false`).

20.5.2 Proxy Forwarding and Notification Originators

By default a trap is sent as a notification originator. The difference between acting as a notification originator or acting as a proxy when forwarding the trap is detailed in RFC 2576 *"Coexistence Between SNMP versions"*

20.5.2.1 Proxy Forwarding

Proxy forwarding is activated by calling the `forwardAsProxy()` method.

When translating SNMPv1 traps into SNMPv2 traps, the varbind received is reused. Three additional varbinds are appended, if these three additional varbinds do not already exist in the SNMPv1 varbinds:

- The name portion of the first additional varbind contains `snmpTrapAddress.0`, and the value contains the SNMPv1 `agent-addr` parameter.
- The name portion of the second additional varbind contains `snmpTrapCommunity.0`, and the value contains the value of the community string field from the received SNMPv1 message which contained the SNMPv1 trap PDU.
- The name portion of the third additional variable binding contains `snmpTrapEnterprise.0`, and the value is the SNMPv1 enterprise parameter.

The SNMPv1 agent-addr parameter is determined by the proxy extracting the original source of the notification from the varbinds, If the SNMPv2 varbinds contain a varbind whose name is snmpTrapAddress.0, the agent-addr parameter is set to the value of that varbind. Otherwise, the SNMPv1 agent-addr parameter is set to 0.0.0.0.

20.5.2.2 Notification Originators

Traps are forwarded as notification originators by calling the forwardAsNotificationOriginator() method.

When translating from SNMPv1 to SNMPv2, the SNMPv2 varbinds are the same as the SNMPv1 varbinds.

When translating from SNMPv2 to SNMPv1, the SNMPv1 agent-addr parameter is determined as follow:

- If the notification is sent over IP, the SNMPv1 agent-addr parameter is set to the IP address of the SNMP entity in which the notification originator resides.
- If the notification is sent over some other transport protocol, the SNMPv1 agent-addr parameter is set to 0.0.0.0.

20.6 Protocol Translation

As explained in the previous section, both the SnmpProxy and SnmpTrapForwarder are compliant with RFC 2576. However, RFC 2576 does not simply describe protocol conversion, but rather describes how protocol conversion is performed in the scope of a proxy forwarding application. In a master agent application, the question is more complex because we want the master agent to behave as a single agent responding with the same protocol as is used by the manager. We also want to hide the presence of subagents from the manager.

When looking from an end-to-end perspective, the master agent does not strictly behave as described in RFC 2576. This is because the protocol translation applied by the SnmpProxy is post-processed and aggregated in the master agent with results coming from other SnmpMibAgents, such as SnmpProxy objects and mibgen generated local MIBs, each possibly translating from different protocol versions.

The main differences, when looking from an end to end perspective, between protocol translation in the SNMP master agent and RFC 2576 are the following:

- When the incoming request is an SNMPv2 or SNMPv3 request, the error codes returned are SNMPv2 error codes, when applicable, even if the subagent from which the error originally came is a SNMPv1 agent.

- When the incoming request is an SNMPv2 or SNMPv3 request, `get`, `get-next` and `get-bulk` always transform global SNMPv1 errors into a varbind value of `noSuchInstance`, `noSuchObject`, or `endOfMibView`, and skip to the next varbind; this is the same behavior as for an SNMPv2/v3 agent.
- If an exception is raised or an error is returned by a subagent during the `set()` phase of a `set` request, then the error returned is always `undoFailed`. In fact, there is no way to apply a two-phased `check()/set()` action to a remote SNMP agent because the SNMP protocol does not have any CHECK primitive. Consequently, the atomicity of a `set` request is only guaranteed when all the OIDs in the request are processed by the same SNMP entity. See "20.6.3 Atomicity and Error Handling" on page 397 for details.

  When creating an `SnmpProxy` object, the application must decide whether the remote request is sent during the `check()` phase or the `set()` phase of the `set` operation. If the operation is performed during the `check()` phase, the error returned to the manager is that which is emitted by the subagent. However, the atomicity is no longer guaranteed, because the `check()` phase can now actually modify some of the sub agent MIBs. According to SNMPv2, the error returned in that case should be `undoFailed`, but in fact it is whatever error was returned or raised by the `SnmpMibAgent`. If the operation is performed during the `set()` phase, the error returned is always `undoFailed` for SNMPv2 or SNMPv3, or `genError` for SNMPv1. This will hide the actual error that was returned by the subagent. Whether one method or the other is used is entirely the responsibility of the application code.

20.6.1 SNMP Proxy Translation

The `SnmpProxy` object implements the following translation:

- `SnmpProxy` forwards the request in the protocol specified by the subagent; a `get` request makes the `SnmpProxy` object send a `get` to the subagent, a `set` request makes the `SnmpProxy` object send a `set` to the subagent, and so on.
- If the incoming request is a `get-bulk` request, and the subagent is an SNMPv1 subagent, `SnmpProxy` forwards the request as a series of `get-next` requests.
- If the incoming request is an SNMPv2 or SNMPv3 `get` or `get-next` request, and the subagent is an SNMPv1 subagent, `SnmpProxy` subtracts the varbinds for which an error was returned, and resends the request until either a value or an error is obtained for each varbind. The requests are sent to emulate SNMPv2 or SNMPv3 `get` and `get-next` behavior.
- If the incoming request is a `set`, then two types of behavior are possible, depending on how the `SnmpProxy` object is configured:
  - The `set` request can be performed during the `check` phase, in which case nothing is done during the `set`
  - The `set` request can be performed during the `set` phase, in which case nothing is performed during the `check`; this is the default behavior

Whichever solution is chosen has different impacts on the atomicity of set requests. See “20.6.3 Atomicity and Error Handling” on page 397 for details.

- If the incoming request contains SNMPv1 or SNMPv2 community strings, they must be translated using the translateParameters() method. The translateParameters() method reuses the information contained in the SnmpMibRequest PDU to construct new SnmpParams.

  You can overload this method to change its behavior.
- When an SNMPv1 or SNMPv2 request is received, the translateParameters() method is called. The received community string is reused in the forwarded request. If an SNMPv3 request is received, the community string is forwarded as public in a get request and as private in a set request.

20.6.2 SNMP USM Proxy Translation

The SnmpUsmProxy object inherits from SnmpProxy. When a request is received, the translateParameters() method is called.

If the request is an SNMPv1 or SNMPv2 request, the security level is set to noAuthNoPriv. If the received community string is of the form community@context, the context name is set to the received context value. If it is in any other form it is null.

If the request is an SNMPv3 request, the received context, security level, and other values are reused.

20.6.3 Atomicity and Error Handling

As stated in the previous section, the atomicity of set requests is no longer guaranteed when remote MIBs are involved. Although some strategies exist that try to offer a best-effort regarding the atomicity of set requests, there is no generic mechanism that is guaranteed to work in a master agent application. The best that can be done in a generic toolkit is to identify those cases where atomicity might have been broken, and to inform the manager of that situation. Java DMK 5.1 handles this by responding with undoFailed when an error occurs during the set() phase of a set request. In its default configuration, when an SNMPv2 set request is received, Java DMK guarantees that undoFailed is sent when atomicity might have been broken. This no longer holds if the application code configures the SnmpProxy object to send the remote SNMP set request during the check() phase of the set operation.

Some toolkits attempt to implement atomicity by:

1. Getting the current values of all variables included in the set.
2. Performing the set.
3. Reverting to the old values by sending a second set with the values obtained in 1 above, or the values obtained in 2 if no error is returned.

Although this might seem more satisfactory it is not guaranteed to work. Depending on the semantics of the variables involved in the set, several things might happen:

- If the transition from value#1 to value#2 is valid, there is no guarantee that the transition from value#2 to value#1 will be accepted by the agent; reverting to the old value might not be possible
- Setting an object to a specific value might already have had unrecoverable effects on the agent; for example removing a resource, destroying a row, and so on
- Some objects might be of type *Test-And-Increment* which means that getting their value in 1 above already modifies them. Trying to perform a set on them will probably generate an error as these objects are usually read-only. In the end, the whole process will return an error while still having modified the objects as a side effect

Even if no generic mechanism is supported, the Java DMK can still be customized to implement any specific behavior. The SnmpUserDataFactory makes it possible to allocate and release contextual data about requests, which can be used to implement transactional behavior. By subclassing SnmpProxy, and mibgen–generated metadata classes, any kind of specific transactional behavior can be implemented. However, no generic solution exists, and if transactional behavior can be implemented, it is specific to the semantics of the objects contained in the application and subagent MIBs.

Another special case is when a subagent is entirely responsible for a given context scope. In that case, the atomicity of set requests can still be achieved by performing the remote SNMP set during the check() phase. See "20.4 MIB Scoping in Master Agents" on page 393 for details.

The following tables show the end-to-end error translation performed by a master agent application.

TABLE 20–1 Error Translation from SNMPv1 Subagents to SNMPv2/v3 Managers

PDU Type	Error From SNMPv1 Subagents	Error Sent to SNMPv2/v3 Managers
get	genErr	genErr
get	noSuchName	noError => noSuchInstance in varbind
get	tooBig	handled by stack =>resend or tooBig
get	any other error	genErr
set	any error	undoFailed(**)
get-next/get-bulk	genErr(*)	genErr

TABLE 20–1 Error Translation from SNMPv1 Subagents to SNMPv2/v3 Managers *(Continued)*

PDU Type	Error From SNMPv1 Subagents	Error Sent to SNMPv2/v3 Managers
get-next/get-bulk	noSuchName	noError => skip to next SnmpMibAgent or endOfMibView if none
get-next/get-bulk	tooBig	handled by stack resend or tooBig cated response (get-bulk)
get-next/get-bulk	any other error(*)	genErr

(*) The SnmpProxy objects can be configured to hide such errors. In this case the master agent skips to the next SnmpMibAgent. This behavior can be very useful when dealing with faulty legacy agents.

(**) See Table 20–5.

TABLE 20–2 Error Translation from SNMPv2/v3 Subagents to v1 Managers

PDU Type	Error From SNMPv2/v3 Subagents	Error Sent to SNMPv1 Managers
get	genErr	genErr
get	noError => noSuchInstance in varbind	noSuchName
get	noError => noSuchInstance in varbind	noSuchName
get	tooBig	handled by stack => resend or tooBig
get	any other error	genErr
set	any error	genErr (**) - undoFailed is translated into genErr
get-next	genErr	genErr
get-next	noError => endOfMibView in varbind	noSuchName
get-next	tooBig	handled by stack => resend or tooBig
get-next	any other error	genErr

(**) See Table 20–5.

TABLE 20–3 Error Translation From SNMPv1 Subagents to SNMPv1 Managers

PDU Type	Error From SNMPv1 Subagents	Error Sent to SNMPv1 Managers
get	genErr	genErr
get	noSuchName	noSuchName
get	tooBig	handled by stack => resend or tooBig
get	any other error	genErr
set	any error	genErr (**) - undoFailed is translated into genErr
get-next	genErr	genErr
get-next	noSuchName	noSuchName
get-next	tooBig	handled by stack => resend or tooBig
get-next	any other error	genErr

(**) See Table 20–5.

TABLE 20–4 Error Translation from SNMPv2/v3 Subagents to SNMPv2/v3 Managers

PDU Type	Error From SNMPv2/v3 Subagents	Error Sent to SNMPv2/v3 Managers
get	noError	noError
get	tooBig	handled by stack => resend or tooBig
get	any other error	same error (if valid) or genErr
set	any error	undoFailed (**)
get-next/get-bulk	noError	noError
get-next/get-bulk	tooBig	handled by stack => resend or tooBig or truncated response (GET-BULK)
get-next/get-bulk	any other error	same error (if valid) or genErr

(**) By default SnmpProxy sends the remote set request during the set() phase of the set operation. When an error occurs during the set() phase, undoFailed is returned to the manager because the atomicity is no longer guaranteed. Note that in the special case where an SnmpProxy is configured to perform the remote set request during the check() phase of the set operation, the following translation is applied for errors returned by the remote set request, even if the atomicity of the set request is broken, as shown in the following table.

TABLE 20–5 Error Translation When SnmpProxy Performs Remote set

Before Translation	After Translation
v1 errorStatus	**v2/v3 errorStatus**
noError	noError
noSuchName	noSuchName
genErr	genErr
badValue	wrongValue
readOnly	wrongValue
v2/v3 errorStatus	**v1 errorStatus**
noError	noError
genErr	genErr
wrongValue, wrongEncoding, wrongLength, wrongType, inconsistentValue	badValue
noAccess, notWritable, noCreation, inconsistentName, authorizationError	noSuchName
resourceUnavailable, commitFailed, undoFailed, any other error	genErr
v1 errorStatus	**v1 errorStatus**
any error	same error (if valid); genErr otherwise
v2/v3 errorStatus	**v2/v3 errorStatus**
any error	same error (if valid); genErr otherwise
v1 errorStatus	**v2/v3 errorStatus**
noError	noError
noSuchName	noSuchName
genErr	genErr
badValue	wrongValue
readOnly	wrongValue

TABLE 20–5 Error Translation When `SnmpProxy` Performs Remote `set` *(Continued)*

Before Translation	After Translation
v2/v3 errorStatus	**v1 errorStatus**
`noError`	`noError`
`genErr`	`genErr`
`wrongValue, wrongEncoding, wrongLength, wrongType, inconsistentValue`	`badValue`
`noAccess, notWritable, noCreation, inconsistentName, authorizationError`	`noSuchName`
`resourceUnavailable, commitFailed, undoFailed, any other error`	`genErr`
v1 errorStatus	**v1 errorStatus**
any error	same error (if valid); `genErr` otherwise
v2/v3 errorStatus	**v2/v3 errorStatus**
any error	same error (if valid); `genErr` otherwise

20.7 SNMP Master Agent Examples

These examples show you how to register local and remote MIBs within the master agent SNMP adaptor. They demonstrate how to instantiate and use the `SnmpProxy` and `SnmpUsmProxy` classes to register the remote MIBs.

In these examples, we demonstrate two different types of SNMP master agent that delegate handling of SNMP requests to subagents:

- A master agent, which hides a single subagent and contains a single remote MIB implemented by that subagent
- An overlap master agent, which locally implements part of a MIB whose OID space overlaps with the MIB implemented by its subagent

These examples also contain a manager which is used to send requests to the master agent. This manager operates in synchronous mode. Manager requests are sent both in SNMPv2 and SNMPv3.

Before going through this example, you must be familiar with the agent and manager examples described in Chapter 16 and Chapter 17.

The following applications are used in these examples:

`SimpleManager`

A manager used to query the master agent. This class is in the *examplesDir*`/current/Snmp/MasterAgent/manager` directory. The `SimpleManager` application makes SNMP requests both in SNMPv3 and in SNMPv2 on a Java DMK master agent, and listens to forwarded traps.

`StandAloneAgent`

A Java DMK SNMPv1/v2 standalone subagent. This class is in the *examplesDir*`/current/Snmp/MasterAgent/standalone` directory. The `StandAloneAgent` application is a multi-protocol SNMP agent implementing a simple MIB containing four OIDs:

- `sysDescr1 value "error1"`
- `sysDescr2 value "error2"`
- `sysDescr3 value "sysDescr3"`
- `sysDescr4 value "sysDescr4"`

This simple MIB contains a single fake system group registered in the OID space of MIB-II.

`MasterAgent`

A simple SNMPv1/v2 master agent, configured with a single remote MIB implemented by a single subagent. This class is in the *examplesDir*`/current/Snmp/MasterAgent/master` directory. The `MasterAgent` application registers an `SnmpProxy` to forward requests to the remote MIB implemented by the `StandAloneAgent`, and forwards SNMPv1 and v2 traps to SNMPv1 and v2 managers.

`MasterAgentV3`

A simple SNMPv3 master agent, configured with a single remote MIB implemented by a single subagent. This class is in the *examplesDir*`/current/Snmp/MasterAgent/master` directory. The `MasterAgentV3` application registers an `SnmpUsmProxy` to forward requests to the remote MIB implemented by the `StandAloneAgent` and forwards SNMPv1 and SNMPv2 traps to SNMPv3 managers.

`MasterAgent`

A simple master agent that deals with MIBs whose OID spaces overlap. This class is in the *examplesDir*`/current/Snmp/MasterAgent/overlap` directory. The `MasterAgent` application registers an `SnmpProxy` to forward requests to the remote MIB implemented by the `StandAloneAgent` subagent. It also locally implements a MIB whose variables `sysDescr1` and `sysDescr2` with values `"sysDescr1"`, `"sysDescr2"` overlap with the variables defined in the subagent’s MIB.

The `SimpleManager` is identical to a standard Java DMK SNMP manager. The `StandAloneAgent` is also identical to a standard Java DMK SNMP standalone agent. It is unaware of the existence of the master agent, unless it is configured to send traps.

There are, however, differences in the way each of the SNMPv1/v2 master agent and the SNMPv3 master agent creates a proxy, as shown in Example 20–2 and Example 20–3. Example 20–4 shows how the MIB overlapping is programmed.

20.7.1 Proxy Creation in SNMPv1 and SNMPv2 Master Agents

In the SNMPv1/v2 master agent example, proxy registration is identical to standard MIB registration. The proxies are created as shown in the following example.

EXAMPLE 20–2 Proxy Creation in the `MasterAgent` Example

```
        [...]

         // First we need to create a peer to the distant sub agent.
        //
        final SnmpPeer peer =
         new SnmpPeer(host, Integer.parseInt(port));
        final SnmpParameters p = new SnmpParameters();

        // The sub agent is seen as a SNMPv2.
        //
        p.setProtocolVersion(SnmpDefinitions.snmpVersionTwo);

        //Set the parameters to the peer.
        //
        peer.setParams(p);

        // SnmpProxy creation.
        //
        final SnmpProxy proxy = new SnmpProxy(snmpAdaptor.getEngine(),
            peer,"1.3.6.1.2.1");

        // A SnmpProxy is also an MBean. Register it in the
        // MBeanServer.
        //
        proxyObjName= new ObjectName("snmp:class=SnmpProxy");
        server.registerMBean(proxy, proxyObjName);
        proxy.setSnmpAdaptor(snmpAdaptor);

[...]
```

To instantiate and register the proxy, the `MasterAgent` example first of all creates a peer to the subagent using `new SnmpPeer()`, and creates security parameters `p` using the `SnmpParameters()` method. It then sets the subagent as an SMNPv2 agent using `setProtocolVersion`.

The `MasterAgent` passes the parameters `p` to the new peer using `peer.setParams`. Finally, `MasterAgent` creates the proxy, passing it the following information:

- The adaptor's engine ID, via `snmpAdaptor.getEngine()`
- The peer
- The distant MIB's OID, `1.3.6.1.2.1`

20.7.2 Proxy Creation in SNMPv3 Master Agents

In the SNMPv3 master agent example, proxy registration is identical to standard MIB registration. The proxies are created as shown in the following example.

EXAMPLE 20–3 Proxy Creation in the `MasterAgentV3` Example

```
    [...]

     // First we need to create a peer to the distant SNMPv3 sub
    // agent
    final SnmpUsmPeer peer =
     new SnmpUsmPeer(snmpAdaptor.getEngine(),
    host, Integer.parseInt(port));

    // Forward the requests.
    //
    final SnmpUsmParameters p =
    new SnmpUsmParameters(snmpAdaptor.getEngine(),
                   "defaultUser");
    p.setContextEngineId(peer.getEngineId().getBytes());
    p.setSecurityLevel(SnmpDefinitions.authNoPriv);
    peer.setParams(p);
    peer.processUsmTimelinessDiscovery();

    final SnmpUsmProxy proxy =
new SnmpUsmProxy(snmpAdaptor.getEngine(),
   peer, "1.3.6.1.2.1");

    // A SnmpUsmProxy is also an MBean. Register it in the
    // MBeanServer.
    //
    proxyObjName= new ObjectName("snmp:class=SnmpUsmProxy");
    server.registerMBean(proxy, proxyObjName);
    proxy.setSnmpAdaptor(snmpAdaptor);

    [...]
```

To instantiate and register the proxy, the MasterAgentV3 example first of all creates a peer to the subagent using new SnmpUsmPeer(), and creates SNMPv3 security parameters p using the SnmpUsmParameters() method. These new security parameters are then passed to the peer using peer.setParams, thus translating all requests that come through the peer into SNMPv3 requests.

Finally, MasterAgent creates the proxy, passing it the following information:

- The adaptor's engine ID, via snmpAdaptor.getEngine()
- The peer
- The distant MIB's OID, 1.3.6.1.2.1

20.7.3 MIB Overlapping in Master Agents

In the overlapping MIB master agent example, proxy registration is identical to standard MIB registration and the proxies are created in the same way as for SNMPv1 and v2 master agents.

EXAMPLE 20–4 Overlapping MIBs MasterAgent Example

```
 // Instantiate a Mib
//
final MIB1_MIB mib1 = new MIB1_MIB();

// Register the local MIB1 implementation for sysDescr1
// and sysDescr2.
//
final SnmpOid oid1 = new SnmpOid("1.3.6.1.2.1.1.1");
final SnmpOid oid2 = new SnmpOid("1.3.6.1.2.1.1.2");

// Make an array of these OIDs.
//
final SnmpOid[] local_oids = {oid1, oid2};

// Add the local MIB1 implementation for each redefined oid
//
snmpAdaptor.addMib(mib1, local_oids);

// Initialize the mib
//
mib1.init();
```

The overlapping MIB MasterAgent example creates a new MIB using new MIB1_MIB. It then registers the local MIB1 implementation for sysDescr1 and sysDescr2, which are configured into the StandAloneAgent example as error1 and error2, respectively. The local MIB thus shadows the implementation of sysDescr1 and sysDescr2 instrumented in the remote MIB, because it is registered with deeper OIDs than the remote MIB.

The MIBs are then added to the SNMP adaptor server using `snmpAdaptor.addMib` and initialized using `init()`.

20.7.4 Running the SNMP Master Agent Examples

Before running the example applications, compile all the Java classes in each of the four subdirectories inside *examplesDir*/`current/Snmp/MasterAgent`, by typing the following command in each directory.

```
$ javac -classpath classpath -d . *.java
```

The following procedures give instructions to run five different SNMP master agent scenarios:

- The manager sends requests to the master agent with an SNMPv2 subagent.
- The manager sends requests to the master agent with an SNMPv3 subagent.
- The manager sends requests to the master agent, with overlapping.
- The manager receives forwarded SNMPv1 and SNMPv2 traps. Both the agent and manager must be running on the same host.
- The manager receives forwarded SNMPv3 traps.

Ensure that no agents are already running before you start the examples.

▼ To Send Requests From a Manager to the SNMPv2 Master Agent

Because there are no MIBs overlapping in this first example, all the variables implemented by the subagent MIB can be seen from the manager. Thus, you can see that some of the OIDs queried return a `DisplayString` value of `error#`.

1. **Start the `StandAloneAgent` subagent.**

   In *examplesDir*/`current/Snmp/MasterAgent/standalone`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   StandAloneAgent 8085
   ```

   This binds `StandAloneAgent` to port 8085. Do not send traps when prompted.

2. **Start the SNMPv2 `MasterAgent` application.**

   You need to provide `MasterAgent` with the following information:

   - The location of its security configuration file, `jdmk.security`
   - The port on which it should listen for incoming requests, in this case we choose 8087

- The subagent's host, in this case the local host
- The subagent's port number, in this case 8085

In *examplesDir*/`current/Snmp/MasterAgent/master`, type the following command:

```
$ java -classpath classpath -Djdmk.security.file=jdmk.security
MasterAgent 8087 localhost 8085
```

The following output is displayed:

```
NOTE: HTML adaptor is bound on TCP port 8082
NOTE: SNMP Adaptor is bound on UDP port 8087
The master agent forward traps on port : 8088

>> Press Enter if you want to stop.
```

3. **Start the `SimpleManager` application.**

   You need to provide `SimpleManager` with the following information:

   - The location of its security configuration file, `jdmk.security`
   - The master agent's host, in this case the local host
   - The port on which the master agent is listening for request, in this case 8087

   In *examplesDir*/`current/Snmp/MasterAgent/manager`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   SimpleManager localhost 8087
   >> Press Enter if you want to send a SNMPv3 request.
   ```

4. **Press `Enter` to send an SNMPv3 request**

   The following output is displayed:

   ```
   SimpleManager::main: Send SNMPv3 get request to SNMPv3 agent
   on localhost at port 8087
   Result:
   [Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
   Value : error1, Object ID : 1.3.6.1.2.1.1.2.0  (Syntax : String)
   Value : error2, Object ID : 1.3.6.1.2.1.1.3.0  (Syntax : String)
   Value : sysDescr3, Object ID : 1.3.6.1.2.1.1.4.0  (Syntax : String)
   Value : sysDescr4]

   >> Press Enter if you want to send a SNMPv2 request.
   ```

5. **Press `Enter` to send an SNMPv2 request**

   The following output is displayed:

   ```
   SimpleManager::main: Send SNMPv2 get request to SNMP agent
   on localhost at port 8087
   Result:
   [Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
   Value : error1, Object ID : 1.3.6.1.2.1.1.2.0  (Syntax : String)
   Value : error2, Object ID : 1.3.6.1.2.1.1.3.0  (Syntax : String)
   ```

```
Value : sysDescr3, Object ID : 1.3.6.1.2.1.1.4.0  (Syntax : String)
Value : sysDescr4]
$
```

The manager has successfully sent the two requests to the subagent, via the master agent.

▼ To Send Requests From a Manager to the SNMPv3 Master Agent

Because there is no MIB overlapping in this second example, all the variables implemented by the subagent MIB can be seen from the manager. Thus, you see that some of the OIDs queried return a `DisplayString` value of `error#`.

1. **Start the `StandAloneAgent` subagent.**

   In *examplesDir*`/current/Snmp/MasterAgent/standalone`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   StandAloneAgent 8085
   ```

   This binds `StandAloneAgent` to port 8085. Do not send traps when prompted.

2. **Start the SNMPv3 `MasterAgentV3` application.**

   You need to provide `MasterAgentV3` with the following information:

   - The location of its security configuration file, `jdmk.security`
   - The port on which it should listen for incoming requests, in this case we choose 8087
   - The subagent's host, in this case the local host
   - The subagent's port number, in this case 8085
   - The manager's host, in this case the local host

   In *examplesDir*`/current/Snmp/MasterAgent/master`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   MasterAgentV3 8087 localhost 8085 localhost
   ```

   The following output is displayed:

   ```
   NOTE: HTML adaptor is bound on TCP port 8082
   NOTE: SNMP Adaptor is bound on UDP port 8087

   >> Press Enter if you want to stop.
   ```

3. **Start the `SimpleManager` application.**

   You need to provide `SimpleManager` with the following information:

   - The location of its security configuration file, `jdmk.security`

- The master agent's host, in this case the local host
- The port on which the master agent is listening for requests, in this case 8087

In *examplesDir*`/current/Snmp/MasterAgent/manager`, type the following command:

```
$ java -classpath classpath -Djdmk.security.file=jdmk.security
SimpleManager localhost 8087
>> Press Enter if you want to send a SNMPv3 request.
```

4. **Press `Enter` to send an SNMPv3 request**

   The following output is displayed:

```
SimpleManager::main: Send SNMPv3 get request to SNMPv3 agent
on localhost at port 8087
Result:
[Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
Value : error1, Object ID : 1.3.6.1.2.1.1.2.0  (Syntax : String)
Value : error2, Object ID : 1.3.6.1.2.1.1.3.0  (Syntax : String)
Value : sysDescr3, Object ID : 1.3.6.1.2.1.1.4.0  (Syntax : String)
Value : sysDescr4]

>> Press Enter if you want to send a SNMPv2 request.
```

5. **Press `Enter` to send an SNMPv2 request**

   The following output is displayed:

```
SimpleManager::main: Send SNMPv2 get request to SNMP agent
on localhost at port 8087
Result:
[Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
Value : error1, Object ID : 1.3.6.1.2.1.1.2.0  (Syntax : String)
Value : error2, Object ID : 1.3.6.1.2.1.1.3.0  (Syntax : String)
Value : sysDescr3, Object ID : 1.3.6.1.2.1.1.4.0  (Syntax : String)
Value : sysDescr4]
$
```

The manager has successfully sent the two requests to the subagent, via the SNMPv3 master agent.

▼ To Send Requests From a Manager to a Master Agent With Overlapping MIBs

This example is very similar to "To Send Requests From a Manager to the SNMPv2 Master Agent" on page 407, except that the master agent now implements a local MIB which partly shadows the MIB implemented remotely by the agent.

Because the local MIB now shadows the variables for which the value was `error#`, these pseudo error messages no longer appear in the result displayed.

1. **Start the `StandAloneAgent` subagent.**

In *examplesDir*/current/Snmp/MasterAgent/standalone, type the following command:

```
$ java -classpath classpath -Djdmk.security.file=jdmk.security
StandAloneAgent 8085
```

This binds StandAloneAgent to port 8085. Do not send traps when prompted.

2. **Start the overlap MasterAgent application.**

   You need to provide MasterAgent with the following information:

   - The location of its security configuration file, jdmk.security
   - The port on which it should listen for incoming requests, in this case we choose 8087
   - The subagent's host, in this case the local host
   - The subagent's port number, in this case 8085

   In *examplesDir*/current/Snmp/MasterAgent/overlap, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   MasterAgent 8087 localhost 8085
   ```

   The following output is displayed:

   ```
   NOTE: HTML adaptor is bound on TCP port 8082
   NOTE: SNMP Adaptor is bound on UDP port 8087

   >> Press Enter if you want to stop.
   ```

3. **Start the SimpleManager application.**

   You need to provide SimpleManager with the following information:

   - The location of its security configuration file, jdmk.security
   - The master agent's host, in this case the local host
   - The port on which the master agent is listening for requests, in this case 8087

   In *examplesDir*/current/Snmp/MasterAgent/manager, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   SimpleManager localhost 8087
   >> Press Enter if you want to send a SNMPv3 request.
   ```

4. **Press Enter to send an SNMPv3 request**

   The following output is displayed:

   ```
   SimpleManager::main: Send SNMPv3 get request to SNMPv3
   agent on localhost at port 8087
   Result:
   [Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
   Value : sysDescr1, Object ID : 1.3.6.1.2.1.1.2.0  (Syntax : String)
   Value : sysDescr2, Object ID : 1.3.6.1.2.1.1.3.0  (Syntax : String)
   Value : sysDescr3, Object ID : 1.3.6.1.2.1.1.4.0  (Syntax : String)
   ```

```
Value : sysDescr4]

>> Press Enter if you want to send a SNMPv2 request.
```

5. **Press `Enter` to send an SNMPv2 request**

   The following output is displayed:

   ```
   SimpleManager::main: Send SNMPv2 get request to SNMP agent
   on localhost at port 8087
   Result:
   [Object ID : 1.3.6.1.2.1.1.1.0  (Syntax : String)
   Value : sysDescr1, Object ID : 1.3.6.1.2.1.1.2.0  (Syntax : String)
   Value : sysDescr2, Object ID : 1.3.6.1.2.1.1.3.0  (Syntax : String)
   Value : sysDescr3, Object ID : 1.3.6.1.2.1.1.4.0  (Syntax : String)
   Value : sysDescr4]
   $
   ```

   The manager has successfully sent the two requests to the subagent, via the overlap master agent. As you can see, the `error#` values have been replaced by the variables `sysDescr1` and `sysDescr2`, which were configured in the master agent to overlap with the variables in the subagent's MIB.

▼ To Receive Forwarded SNMPv1/v2 Traps in the Manager

In this scenario, an SNMPv1 trap or a v2 trap is sent by the subagent. The master agent forwards the trap as is and sends a translated trap. The version translation from v1 to v2, or from v2 to v1 is performed by the master agent. The manager therefore receives two traps, namely, the initial trap and the translated trap.

1. **Start the `StandAloneAgent` subagent.**

   In *examplesDir*`/current/Snmp/MasterAgent/standalone`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   StandAloneAgent 8085
   ```

   The following output is displayed:

   ```
   NOTE: SNMP Adaptor is bound on UDP port 8085
   Press Return if you want to send trap
   ```

2. **Press `Return` to activate stand alone traps**

   The following output is displayed:

   ```
   Press crtl-c in order to kill the agent

   Type 1 in order to launch snmp V1 traps, 2 for snmp V2.
   ```

   Do not send any traps yet, but leave the `StandAloneAgent` active.

3. **Start the SNMPv2 `MasterAgent` application.**

You need to provide `MasterAgent` with the following information:

- The location of its security configuration file, `jdmk.security`
- The port on which it should listen for incoming requests, in this case we choose 8087
- The subagent's host, in this case the local host
- The subagent's port number, in this case 8085

In *examplesDir*`/current/Snmp/MasterAgent/master`, type the following command:

```
$ java -classpath classpath -Djdmk.security.file=jdmk.security
MasterAgent 8087 localhost 8085
```

The following output is displayed:

```
NOTE: HTML adaptor is bound on TCP port 8082
NOTE: SNMP Adaptor is bound on UDP port 8087
The master agent forward traps on port : 8088

>> Press Enter if you want to stop.
```

4. **Start the `SimpleManager` application.**

   You need to provide `SimpleManager` with the following information:

   - The location of its security configuration file, `jdmk.security`
   - The master agent's host, which in this example must be the local host
   - The port on which the master agent is listening for requests, in this case 8087

   In *examplesDir*`/current/Snmp/MasterAgent/manager`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   SimpleManager localhost 8087
   >> Press Enter if you want to send a SNMPv3 request.
   ```

   Do not send a request this time.

5. **Go back to *examplesDir*`/current/Snmp/MasterAgent/standalone`**

   The `StandAloneAgent` should still be waiting to send traps.

6. **Type either `1` or `2` to send SNMPv1 or SNMPv2 traps**

   The following output is displayed:

   ```
   1
   V1 TRAP to send

   Trap V1 sent!

   Type 1 in order to launch snmp V1 traps, 2 for snmp V2.
   2
   V2 TRAP to send

   Trap V2 sent!
   ```

```
Type 1 in order to launch snmp V1 traps, 2 for snmp V2.
```

7. **Go back to *examplesDir*`/current/Snmp/MasterAgent/manager`**

   Notice that the `SimpleManager` application has received SNMPv1 and v2 traps from the `StandAloneAgent`. The following output from `SimpleManager` is displayed:

   ```
   NOTE: TrapListenerImpl received SNMPv1 trap :
           Generic 0
           Specific 0
           TimeStamp 104549
           Agent adress 129.157.203.98
   NOTE: TrapListenerImpl received SNMPv2 trap:
           CommunityString :
           VarBind list :
   oid : 1.3.6.1.2.1.1.3.0 val : 0:17:25
   oid : 1.3.6.1.6.3.1.1.4.1.0 val : 1.3.6.1.6.3.1.1.5.1
   oid : 1.2.3.4.5.6.0.0 val : Test values
   oid : 1.2.3.4.5.6.1.0 val : NULL
   oid : 1.2.3.4.5.6.2.0 val : 43
   oid : 1.2.3.4.5.6.3.0 val : Test values
   oid : 1.3.6.1.6.3.18.1.3.0 val : 129.157.203.98
   oid : 1.3.6.1.6.3.18.1.3.0 val : 129.157.203.98
   oid : 1.3.6.1.6.3.18.1.4.0 val :
   oid : 1.3.6.1.6.3.1.1.4.3.0 val : 1.3.6.1.4.1.42
   ```

8. **Stop the `StandAloneAgent` and the `SimpleManager` by pressing `Control-C`, and stop the `MasterAgent` by pressing `Enter`.**

▼ To Receive Forwarded SNMPv3 Traps in the Manager

In this scenario an SNMPv1 or SNMPv2 trap is sent by the subagent. The master agent translates the trap into an SNMP v3 trap and sends it to the manager.

1. **Start the `StandAloneAgent` subagent.**

   In *examplesDir*`/current/Snmp/MasterAgent/standalone`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   StandAloneAgent 8085
   ```

   The following output is displayed:

   ```
   NOTE: SNMP Adaptor is bound on UDP port 8085
   Press Return if you want to send trap
   ```

2. **Press `Return` to activate stand alone traps**

   The following output is displayed:

```
Press crtl-c in order to kill the agent

Type 1 in order to launch snmp V1 traps, 2 for snmp V2.
```

Do not send any traps yet, but leave the `StandAloneAgent` active.

3. **Start the SNMPv3 `MasterAgentV3` application.**

   You need to provide `MasterAgentV3` with the following information:

   - The location of its security configuration file, `jdmk.security`
   - The port on which it should listen for incoming requests, in this case we choose 8087
   - The subagent's host, in this case the local host
   - The subagent's port number, in this case 8085
   - The manager's host, in this case the local host

   In *examplesDir*`/current/Snmp/MasterAgent/master`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   MasterAgentV3 8087 localhost 8085 localhost
   ```

   The following output is displayed:

   ```
   NOTE: HTML adaptor is bound on TCP port 8082
   NOTE: SNMP Adaptor is bound on UDP port 8087
   The master agent forward traps on port : 8088

   >> Press Enter if you want to stop.
   ```

4. **Start the `SimpleManager` application.**

   You need to provide `SimpleManager` with the following information:

   - The location of its security configuration file, `jdmk.security`
   - The master agent's host, in this case the local host
   - The port on which the master agent is listening for requests, in this case 8087

   In *examplesDir*`/current/Snmp/MasterAgent/manager`, type the following command:

   ```
   $ java -classpath classpath -Djdmk.security.file=jdmk.security
   SimpleManager localhost 8087
   >> Press Enter if you want to send a SNMPv3 request.
   ```

   Do not send a request this time.

5. **Go back to *examplesDir*`/current/Snmp/MasterAgent/standalone`**

   The `StandAloneAgent` should still be waiting to send traps.

6. **Type either `1` or `2` to send SNMPv1 or SNMPv2 traps**

   The following output is displayed:

```
1
V1 TRAP to send

Trap V1 sent!

Type 1 in order to launch snmp V1 traps, 2 for snmp V2.
2
V2 TRAP to send

Trap V2 sent!

Type 1 in order to launch snmp V1 traps, 2 for snmp V2.
```

7. **Go back to *examplesDir*`/current/Snmp/MasterAgent/manager`**

   Notice that the `SimpleManager` application has received SNMPv3 traps from the `StandAloneAgent`, via `MasterAgentV3`. The following output from `SimpleManager` is displayed:

   ```
   NOTE: TrapListenerImpl received trap V3:
           ContextEngineId : 0x8000002a05819dcb6e00001f95
           ContextName : TEST-CONTEXT
           VarBind list :
   oid : 1.3.6.1.2.1.1.3.0 val : 0:5:24
   oid : 1.3.6.1.6.3.1.1.4.1.0 val : 1.3.6.1.6.3.1.1.5.1
   oid : 1.2.3.4.5.6.0.0 val : Test values
   oid : 1.2.3.4.5.6.1.0 val : NULL
   oid : 1.2.3.4.5.6.2.0 val : 43
   oid : 1.2.3.4.5.6.3.0 val : Test values
   oid : 1.3.6.1.6.3.18.1.3.0 val : 129.157.203.98
   oid : 1.3.6.1.6.3.18.1.3.0 val : 129.157.203.98
   oid : 1.3.6.1.6.3.18.1.4.0 val :
   oid : 1.3.6.1.6.3.1.1.4.3.0 val : 1.3.6.1.4.1.42
   ```

8. **Stop the `StandAloneAgent` and the `SimpleManager` by pressing `Control-C`, and stop the `MasterAgentV3` by pressing `Enter`**

PART VI

Legacy Features

This part describes features of the Java Dynamic Management Kit (Java DMK) that have been superseded by newer features, but that have been retained for purposes of backwards compatibility. Most of these features have been superseded by the implementation of the Java Management Extensions (JMX) Remote API in Java DMK 5.1. The old remote management applications described in this chapter are still fully functional, however, as are the old implementations of cascading and heartbeat.

In Part III, we saw how to manage agents programmatically using the new connectors defined by JMX Remote API. In this part, we present the older protocol connectors and proxy MBeans you can use for managing agents programmatically.

Java DMK defines the APIs needed to develop distributed management applications in the Java programming language. As of release 5.1 of Java DMK, the connectors defined by the JMX Remote API are the standard means for remote applications to establish connections with agents. However, in previous versions of Java DMK, connections were established through protocol connectors over remote method invocation (RMI), hypertext transfer protocol (HTTP), or secure HTTP (HTTPS). The principle, however, remains the same: a connector client object exposes a remote version of the MBean server interface, the connector server object in the agent transmits management requests to the MBean server and forwards any replies.

The legacy connectors and proxies described here provide an abstraction of the communication layer between agent and manager, but they also provide other mechanisms. Notification forwarding with configurable caching and pull policies lets you dynamically optimize bandwidth usage. The connector heartbeat monitors a connection, applies a retry policy, and automatically frees the resources when there is a communication failure. Finally, the connector server in an agent can act as a watchdog and refuse unauthorized requests based on the type of operation.

The implementation of JMX Remote API in Java DMK has also superseded the discovery and cascading services provided with previous versions of Java DMK. The legacy cascading and discovery services are also described in this part.

This part contains the following chapters:

- Chapter 21 shows how the legacy *protocol connectors* establish a connection between a management application written in the Java programming language and a Java dynamic management agent. Agents are identified by an address and port number; all other details of the communication are hidden. All connectors also implement the heartbeat mechanism to monitor the connection.
- Chapter 22 shows how *notification forwarding* extends the concept of notification listeners to the manager side. This chapter covers how manager-side listeners can register with agent-side broadcasters. The example then shows how the connector client and server interact to provide both pull and push modes for forwarding notifications from the agent to the manager.
- Chapter 23 presents the *security features* that can be enabled for a given legacy connection. The HTTP-based legacy connectors include a password mechanism that refuses unauthorized connections. Context checking and the general filtering mechanism can implement a complex algorithm for allowing or disallowing management requests to an agent. Finally, the HTTPS connector encrypts and protects data as it transits over the network between connector client and server components.
- Chapter 24 describes how legacy *MBean proxies* represent MBeans so that they are easier to access. A proxy object exposes the interface of its MBean for direct invocation. It then encodes the management request which it forwards to the MBean server and returns any response to its caller. Specific proxies can be generated for standard MBeans, but dynamic MBeans can only be accessed through generic proxies, similar to their `DynamicMBean` interface.
- Chapter 25 shows how *cascading agents* allow a manager to access a hierarchy of agents through a single point-of-access. The subagents can be spread across the network, but their resources can be controlled by the manager through a connection to a single master agent. MBeans of a subagent are mirrored in the master agent and respond as expected to all management operations, including their removal. No special proxy classes are needed for the mirror MBean, meaning that every MBean object can be mirrored anywhere without requiring any class loading.

CHAPTER 21

Legacy Protocol Connectors

Protocol connectors provide a point-to-point connection between a Java dynamic management agent and a management application. Each of the connectors relies on a specific communication protocol, but the API that is available to the management application is identical for all connectors and is entirely protocol-independent.

A connector consists of a connector server component registered in the agent and a connector client object instance in the management application. The connector client exposes a remote version of the MBean server interface. Each connector client represents one agent to which the manager wants to connect. The connector server replies to requests from any number of connections and fulfills them through its MBean server. Once the connection is established, the remoteness of the agent is transparent to the management application, except for any communication delays.

From Java Dynamic Management Kit (Java DMK) 5.1 onwards, remote connections are established via the remote method invocation (RMI) and JMX messaging protocol (JMXMP) connectors defined by Java Management Extensions (JMX) Remote API. However, for reasons of backwards compatibility, the connectors used in previous versions of Java DMK are retained as *legacy* features of the product.

All the connectors rely on the Java serialization package to transmit data as Java objects between client and server components. Therefore, all objects needed in the exchange of management requests and responses must be instances of a serializable class. However, the data encoding and sequencing are proprietary, and the raw data of the message contents in the underlying protocol are not exposed by the connectors.

All the legacy connectors provided with Java DMK implement a heartbeat mechanism to detect automatically when a connection is lost. This enables the manager to be notified when the communication is interrupted and when it is restored. If the connection is not reestablished, the connector automatically frees the resources that were associated with the lost connection.

The code samples in this chapter are taken from the files in the `legacy/SimpleClients` and `legacy/HeartBeat` directories located in the main *examplesDir* (see Directories and Classpath in the Preface).

This chapter covers the following topics:

- "21.1 Legacy Connector Servers" on page 420 presents the agent-side component of a legacy protocol connector.
- "21.2 Legacy Connector Clients" on page 423 presents the manager-side component and how it establishes a connection.
- "21.3 Legacy Heartbeat Mechanism" on page 431 explains how a legacy connector monitors a connection and demonstrates how to run the heartbeat example.

21.1 Legacy Connector Servers

Java DMK 5.0 introduced the concept of *multihome interfaces*. This enables you to work in an environment in which multiple network protocols can be used at different times by the legacy connector servers.

The Java DMK legacy connector servers on the server side do not need to have specified local interfaces. By default, the connector server listens on all local interfaces.

There is a legacy connector server for each of the protocols supported in the previous versions of Java DMK: RMI, hypertext transfer protocol (HTTP) and secure HTTP (HTTPS). Support for these protocols is retained in Java DMK 5.1 for reasons of backwards compatibility. All the legacy connectors inherit from the `CommunicatorServer` class that is the superclass for all the legacy protocol adaptors and connector servers. This class defines the methods needed to control the port and communication settings that are common to all. Each connector server class then provides specific controls, such as the service name for RMI and authentication information for HTTP. This example covers the legacy RMI connector server, and the HTTP authentication mechanism is described in Chapter 23.

A connector server listens for incoming requests from its corresponding connector client, decodes that request and encodes the reply. Several connector clients can establish connections with the same connector server, and the connector server can handle multiple requests simultaneously. There only needs to be one connector server MBean per protocol to which the agent needs to respond. However, several connector servers for the same protocol can coexist in an agent for processing requests on different ports.

21.1.1 Instantiating a Legacy RMI Connector Server

The legacy RMI connector server is an MBean, so we instantiate its class and register it in the MBean server. This operation could also be performed remotely, for example if a management application wants to access an agent through an alternative protocol.

The code extract shown in Example 21–1 is taken from the BaseAgent class in the *examplesDir*/current/BaseAgent directory.

EXAMPLE 21–1 Instantiating the Legacy RMI Connector Server

```
// Instantiate an RMI connector server with default port
//
CommunicatorServer rmiConnector = new RmiConnectorServer();

try {
    // We let the RMI connector server provide its default name
    ObjectName rmiConnectorName = null;
    ObjectInstance rmiConnectorInstance =
        myMBeanServer.registerMBean( rmiConnector, rmiConnectorName );

} catch(Exception e) {
    e.printStackTrace();
}
```

Other constructors for the legacy RmiConnectorServer class have parameters for specifying the port and the RMI service name that the connector server will use. The default constructor assigns port 1099 and name=RmiConnectorServer, as given by the static variables RMI_CONNECTOR_PORT and RMI_CONNECTOR_SERVER, respectively, of the ServiceName class. Both attributes can also be accessed through the getter and setter methods of the legacy RmiConnectorServer class.

Each legacy connector uses different parameters that are specific to its protocol. For example, the HTTP connector does not need a service name. The default values for all parameters are given by the static variables of the ServiceName class.

Registering a connector server as an MBean implies that its MBean server will handle the remote requests that it receives. However, you can specify a different object for fulfilling management requests through the setMBeanServer method that the RmiConnectorServer class inherits from the CommunicatorServer class. For security reasons, this method is not exposed in the legacy RMI connector server MBean, so it must be called from the agent application.

Registering the connector server as an MBean is optional. For example, you might not want it exposed for management. In this case, you must use the setMBeanServer method to specify an object that implements the MBeanServer interface so that it can fulfill management requests.

A user can select a local server interface for a legacy RMI connector server using the RMI property java.rmi.server.hostname.

21.1.2 Connector States

Like all communicator servers, the legacy RMI connector server has a connection state that is identified by the static variables of the CommunicatorServer class:

- OFFLINE – Stopped and not responding
- STARTING – In a transitional state and not yet responding
- ONLINE – Able to respond to management requests
- STOPPING – In a transitional state and no longer responding

All connector servers are OFFLINE after their instantiation, so they must be started explicitly. Then, you must wait for a connector server to come ONLINE before it can respond to connections on its designated port.

EXAMPLE 21–2 Starting the RMI Connector Server

```
// Explicitly start the RMI connector server
//
rmiConnector.start();

// waiting for it to leave starting state...
while (rmiConnector.getState() == CommunicatorServer.STARTING) {
    try {
        Thread.sleep( 1000 );
    } catch (InterruptedException e) {
        continue;
    }
}
```

Instead of blocking the application thread, you can register a listener for attribute change notifications concerning the State attribute of the connector server MBean. All connector servers implement this notification which contains both old and new values of the attribute (see “8.3 Attribute Change Notifications” on page 113). Listeners in the agent can then asynchronously detect when the state changes from STARTING to ONLINE.

Note – During the STARTING state, the legacy RMI connector server registers its RMI service name with the RMI registry on the local host for its designated port. If no registry exists, one is instantiated for the given port. Due to a limitation of version 1.4 of the Java platform, creating a second registry in the same Java virtual machine (JVM) will fail. As a workaround, before starting an RMI connector server on a new, distinct port number in an agent, you must run the rmiregistry command from a terminal on the same host. This limitation is specific to the legacy RMI connector. The HTTP protocols do not require a registry.

The stop method is used to take a connector server offline. The stop method is also called by the preDeregister method that all connector servers inherit. Stopping a connector server interrupts all requests that are pending and closes all connections that are active. When a connection is closed, all of its resources are cleaned up, including all notification listeners, and the connector client can be notified by a heartbeat notification (see “21.3 Legacy Heartbeat Mechanism” on page 431). A connection that is closed can no longer be reopened. The connector client must establish a new connection when the connector server is restarted.

The setPort method that all connector servers inherit from the CommunicatorServer class enables you to change the port on which management requests are expected. You can only change the port when the connector server is offline, so it must be explicitly stopped and then restarted. The same rule applies to the setServiceName method that is specific to the RMI connector server. These methods are also exposed in the MBean interface, along with start and stop, enabling a remote management application to configure the connector server through a different connection.

21.2 Legacy Connector Clients

The manager application interacts with a connector client to access an agent through an established connection. Through its implementation of the RemoteMBeanServer interface, a legacy connector client provides methods to handle the connection and access the agent. This interface specifies nearly all of the same methods as the MBeanServer interface, meaning that an agent is fully manageable from a remote application.

Through the connector, the management application sends management requests to the MBeans located in a remote agent. Components of the management application access remote MBeans by calling the methods of the connector client for getting and setting attributes and calling operations on the MBeans. The connector client then returns the result, providing a complete abstraction of the communication layer.

By default, a legacy connector client uses the default host name, InetAddress.getLocalHost().getHostName()

21.2.1 Multihome Interfaces

The multihome interfaces introduced in Java DMK 5.0 enable you to select the interfaces for the different network protocols used by legacy connector clients.

An HTTP(S) client enables you to select a local interface to receive notifications from a server. This user-specific local interface address is sent to a server and is then used by that server to push notifications. The interface address can be specified as either an IPv4 or an IPv6 address, or as a host name.

A legacy RMI connector client does not need to specify a local interface.

21.2.2 RemoteMBeanServer Interface

All connector clients implement the RemoteMBeanServer interface to expose the methods needed to access and manage the MBeans in a remote agent. This interface allows all management operations that would be possible directly in the agent application. In fact, the methods of the connector client for accessing MBeans remotely have exactly the same name and signature as their equivalent methods in the MBeanServer interface.

Table 21–1 lists the methods that are defined identically in both the MBeanServer and the RemoteMBeanServer interfaces.

TABLE 21–1 Shared Methods

Type	Signature
* void	**addNotificationListener**(ObjectName name, NotificationListener listener, NotificationFilter filter, java.lang.Object handback)
ObjectInstance	**createMBean**(*) – All four overloaded forms of this method
* java.lang.Object	**getAttribute**(ObjectName name, java.lang.String attribute)
* AttributeList	**getAttributes**(ObjectName name, java.lang.String[] attributes)
java.lang.String	**getDefaultDomain**()
java.lang.Integer	**getMBeanCount**()
* MBeanInfo	**getMBeanInfo**(ObjectName name)
ObjectInstance	**getObjectInstance**(ObjectName name)
* java.lang.Object	**invoke**(ObjectName name, java.lang.String operationName, java.lang.Object[] params, java.lang.String[] signature)
boolean	**isInstanceOf**(ObjectName name, java.lang.String className)
boolean	**isRegistered**(ObjectName name)
java.util.Set	**queryMBeans**(ObjectName name, QueryExp query)
java.util.Set	**queryNames**(ObjectName name, QueryExp query)

TABLE 21–1 Shared Methods *(Continued)*

Type	Signature
* `void`	**`removeNotificationListener`**`(ObjectName name, NotificationListener listener)`
* `void`	**`setAttribute`**`(ObjectName name, Attribute attribute)`
* `AttributeList`	**`setAttributes`**`(ObjectName name, AttributeList attributes)`
* `void`	**`unregisterMBean`**`(ObjectName name)`

These methods are defined in the `ProxyHandler` interface (see "24.1.1 Legacy Local and Remote Proxies" on page 471).

Components of a management application that rely solely on this common subset of methods can be instantiated in either the agent or the manager application. Such components are location-independent and can be reused either locally or remotely as management solutions evolve. This symmetry also allows the design of advanced management architectures where functionality can be deployed either in the agent or in the manager, depending on runtime conditions.

The other, unshared methods of the `RemoteMBeanServer` interface are used to establish and monitor the connection. "21.2.3 Establishing a Connection" on page 425 shows how to establish a connection and access MBeans directly through the connector client."21.3 Legacy Heartbeat Mechanism" on page 431 shows how to monitor a connection and detect when it is lost.

21.2.3 Establishing a Connection

The target of a connection is identified by a protocol-specific implementation of the `ConnectorAddress` interface. This object contains all the information that a connector client needs to establish a connection with the target agent. All address objects specify a host name and port number. An RMI address adds the required service name, and HTTP-based addresses have an optional authentication field (see "23.1 Password-Based Authentication (Legacy Connectors)" on page 456). In addition, the `ConnectorType` string identifies the protocol without needing to introspect the address object.

In our example, the target of the connection is an active legacy RMI connector server identified by an `RmiConnectorAddress` object. We use the default constructor to instantiate a default address object, but otherwise, these parameters can be specified in a constructor or through setter methods. The default values of the information contained in the `RmiConnectorAddress` object are the following:

- The ConnectorType identifies the protocol that is used; its value is SUN RMI for the RmiConnectorAddress class.
- The default RMI port is 1099, as given by the static variable RMI_CONNECTOR_PORT in the ServiceName class.
- The Host is the name of the host where the target agent is running; by default, its value is the local host.
- The Name attribute specifies the RMI registry service name of the adaptor server. Its default value is name=RmiConnectorServer, which is the value of the RMI_CONNECTOR_SERVER static variable in the ServiceName class.

The RmiConnectorAddress object is used as the parameter to the connect method of the RmiConnectorClient instance. This method tries to establish the connection and throws an exception if there is a communication or addressing error. Otherwise, when the connect method returns, the connector client is ready to perform management operations on the designated agent.

The code segment shown inExample 21–3 is taken from the ClientWithoutProxy class in the examples directory.

EXAMPLE 21–3 Establishing a Connection

```
echo("\t>> Instantiate the RMI connector client...");
connectorClient = new RmiConnectorClient();

echo("\t>> Instantiate a default RmiConnectorAddress object...");
RmiConnectorAddress address = new RmiConnectorAddress();

// display the default values
echo("\t\tTYPE\t= "   + address.getConnectorType());
echo("\t\tPORT\t= "   + address.getPort());
echo("\t\tHOST\t= "   + address.getHost());
echo("\t\tSERVER\t= " + address.getName());
echo("\t<< done <<");

echo("\t>> Connect the RMI connector client to the agent...");
try {
     connectorClient.connect( address );

} catch(Exception e) {
     echo("\t!!! RMI connector client could not connect to the agent !!!");
     e.printStackTrace();
     System.exit(1);
}
```

21.2.4 Managing MBeans Remotely

Once the connection to an agent is established, the management application can access that agent's MBeans through the legacy RemoteMBeanServer interface of the connector client. Invoking these methods has exactly the same effect as calling the equivalent methods directly on the MBean server instance.

It is possible to restrict access to certain methods of the MBean server when they are called through the legacy connector client, but this is performed by a security mechanism in the legacy connector server. See "23.2 Context Checking" on page 459 for more details.

21.2.4.1 Creating and Unregistering MBeans in the Agent

We use the createMBean method to instantiate and register an object from its class name. This class must already be available in the agent application's classpath.

EXAMPLE 21–4 Creating and Unregistering an MBean Remotely

```
private void doWithoutProxyExample(String mbeanName) {

    try {

        // build the MBean ObjectName instance
        //
        ObjectName mbeanObjectName = null;
        String domain = connectorClient.getDefaultDomain();
        mbeanObjectName = new ObjectName( domain + ":type=" + mbeanName );

        // create and register an MBean in the MBeanServer of the agent
        //
        echo("\nCurrent MBean count in the agent = "+
             connectorClient.getMBeanCount());
        echo("\n>>> CREATE the " + mbeanName +
             " MBean in the MBeanServer of the agent:");
        String mbeanClassName = mbeanName;

        ObjectInstance mbeanObjectInstance =
        connectorClient.createMBean( mbeanClassName, mbeanObjectName );

        echo("\tMBEAN CLASS NAME      = " +
             mbeanObjectInstance.getClassName() );
        echo("\tMBEAN OBJECT NAME     = " +
             mbeanObjectInstance.getObjectName() );
        echo("\nCurrent MBean count in the agent = "+
             connectorClient.getMBeanCount() );

        [...] // Retrieve MBeanInfo and access the MBean (see below)

        // unregister the MBean from the agent
        //
        echo("\n>>> UNREGISTERING the "+ mbeanName +" MBean");
        connectorClient.unregisterMBean(
            mbeanObjectInstance.getObjectName() );

        [...]

    } catch (Exception e) {
        e.printStackTrace();
```

EXAMPLE 21–4 Creating and Unregistering an MBean Remotely *(Continued)*

```
    }
}
```

Example 21–4 shows the use of other calls to the remote agent, such as `getDefaultDomain` and `getMBeanCount` that have the same purpose as in an agent application.

21.2.4.2 Accessing MBean Attributes and Operations

Once you can access the object names for MBeans in the agent, you can know their management interface from their `MBeanInfo` object. The code in Example 21–5 is actually called in between the MBean creation and unregistration shown in Example 21–4.

EXAMPLE 21–5 Retrieving the `MBeanInfo` Object

```
ObjectName mbeanObjectName = mbeanObjectInstance.getObjectName();

echo("\n>>> Getting the management information of the MBean");
MBeanInfo info = null;
try {

    info = connectorClient.getMBeanInfo( mbeanObjectName );

} catch (Exception e) {
    echo("\t!!! Could not get MBeanInfo object for "+ mbeanObjectName );
    e.printStackTrace();
    return;
}

// display content of MBeanInfo object
//
echo("\nCLASSNAME: \t"+ info.getClassName());
echo("\nDESCRIPTION: \t"+ info.getDescription());
echo("\nATTRIBUTES");
MBeanAttributeInfo[] attrInfo = info.getAttributes();
if ( attrInfo.length>0 ) {
    for( int i=0; i<attrInfo.length; i++ ) {
        echo(" ** NAME: \t"+ attrInfo[i].getName());
        echo("    DESCR: \t"+ attrInfo[i].getDescription());
        echo("    TYPE: \t"+ attrInfo[i].getType() +
             "\tREAD: "+ attrInfo[i].isReadable() +
             "\tWRITE: "+ attrInfo[i].isWritable());
    }
} else echo(" ** No attributes **");

[...]
```

It is then straightforward to perform management operations on MBeans through the legacy connector client. As in an agent, we call the generic getters, setters, and call methods with the object name of the MBean, the name of the attribute or operation, and any parameters. As in the methods of the MBean server, we need to use the `Attribute` and `AttributeList` classes to pass attributes as name-value pairs.

EXAMPLE 21–6 Accessing an MBean Through the Legacy Connector Client

```
try {
    // Getting attribute values
    String State = (String)
        connectorClient.getAttribute( mbeanObjectName, "State" );
    Integer NbChanges = (Integer)
        connectorClient.getAttribute( mbeanObjectName, "NbChanges" );
    echo("\tState     = \"" + State + "\"");
    echo("\tNbChanges = " + NbChanges);

    // Setting the "State" attribute
    Attribute stateAttr = new Attribute("State", "new state from client");
    connectorClient.setAttribute(mbeanObjectName, stateAttr);

    // Invoking the "reset" operation
    Object[] params = new Object[0];
    String[] signature = new String[0];
    connectorClient.invoke(mbeanObjectName, "reset", params, signature);

} catch (Exception e) {
    e.printStackTrace();
    return;
}
```

All other MBean access methods are available in the same manner, such as bulk getters and setters, and the query methods.

21.2.4.3 Creating and Accessing Dynamic MBeans

In the first run of the example, the management application creates, manages, and unregisters a standard MBean in the remote agent. However, standard and dynamic MBeans are designed to be managed through the same methods, both in the `MBeanServer` and in the `RemoteMBeanServer` interfaces.

As shown in Example 21–4, the subroutine of the example application takes only a single class name parameter. The first time this subroutine is called, the example passes the class name of a standard MBean, and the second time, that of a dynamic MBean. For the example to run, the two MBeans must have an identical management interface. By extension of this special case, we see that the connector client can manage dynamic MBeans through the same methods as it manages standard MBeans, without making any distinction between the two.

21.2.5 Running the `SimpleClients` Example

The *examplesDir*/`legacy/SimpleClients` directory contains all of the files for three sample managers, a base agent, and some MBeans to manage. In this chapter, we run the `ClientWithoutProxy` application that demonstrates simple operations on MBeans through a legacy RMI connector.

▼ To Run the Simple Client Example

1. **Compile all files in this directory with the `javac` command.**

   For example, on the Solaris platform with the Korn shell, type:

   ```
   $ cd examplesDir/legacy/SimpleClients/
   $ javac -classpath classpath *.java
   ```

   We do not need all the files for this chapter, but they are used in Chapter 24. In this demonstration, we only need the `BaseAgent` and `ClientWithoutProxy` applications, as well as the standard and dynamic MBeans.

2. **Start the agent in another terminal window on the *same* host with the following command:**

   ```
   $ java -classpath classpath BaseAgent
   ```

   The agent creates an HTML protocol adaptor and a legacy RMI connector server to which the client application establishes a connection, and then it waits for management operations.

3. **Wait for the agent to be completely initialized, then start the manager with the following command:**

   ```
   $ java -classpath classpath ClientWithoutProxy
   ```

   The client application creates the RMI connector client and establishes the connection to the base agent.

4. **Press `Enter` in the manager window to step through the example.**

   The management application instantiates both types of MBeans, looks at their metadata, and performs management operations on them. The results of each step are displayed in the terminal window.

5. **At any time, you can view the agent through its HTML adaptor and interact with the MBeans created by the management application.**

   For example, immediately after the manager creates an MBean, you could modify its attributes and see this change reflected when the connector client access the new values.

6. **Press `Enter` in both windows to stop the agent and manager applications.**

21.3 Legacy Heartbeat Mechanism

The legacy heartbeat mechanism monitors the connection between a manager and an agent and automates the cleanup procedure when the connection is lost. This enables both the manager and the agent to release resources that were allocated for maintaining the connection.

The mechanism is entirely contained in the connector client and connector server components, no additional objects are involved. In addition, connector clients send notifications that the manager application can receive to be aware of changes in the status of a connection.

All legacy connector clients of the Java DMK implement the `HeartBeatClientHandler` interface to provide a heartbeat mechanism. This means that agent-manager connections over the legacy RMI, HTTP, and HTTPS connectors can be monitored and controlled in the same way. A manager application could even use the same notification handler for all connector clients where the heartbeat mechanism is activated.

21.3.1 Configuring the Heartbeat

To monitor the connection, the connector client sends periodic heartbeats (*ping* requests) to the connector server that acknowledges them by sending a reply (*ping* responses). If either heartbeat is lost, the components of the legacy connector retry until either the connection is re-established or the number of retries has been exhausted.

In a connector client, the methods specified by the `HeartBeatClientHandler` interface set the heartbeat period and the number of retries that are attempted. You should determine these parameters empirically to implement the desired connection monitoring behavior, taking into account the network conditions and topology between the hosts of your manager and agent applications.

In Example 21–7, the management application configures the heartbeat mechanism before the connection to an agent is established.

EXAMPLE 21–7 Configuring the Heartbeat in the Connector Client

```
// CREATE a new RMI connector client
//
echo("\tInstantiate the RMI connector client...");
connectorClient = new RmiConnectorClient();

// SET heartbeat period to 1 sec. Default value is 10 secs
//
echo("\tSet heartbeat period to 1 second...");
```

EXAMPLE 21–7 Configuring the Heartbeat in the Connector Client *(Continued)*

```
connectorClient.setHeartBeatPeriod(1000);

// SET heartbeat number of retries to 2. Default value is 6 times
//
echo("\tSet heartbeat number of retries to 2 times...");
connectorClient.setHeartBeatRetries(2);
```

Using the same methods, the heartbeat configuration can also be modified at any time, even after the connection has been established. By default, the heartbeat mechanism is activated in a connector with a 10 second heartbeat and 6 retries, meaning that a connection that cannot be re-established within one minute is assumed to be lost.

Setting the number of heartbeat retries to zero causes a lost connection to be signalled immediately after the heartbeat fails. Setting the heartbeat period to zero deactivates the mechanism and prevents any further connection failures from being detected.

No specific configuration is necessary on the agent-side legacy connector server. It automatically responds to the heartbeat messages. These heartbeat messages contain the current heartbeat settings from the legacy connector client that also configure the connector server. In this way, both client and server apply the same retry policy, and when the configuration is updated in the connector client, it is immediately reflected in the connector server. The connector server can handle multiple connections from different management applications, each with its specific heartbeat configuration.

The legacy connector server applies its retry policy when the next expected heartbeat message is not received within the heartbeat period. From that moment, the connector server begins a timeout period that lasts 20% longer than the number of retries times the heartbeat period. This corresponds to the time during which the connector client attempts to resend the heartbeat, with a safety margin to allow for communication delays. If no further heartbeat is received in that timeout, the connection is determined to be lost.

The heartbeat ping messages also contain a connection identifier so that connections are not erroneously re-established. If a connector server is stopped, thereby closing all connections, and then restarted between two heartbeats or before the client's timeout period has elapsed, the server responds to heartbeats from a previous connection. However, the connector client detects that the identifier has changed and immediately declares that the connection is lost, regardless of the number of remaining retries.

During the timeout period, the notification push mechanism in the connector server is suspended to avoid losing notifications (see Chapter 22). Similarly, while the connector client is retrying the heartbeat, it must suspend the notification pull mechanism if it is in effect.

When a connection is determined to be lost, both the connector client and server free any resources that were allocated for maintaining the connection. For example, the connector server unregisters all local listeners and deletes the notification cache needed to forward notifications. Both components also return to a coherent, functional state, ready to establish or accept another connection.

The state of both components after a connection is lost is identical to the state that is reached after the `disconnect` method of the connector client is called. In fact, the `disconnect` method is called internally by the connector client when a connection is determined to be lost, and the equivalent, internal method is called in the connector server when its timeout elapses without recovering a heartbeat.

21.3.2 Receiving Heartbeat Notifications

The legacy connector client also sends notifications that signal any changes in the state of the connection. These notifications are instances of the `HeartBeatNotification` class. The `HeartBeatClientHandler` interface includes methods specifically for registering for heartbeat notifications. These methods are distinct from those of the `NotificationRegistration` interface that a connector client implements for transmitting agent-side notifications (see "22.1 Registering Manager-Side Listeners" on page 439).

EXAMPLE 21–8 Registering for Heartbeat Notifications

```
// Register this manager as a listener for heartbeat notifications
// (the filter and handback objects are not used in this example)
//
echo("\tAdd this manager as a listener for heartbeat notifications...");
connectorClient.addHeartBeatNotificationListener(this, null, null);
```

Instances of heartbeat notifications contain the connector address object from the connection that generated the event. This enables the notification handler to listen to any number of connectors and retrieve all relevant information about a specific connection when it triggers a notification. The `HeartBeatNotification` class defines constants to identify the possible notification type strings for heartbeat events:

- `CONNECTION_ESTABLISHED` – Emitted when the `connect` method succeeds
- `CONNECTION_RETRYING` – After the heartbeat fails and if the number of retries is not zero, this notification type is emitted *once* when the first retry is sent
- `CONNECTION_REESTABLISHED` – Emitted if the heartbeat recovers during one of the retries
- `CONNECTION_LOST` – Emitted after the heartbeat and all retries, if any, have failed or when a heartbeat contains the wrong connection identifier, indicating that the connector server has been stopped and restarted
- `CONNECTION_TERMINATED` – Emitted when the `disconnect` method successfully terminates a connection and frees all the resources it used; therefore, this notification type is received both after a user–terminated connection and after

a connection is lost

Once they are established, connections can go through any number of retrying/re-established cycles and then be terminated by the user or determined to be lost and terminated automatically. When the heartbeat mechanism is deactivated by setting the heartbeat period to zero, only heartbeat notifications for normally established and normally terminated connections continue to be sent. In that case, connections can be lost but they are not detected nor indicated by a notification.

The following diagram shows the possible sequence of heartbeat notifications during a connection. Retries are enabled when the `getHeartBeatRetries` method returns an integer greater than zero.

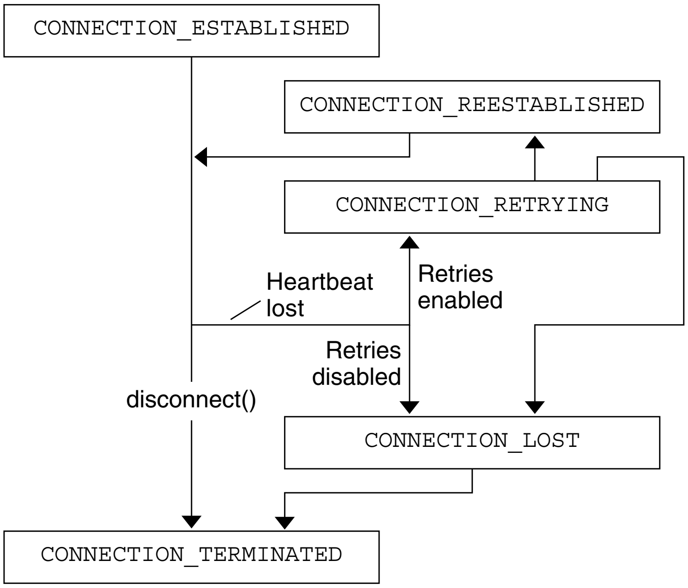


FIGURE 21–1 Sequencing of Heartbeat Notifications

Example 21–9 shows the source code for the notification handler method in our manager class. The handler prints out the notification type and the RMI address associated with the connector that generated the notification.

EXAMPLE 21–9 Heartbeat Notification Handler

```
public void handleNotification(Notification notification, Object handback){

    echo("\n>>> Notification has been received...");
    echo("\tNotification type = " + notification.getType());

    if (notification instanceof HeartBeatNotification) {
        ConnectorAddress notif_address =
            ((HeartBeatNotification)notification).getConnectorAddress();
```

EXAMPLE 21–9 Heartbeat Notification Handler *(Continued)*

```
        if (notif_address instanceof RmiConnectorAddress) {
            RmiConnectorAddress rmi_address =
                (RmiConnectorAddress) notif_address;

            echo("\tNotification connector address:");
            echo("\t\tTYPE   = " + rmi_address.getConnectorType());
            echo("\t\tHOST   = " + rmi_address.getHost());
            echo("\t\tPORT   = " + rmi_address.getPort());
            echo("\t\tSERVER = " + rmi_address.getName());
            }
    }
}
```

In the agent application, the connector server does not emit any notifications about the state of its connections. The HTTP protocol-based connectors do provide a count of active clients, but there is no direct access to heartbeat information in an agent's connector servers.

21.3.3 Running the Legacy `Heartbeat` Example

The *examplesDir*`/legacy/HeartBeat` directory contains all of the files for the `Agent` and `Client` applications that demonstrate the heartbeat mechanism through an RMI connector.

▼ To Run the `Heartbeat` Example: Normal Termination

1. **Compile all files in this directory with the `javac` command.**

   For example, on the Solaris platform with the Korn shell, type:

   ```
   $ cd examplesDir/legacy/HeartBeat/
   $ javac -classpath classpath *.java
   ```

   To demonstrate the various communication scenarios, we will run the example three times: once to see a normal termination, once to see how the manager reacts to a lost connection, and once to see how the agent reacts to a lost connection.

2. **Start the agent on another host or in another terminal window with the following command:**

   ```
   $ java -classpath classpath Agent
   ```

   The agent only creates the legacy RMI connector server to which the client application will establish a connection, and then it waits for management operations.

3. **Wait for the agent to be completely initialized, then start the manager with the following command, where *hostname* is the name of the host running the agent.**

The RMI connector in this example uses port 1099 by default. If you started the agent on the same host, you can omit the *hostname* and the port number:

```
$ java -classpath classpath Client hostname 1099
```

The client application creates the legacy RMI connector client, configures its heartbeat, and registers a notification listener, as seen in the code examples. When the connection is established, the listener outputs the notification information in the terminal window.

4. **Press `Enter` in the manager window to call the `disconnect` method on the connector client and stop the `Client` application.**

   In the terminal window, the heartbeat notification listener outputs the information for the normal connection termination before the application ends.

5. **Leave the agent application running for the next scenario.**

▼ To Run the Legacy `Heartbeat` Example: Connector Client Reaction

1. **Start the `Client` application again:**

   ```
   $ java -classpath classpath Client hostname 1099
   ```

2. **When the connection is established, press `Control-C` in the *agent's* window to stop the connector server and the agent application.**

   This simulates a broken communication channel as seen by the connector client.

   Less than a second later, when the next heartbeat fails, the heartbeat retrying notification is displayed in the manager's terminal window. Two seconds later, after both retries have failed, the lost connection and terminated connection notifications are displayed.

3. **Press `Enter` in the manager window to exit the `Client` application.**

▼ To Run the Legacy `Heartbeat` Example: Connector Server Reaction

1. **Start the agent in debug mode on another host or in another terminal window with the following command:**

   ```
   $ java -classpath classpath -DLEVEL_DEBUG Agent
   ```

2. **Wait for the agent to be completely initialized, then start the `Client` application again:**

   ```
   $ java -classpath classpath Client hostname 1099
   ```

   When the connection is established, the periodic heartbeat messages are displayed in the debug output of the agent.

3. **This time, press `Control-C` in the** *client's* **window to stop the connector client and the manager application.**

   This simulates a broken communication channel as seen by the connector server in the agent.

   After the heartbeat retry timeout elapses in the agent, the lost connection message is displayed in the heartbeat debugging output of the agent.

4. **Type `Control-C` in the agent window to stop the agent application and end the example.**

CHAPTER 22

Notification Forwarding in Legacy Connectors

The notification mechanism that was used with the legacy connectors in versions of the Java Dynamic Management Kit (Java DMK) prior to version 5.1 has been superseded by the much simpler notifications from Java Management Extensions (JMX) Remote API. The legacy notification mechanism has been retained for reasons of backwards compatibility.

The code samples in this topic are taken from the files in the `legacy/Notification` example directory located in the main *examplesDir* (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "22.1 Registering Manager-Side Listeners" on page 439 shows how similar the agent-side and manager-side legacy notification mechanisms are.
- "22.2 Push Mode" on page 444 presents the simplest forwarding policy whereby the agent sends notifications to the manager as they occur.
- "22.3 Pull Mode" on page 446 presents an advanced forwarding policy that buffers notifications in the agent until the manager requests them.
- "22.4 Running the Legacy Notification Forwarding Example" on page 451 demonstrates the two forwarding mechanisms and how to set the parameters for the two strategies.

22.1 Registering Manager-Side Listeners

Like the other structures of Java DMK, the legacy notification mechanism is designed to be homogeneous from the agent to the manager side. For this reason, notification objects and listener interfaces in manager applications are identical to those on the agent side.

The symmetry of the interfaces also means that code for listeners can easily be reused without modification in either agent or manager applications. Listeners in managers are similar to those in agents, and they could even be identical objects in some management solutions. However, in most cases, manager-side listeners will receive different notifications and take different actions from their agent-side peers.

22.1.1 Agent-Side Broadcaster

The notification broadcasters are MBeans registered in an agent's MBean server to which our management application needs to connect. Only notifications sent by registered MBeans can be forwarded to manager applications, and a manager-side listener can receive them only by registering through a connector client or a proxy object.

Other notification broadcasters can exist independently in the manager application, but listeners need to register directly with these local broadcasters. Nothing prevents a listener object from registering both with a connector client or proxy for remote notifications and with a local broadcaster.

The code example below shows how the sample `NotificationEmitter` MBean sends notifications (the code for its `MBean` interface has been omitted). It extends the `NotificationBroadcasterSupport` class to reuse all of its listener registration facilities. It only contains one operation that can be called by our manager to trigger any number of notifications.

EXAMPLE 22–1 Agent-Side Broadcaster MBean

```
import javax.management.MBeanNotificationInfo;
import javax.management.NotificationBroadcasterSupport;
import javax.management.Notification;

public class NotificationEmitter
    extends NotificationBroadcasterSupport
    implements NotificationEmitterMBean {

    // Just to make the inheritance explicit
    public NotificationEmitter() {
        super();
    }

    // Provide details about the notification type and class that is sent
    public MBeanNotificationInfo[] getNotificationInfo() {

        MBeanNotificationInfo[] ntfInfoArray = new MBeanNotificationInfo[1];

        String[] ntfTypes = new String[1];
        ntfTypes[0] = myType;

        ntfInfoArray[0] = new MBeanNotificationInfo( ntfTypes,
            "javax.management.Notification",
```

EXAMPLE 22–1 Agent-Side Broadcaster MBean *(Continued)*

```
            "Notifications sent by the NotificationEmitter");
        return ntfInfoArray;
    }

    // The only operation: sends any number of notifications
    // whose sequence numbers go from 1 to "nb"
    public void sendNotifications( Integer nb ) {

        for (int i=1; i<=nb.intValue(); i++) {
            sendNotification(new Notification(myType, this, i));
        }
    }
    private String myType = "notification.my_notification";
}
```

Our MBean invents a notification type string and exposes this information through the `getNotificationInfo` method. To demonstrate the forwarding mechanism, we are more interested in the sequence number, which enables us to identify the notifications as they are received in the manager.

This MBean demonstrates that the broadcaster has total control over the contents of its notifications. Constructors for the `Notification` object enable you to specify all of the fields, even ones such as the time stamp. In this example, we control the sequence number and our chosen policy is to reset the sequence number to 1 with every call to the operation. Of course, you are free to choose the notification contents, including the time-stamping and sequence-numbering policies that fit your management solution.

Note – Due to possible loss in the communication layer and the inherent indeterminism of thread execution, the legacy notification model does not guarantee that remote notifications will be received nor that their sequence will be preserved. If notification order is critical to your application, your broadcaster should set the sequence numbers appropriately, and your listeners should sort the received notifications.

22.1.2 Manager-Side Listener

In our simple example, the `Client` class itself is the listener object. Usually, a listener would be a separate instance of a special listener class and depending on the complexity of the manager, there might be several classes of listeners, each for a specialized category of notifications.

EXAMPLE 22–2 The Manger-Side Listener

```
public class Client implements NotificationListener {

    [...] // Constructor omitted
```

EXAMPLE 22–2 The Manger-Side Listener *(Continued)*

```
    // Implementation of the NotificationListener interface
    //
    public void handleNotification(Notification notif, Object handback) {

        System.out.println("Client: received a notification of type "
            + notif.getType() + "\nwith the sequence number "
            + notif.getSequenceNumber());
    }
    [...]  // main omitted
}
```

As explained in the notification mechanism “8.1 Overview of Notifications” on page 108, a listener on the agent side is typically an MBean that receives notifications about the status of other MBeans and then processes or exposes this information in some manner. Only if a key value or some management event is observed will this information be passed to a listening manager, probably by sending a different notification.

In this manner, the notification model reduces the communication that is necessary between agents and managers. Your management solution determines how much decisional power resides in the agent and when situations are escalated. These parameters will affect your design of the notification flow between broadcasters, listeners, agents, and managers.

The usual role of a manager-side listener is to process the important information in a notification and take the appropriate action. As we shall see, our notification example is much simpler. Our goal is not to construct a real-world example, but to demonstrate the mechanisms that are built into the Java DMK.

22.1.3 Adding a Listener Through the Connector

By extension of the `ClientNotificationHandler` interface, the `RemoteMBeanServer` interface exposes methods for adding and removing listeners. The signatures of these methods are identical to those of the agent-side `MBeanServer` interface. The only difference is that they are implemented in the connector client classes that make the communication protocol transparent.

Our manager application uses the RMI protocol connector. After creating the connector client object, we use the methods of its `RemoteMBeanServer` interface to create our broadcaster MBean and then register as a listener to this MBean’s notifications.

EXAMPLE 22–3 Adding a Listener through the Connector

```
// Use RMI connector on port 8086 to communicate with the agent
System.out.println(">>> Create an RMI connector client");
```

EXAMPLE 22–3 Adding a Listener through the Connector *(Continued)*

```
RmiConnectorClient connectorClient = new RmiConnectorClient();

// agentHost was read from the command line or defaulted to localhost
RmiConnectorAddress rmiAddress = new RmiConnectorAddress(
    agentHost, 8086, com.sun.jdmk.ServiceName.RMI_CONNECTOR_SERVER);
connectorClient.connect(rmiAddress);

// Wait 1 second for connecting
Thread.sleep(1000);

// Create the MBean in the agent
ObjectName mbean = new ObjectName ("Default:name=NotificationEmitter");
connectorClient.createMBean("NotificationEmitter", mbean);

// Now add ourselves as the listener (no filter, no handback)
connectorClient.addNotificationListener(mbean, this, null, null);
```

You can see how similar this code is to the agent application by comparing it with the code example for "8.2.3 Adding a Listener Through the MBean Server" on page 112.

If you have generated and instantiated proxy MBeans for your broadcaster MBeans, you can also register through the `addNotificationListener` method that they expose.

Again, the method signatures defined in a proxy MBean are identical to those of the `MBeanServer` or `NotificationBroadcasterClient` interfaces for adding or removing listeners. See the code example for "8.3.3 Adding a Listener Directly to an MBean" on page 116. Listeners added through a proxy MBean receives the same notifications as listeners added to the same MBean through the interface of the connector client.

Note – Following the Java programming model, the connector client limits its resource usage by only running one thread to notify all of its listeners. This thread calls all of the handler callback methods that have been added through this connector. Therefore, the callbacks should return quickly and use safe programming to avoid crashing the connector client.

22.2 Push Mode

The push mode of notification broadcasting only exists for the legacy connectors. The new connectors implemented in Java DMK 5.1 all use pull mode. Push mode is presented here purely for reasons of backwards compatibility with older versions of Java DMK.

Because the broadcaster and the listener are running on separate hosts or in separate virtual machines on the same host, their notifications must be forwarded from one to the other. The mechanism for doing this is completely transparent to the users of the Java DMK components.

Briefly, the legacy connector client instructs the legacy connector server to add its own agent-side listener to the designated broadcaster using the methods of the MBeans server. Then, the connector server implements a buffering cache mechanism to centralize notifications before serializing them to be forwarded to the connector client. By design, it is the connector client in the manager application that controls the buffering and forwarding mechanism for a connection.

Figure 22–1 summarizes the notification forwarding mechanism and its actors. In particular, it shows how the connector server uses internal listener instances to register locally for MBean notifications, even if this mechanism is completely hidden from the user. The path of listener registration through calls to the `addNotificationListener` method of the various interfaces is paralleled by the propagation of notifications through calls to the listeners' `handleNotification` method.

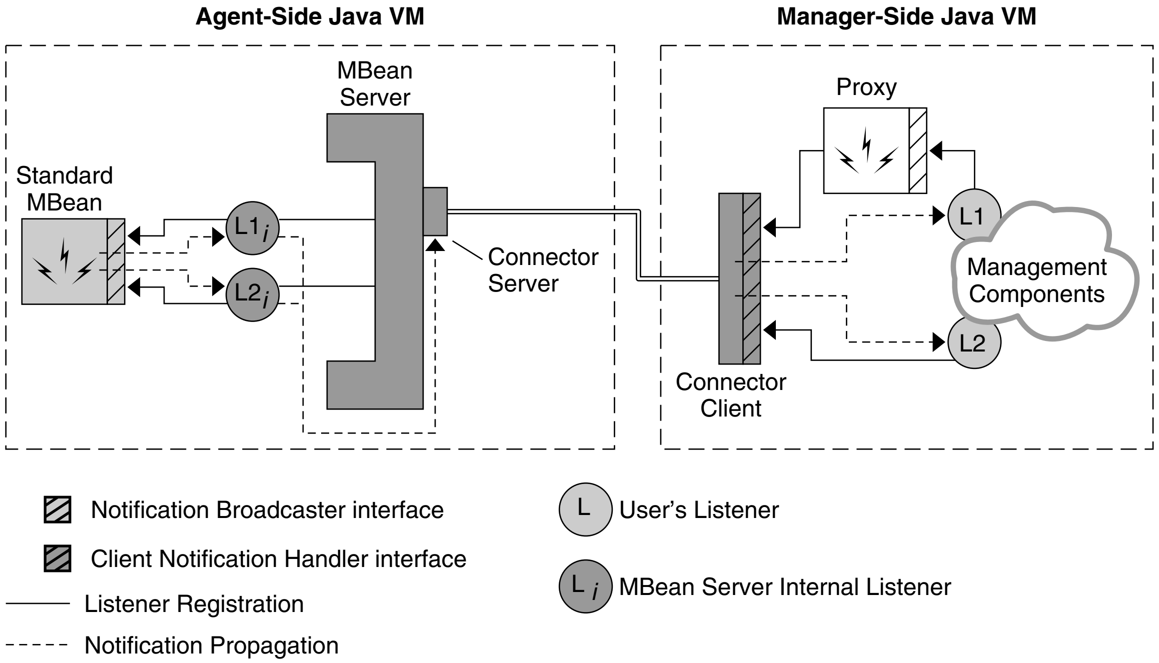


FIGURE 22–1 Notification Forwarding Internals

Neither the broadcaster nor the listener need to implement any extra methods, or even be aware that the other party is remote. Only the designer needs to be aware of communication issues such as delays. You cannot expect the listener to be invoked instantaneously after a remote broadcaster sends a notification.

The forwarding mechanism allows you to configure how and when notifications are forwarded. This enables you to optimize the communication strategy between your agents and managers. There are two basic modes for notification forwarding in legacy connectors: push mode and pull mode. A notification in the connector server's cache is either pushed to the manager at the agent's initiative, or pulled by the manager at its own initiative.

The push mode for notification forwarding is the simplest because it implements the expected behavior of a notification. When a notification is sent from an MBean to its listener, it is immediately pushed to the manager-side where the listener's handler method is called. There is no delay in the caching, and if the communication layer is quick enough, the listener is invoked almost immediately after the notification is sent.

Push mode is the default forwarding policy of a newly instantiated connector client.

In our manager example, we explicitly set the connector client in push mode and then trigger the agent-side notifications.

EXAMPLE 22–4 Switching to the Notification Push Mode

```
System.out.println("\n>>> Set notification forward mode to PUSH.");
connectorClient.setMode(ClientNotificationHandler.PUSH_MODE);

System.out.println(">>> Have our MBean broadcast 10 notifications...");
params[0] = new Integer(10);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);

System.out.println(">>> Done.");
System.out.println(">>> Receiving notifications...\n");

// Nothing to do but wait while our handler receives the notifications
Thread.sleep(2000);
```

The connector client exposes the methods for controlling the agent's notification buffer. This caching buffer is not used in push mode, so these methods do not affect pushed notifications. The methods do however set internal variables that will be taken into account if and when pull mode is enabled.

The advantage of push mode is that it works without any further intervention: notifications eventually reach their remote listeners. Push mode works when the communication layer and the listener's processing capacity are adapted to the notification emission rate, or more specifically to the potential emission rate. Because all notifications are immediately sent to the manager hosts, a burst of notifications will cause a burst of traffic that might or might not be adapted to the communication layer.

If your communication layer is likely to be saturated, either your design should control broadcasters to prevent bursts of notifications, or you should use the pull mode which has this control functionality built-in. The push mode is ideal if you have reliable and fast communication between your agents and your managers. You can also dynamically switch between modes, enabling a management application to fine-tune its communication policy depending on the number of notifications that must be handled.

22.3 Pull Mode

In pull mode, notifications are not immediately sent to their remote listeners. Rather, they are stored in the connector server's internal buffer until the connector client requests that they be forwarded. Instead of being sent individually, the notifications are grouped to reduce the load on the communication layer. Pull mode has the following settings that let the manager define the notification forwarding policy:

- A period for automatic pulling
- The size of the agent-side notification buffer (also called the cache)

- A policy for discarding notifications when this buffer is full

For a given connection, there is one cache for all listeners, not one cache per listener. This cache therefore has one buffering policy whose settings are controlled through the methods exposed by the connector client. The cache buffer contains an unpredictable mix of notifications in transit to all manager-side listeners added through a given connector client or through one of its bound proxy MBeans. The buffer operations such as pulling or overflowing apply to this mix of notifications, not to any single listener's notifications.

22.3.1 Periodic Forwarding

Pull mode forwarding is necessarily a compromise between receiving notifications in a timely manner, not saturating the communication layer, and not overflowing the buffer. Notifications are stored temporarily in the agent-side buffer, but the manager-side listeners still need to receive them. Pull mode includes automatic pulling that retrieves all buffered notifications regularly.

The frequency of the pull forwarding is controlled by the pull period expressed in milliseconds. By default, when pull mode is enabled, the manager will automatically begin pulling notifications once per second. Whether or not there are any notifications to receive depends upon events in the agent.

Our manager application sets a half-second pull period and then triggers the notification broadcaster.

EXAMPLE 22–5 Pulling Notifications Automatically

```
System.out.println(">>> Set notification forward mode to PULL.");
connectorClient.setMode(ClientNotificationHandler.PULL_MODE);

// Retrieve buffered notifications from the agent twice per second
System.out.println(">>> Set the forward period to 500 milliseconds.");
connectorClient.setPeriod(500);

System.out.println(">>> Have our MBean broadcast 20 notifications...");
params[0] = new Integer(20);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);
System.out.println(">>> Done.");

// Wait for the handler to process all notifications
System.out.println(">>> Receiving notifications...\n");
Thread.sleep(2000);
```

When notifications are pulled, *all* notifications in the agent-side buffer are forwarded to the manager and the registered listeners. It is not possible to set a limit on the number of notifications that are forwarded except by limiting the size of the buffer (see

“22.3.3 Agent-Side Buffering” on page 448). Even in a controlled example such as ours, the number of notifications in the agent-side buffer at each pull period is completely dependent upon the agent’s execution paths, and therefore unpredictable from the manager-side.

22.3.2 On-Demand Forwarding

You can disable automatic pulling by setting the pull period to zero. In this case, the connector client will not pull any notifications from the agent until instructed to do so. Use the `getNotifications` method of the connector client to pull all notifications when desired. This method will immediately forward *all* notifications in the agent-side buffer. Again, it is not possible to limit the number of notifications that are forwarded except by limiting the buffer size.

In this example, we disable the automatic pulling and then trigger the notification broadcaster. The notifications will not be received until we request that the connector server pull them. Then, all of the notifications will be received at once.

EXAMPLE 22–6 Pulling Notifications by Request

```
System.out.println(">>> Use pull mode with period set to zero.");
connectorClient.setMode(ClientNotificationHandler.PULL_MODE);
connectorClient.setPeriod(0);

System.out.println(">>> Have our MBean broadcast 30 notifications...");
params[0] = new Integer(30);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);
System.out.println(">>> Done.");

// Call getNotifications to pull all buffered notifications from the agent
System.out.println("\n>>> Press Enter to pull the notifications.");
System.in.read();
connectorClient.getNotifications();

// Wait for the handler to process all notifications
Thread.sleep(100);
```

In the rest of our example, we use the on–demand forwarding mechanism to control how many notifications are buffered on the agent-side and thereby test the different caching policies.

22.3.3 Agent-Side Buffering

In pull mode, notifications are stored by the connector server in a buffer until they are pulled by the connector client. Any one of the pull operations, whether on-demand or periodic, empties this buffer, and it fills up again as new notifications are triggered.

By default, this buffer will grow to contain all notifications. The ClientNotificationHandler interface defines the static NO_CACHE_LIMIT field to represent an unlimited buffer size. If the notifications are allowed to accumulate indefinitely in the cache, this can lead either to an "out of memory" error in the agent application, a saturation of the communication layer, or an overload of the manager's listeners when the notifications are finally pulled.

To change the size of the agent's cache, call the connector client's setCacheSize method. The size of the cache is expressed as the number of notifications that can be stored in its buffer. When a cache buffer of limited size is full, new notifications will overflow and be lost. Therefore, you should also choose an overflow mode when using a limited cache size. The two overflow modes are defined by static fields of the ClientNotificationHandler interface:

- DISCARD_OLD: The oldest notifications will be lost and the buffer will always be renewed with the latest notifications that have been triggered. This is the default value when a limit is first set for the cache size.
- DISCARD_NEW: Once the notification buffer is full, any new notifications will be lost until the buffer is emptied by forwarding the messages. The buffer will always contain the first notifications triggered after the previous pull operation.

We demonstrate each of these modes in our sample manager, first by setting the cache size and the overflow mode, then by triggering more notifications than the cache buffer can hold.

EXAMPLE 22–7 Controlling the Agent-Side Buffer

```
System.out.println(">>> Use pull mode with period set to zero, " +
    "buffer size set to 10, and overflow mode set to DISCARD_OLD.");
connectorClient.setMode(ClientNotificationHandler.PULL_MODE);
connectorClient.setPeriod(0);
connectorClient.setCacheSize(10, true); // see "Buffering Specifics"
connectorClient.setOverflowMode(ClientNotificationHandler.DISCARD_OLD);

System.out.println(">>> Have our MBean broadcast 30 notifications...");
params[0] = new Integer(30);
signatures[0] = "java.lang.Integer";
connectorClient.invoke(mbean, "sendNotifications", params, signatures);
System.out.println(">>> Done.");

// Call getNotifications to pull all buffered notifications from the agent
System.out.println("\n>>> Press Enter to get notifications.");
System.in.read();
connectorClient.getNotifications();

// Wait for the handler to process the 10 notifications
// These should be the 10 most recent notifications
// (the greatest sequence numbers)
Thread.sleep(100);
System.out.println("\n>>> Press Enter to continue.");
System.in.read();
```

EXAMPLE 22–7 Controlling the Agent-Side Buffer *(Continued)*

```
// We should see that the 20 other notifications overflowed the agent buffer
System.out.println(">>> Get overflow count = " +
    connectorClient.getOverflowCount());
```

The overflow count gives the total number of notifications that have been discarded because the buffer has overflowed. The number is cumulative from the first manger-side listener registration until all of the manager's listeners have been unregistered. The manager application can modify or reset this value by calling the `setOverflowCount` method.

In our example application, we repeat the actions above, to cause the buffer to overflow again, but this time using the `DISCARD_NEW` policy. Again, the buffer size is ten and there are thirty notifications. In this mode, the first ten sequence numbers remain in the cache to be forwarded when the manager pulls them from the agent, and twenty more will overflow.

22.3.3.1 Buffering Specifics

When the buffer is full and notifications need to be discarded, the time reference for applying the overflow mode is the order in which notifications have arrived in the buffer. Neither the time stamps nor the sequence numbers of the notifications are considered, because neither of these are necessarily absolute; even the sequence of notifications from the same broadcaster can be non-deterministic. And in any case, broadcasters are free to set both time stamps and sequence numbers as they see fit, or even to make them `null`.

The second parameter of the `setCacheSize` method is a boolean that determines whether or not the potential overflow of the cache is discarded when reducing the cache size. If the currently buffered notifications do not fit into the new cache size and this parameter is `true`, excess notifications are discarded according to the current overflow mode. The overflow count is also updated accordingly.

In the same situation with the parameter set to `false`, the cache will not be resized. You need to check the return value of the method when you set this parameter to `false`. If the cache cannot be resized because it would lead to discarded notifications, you need to empty the cache and try resizing the cache size again. To empty the cache, you can either pull the buffered notifications with the `getNotifications` method or discard them all by calling the connector client's `clearCache` method.

When the existing notifications fit within the new cache size or when increasing the cache size, the second parameter of `setCacheSize` has no effect.

Because several managers can connect through the same connector server object, it must handle the notifications for each separately. This implies that each connected manager has its own notification buffer and its own settings for controlling this cache. The overflow count is specific to each manager as well.

22.3.3.2 Buffering Generalities

Here we have demonstrated each setting of the forwarding mechanism independently by controlling the notification broadcaster. In practice, periodic pulling, agent-side buffering and buffer overflow can all be happening at once. And you can call `getNotifications` at any time to do an on-demand pull of the notifications in the agent-side buffer. You should adjust the settings to fit the known or predicted behavior of your management architecture, depending upon communication constraints and your acceptable notification loss rate.

The caching policy is completely determined by the manager application. If notification loss is unacceptable, it is the manager's responsibility to configure the mechanism so that they are pulled as often as necessary. Also, the legacy notification mechanism can be updated dynamically. For example, the manager can compute the notification emission rate and update any of the settings (buffer size, pull period, and overflow mode) to minimize the risk of a lost notification.

22.4 Running the Legacy Notification Forwarding Example

The *examplesDir*`/legacy/Notification` directory contains all of the files for the broadcaster MBean, the `BaseAgent` application, and our `Client` application that is the listener object. To run the legacy notification forwarding example, we use the `BaseAgent` application that contains an RMI connector server.

▼ To Run the Legacy Notification Forwarding Example

1. **Compile all files in this directory with the `javac` command.**

   For example, on the Solaris platform with the Korn shell, type:

   ```
   $ cd examplesDir/legacy/Notification/
   $ javac -classpath classpath *.java
   ```

2. **Start the agent on another host or in another terminal window with the following command.**

   Be sure that the classes for the `NotificationEmitter` MBean can be found in its *classpath*:

   ```
   $ java -classpath classpath BaseAgent
   ```

3. **Wait for the agent to be completely initialized, then start the manager in another window with the following command, where *hostname* is the name of the host running the agent.**

   If you started the agent on the same host, you can omit the *hostname*:

   ```
   $ java -classpath classpath Client hostname
   ```

   When started, the manager application first creates the `NotificationEmitter` MBean and then registers itself as a listener.

4. **Press `Enter` when the application pauses to step through the various notification forwarding situations that we have seen in this topic.**

5. **Press `Enter` again in the manager window to exit the application.**

   Leave the agent application running if you want to interact with the example through the HTML protocol adaptor of the `BaseAgent`.

▼ To Interact With the Legacy Notification Forwarding Mechanism

1. **Start the manager in another window with the following command, where *hostname* is the name of the host running the agent.**

   If you started the agent on the same host, you can omit the *hostname*:

   ```
   $ java -classpath classpath Client hostname
   ```

2. **Load the following URL in your browser and go to the MBean view of the `NotificationEmitter` MBean:**

   `http://`*hostname*`:8082/`

   If you get an error, you might have to switch off proxies in your browser preference settings. Any browser on your local network can also connect to this agent using this URL.

3. **When the manager application pauses for the first time, call the `sendNotifications` method from your browser with a small integer as the parameter.**

   The listener handles your notifications in the manager's terminal window. Because the manager is still in push mode, they are forwarded immediately.

4. **Press `Enter` in the manager window.**

   The manager is now in pull mode with a pull period of 500 milliseconds.

5. **Through the MBean view, send 1000 notifications.**

   If your agent's host is slow enough, or if your manager's host is fast enough, you might be able to see the manager pause briefly after it has processed all notifications from one period and before the next ones are forwarded.

6. **Press `Enter` in the manager window.**

   The agent now forwards notifications by request.

7. **Through the MBean view, send 15 notifications, then press `Enter` again.**

   The manager pulls all of the notifications: the 30 triggered by the manager and the 15 we just triggered. They were all kept in the buffer, waiting for the manager's request to forward them. Remember that the `sendNotifications` operation resets the sequence numbering every time it is invoked.

8. **Press `Enter` in the manager's window.**

   The cache size is set to 10 notifications and the overflow mode to `DISCARD_OLD`.

9. **Through the MBean view, send 15 more notifications, then press `Enter` again.**

   Only the last 10 of our notifications fit into the cache buffer. All the rest, including those already triggered by the manager, overflow and are discarded.

10. **Press `Enter` to see that the discarded notifications are tallied in the overflow count.**

11. **Press `Enter` in the manager's window.**

    The cache size is still 10 notifications and the overflow mode is set to `DISCARD_NEW`.

12. **Through the MBean view, send only 5 more notifications, then press `Enter` again.**

    The first 10 of the manager-triggered notifications are received. All of the more recent notifications, including ours, overflow the cache buffer and are lost.

13. **Press `Enter` to see that the lost notifications are tallied in the overflow count:**

    35 from Step 12 plus 25 more from this step, for a total of 60.

14. **Press `Enter` in the manager's window again to stop the `Client` application.**

15. **Press `Enter` in the other window to stop the agent application when you have finished running the example.**

CHAPTER 23

Legacy Connector Security

The security mechanisms presented in this chapter relate to the legacy connectors described in Chapter 21. These legacy connectors, and hence their associated security mechanisms, have been superseded by the connectors and security mechanisms brought to Java Dynamic Management Kit (Java DMK) 5.1 by the inclusion of Java Management Extensions (JMX) Remote API, and are retained only for reasons of backwards compatibility.

There are two categories of access-control implemented in the legacy connectors: connection-level control through a password and request-level control through a context object. Context checkers work as filters between the connector server and the MBean server. The filter logic can be determined dynamically, based on the nature of the request and on a context object provided by the client.

Security in the communication layer is achieved through the cryptography of a Secure Socket Layer (SSL) and the secure hypertext transfer protocol (HTTPS) connector. Using other components of the Java platform, legacy connectors can effectively make all open communication undecipherable.

The code samples in this chapter are taken from the files in the `legacy/Context` example directory located in the main *examplesDir* (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "23.1 Password-Based Authentication (Legacy Connectors)" on page 456 shows how to provide connection-level access–control through the HTTP–based legacy connectors.
- "23.2 Context Checking" on page 459 demonstrates the filter mechanism for fine-grained access control of incoming requests to an agent.
- "23.3 Legacy HTTPS Connector" on page 465 explains how to use the older security tools of the Java platform to implement cryptography on the data between agents and managers.

23.1 Password-Based Authentication (Legacy Connectors)

The simplest form of agent security you can implement in the legacy connectors is to accept management requests only if they contain a valid login identity and password. Agents recognize a given list of login-password pairs, and managers must provide a matching login and password when they try to establish a connection.

Among the legacy connectors, only the HTTP–based connectors support password–based authentication. However, both the new remote method invocation (RMI) and JMX messaging protocol (JMXMP) connectors added in Java DMK 5.1 support password-based authentication. The SNMP protocol adaptor also supports access control, but it is based on a different mechanism (see "19.1 IP-Based Access Control Lists" on page 353).

By default, no authentication is enabled in the HTTP-based legacy connectors and any manager can establish a connection. The password-checking behavior is enabled by defining the list of authorized login-password pairs.

You can define this authentication information in one of the following ways:

- Through the constructor that takes an `AuthInfo` array parameter: `HttpConnectorServer(int port, AuthInfo[] authInfoList)`
- Through the methods inherited from the `GenericHttpConnectorServer` class: `addUserAuthenticationInfo(AuthInfo authinfo)` `removeUserAuthenticationInfo(AuthInfo authinfo)`

In both cases, only the agent application has access to these methods, meaning that the agent controls the authentication mechanism. As soon as an `AuthInfo` object is added to the connector server through either method, all incoming requests must provide a recognized name and password. In our example, we read the authentication information from the command line and call the `addUserAuthenticationInfo`.

EXAMPLE 23–1 Implementing Password Authentication in the Legacy HTTP Connector Server

```
// Here we show the code for reading the
// id-password pairs from the command line
//
int firstarg = 0;
boolean doAuthentication = (args.length > firstarg);

AuthInfo[] authInfoList;

if (doAuthentication) {
    authInfoList = new AuthInfo[(args.length - firstarg) / 2];
    for (int i = firstarg, j = 0; i < args.length; i += 2, j++)
```

EXAMPLE 23–1 Implementing Password Authentication in the Legacy HTTP Connector Server *(Continued)*

```
        authInfoList[j] = new AuthInfo(args[i], args[i + 1]);

} else

    authInfoList = null;

[...] // instantiate and register an HTTP connector server

// Define the authentication list
//
if (doAuthentication) {
    for (int i = 0; i < authInfoList.length; i++)
        http.addUserAuthenticationInfo(authInfoList[i]);
}
```

On the manager-side, identifiers and passwords are given in the address object, because authentication applies when the connection is established.

EXAMPLE 23–2 Specifying the Login and Password in the Legacy HTTP Connector Server

```
// login and password were read from the command line
//
AuthInfo authInfo = null;

if (login != null) {
    authInfo = new AuthInfo( login, password );
}

// agentHost and agentPort are read from the command
// line or take on default values
//
HttpConnectorAddress addr =
    new HttpConnectorAddress(
        agentHost, agentPort, authInfo );

final RemoteMBeanServer connector =
    (RemoteMBeanServer) new HttpConnectorClient();

connector.connect( addr );
```

The connector is identified by the one `AuthInfo` object it uses to instantiate the connector address. If the agent has authentication enabled, both the login and the password must match one of the `AuthInfo` objects in the agent. If the agent does not perform authentication, providing a login and password has no effect because all connections are accepted.

If the authentication fails, the call to the `connect` method returns an exception. Normally, the client's code should catch this exception to handle this error case.

As demonstrated by the code examples, the authentication mechanism is very simple to configure. It prevents unauthorized access with very little overhead.

Note – The HTML adaptor provides a similar authentication mechanism, where the list of accepted identities is given to the server object. In the case of the HTML protocol, the web browser is the management application that must provide a login and password. The behavior is browser-dependent, but the browser usually requests that the user type this login and password in a dialog box.

23.1.1 Running the Legacy Security Example With Authentication

The *examplesDir*`/legacy/Context` directory contains the applications that demonstrate the use of password authentication through the legacy HTTP connector.

▼ To Run the Legacy Security Example With Authentification

1. **Compile all files in this directory with the `javac` command.**

   For example, on the Solaris platform with the Korn shell, type:

   ```
   $ cd examplesDir/legacy/Context/
   $ javac -classpath classpath *.java
   ```

2. **Start the agent in a terminal window and specify a list of login-password pairs, as in the following command:**

   ```
   $ java -classpath classpath ContextAgent  jack jill  billy bob
   ```

3. **Wait for the agent to be completely initialized, then start the manager in another window with the following command:**

   ```
   $ java -classpath classpath ContextClient -ident andy bob
   ```

   The client application tries to establish a connection with the login `andy` and the password `bob`. The authentication mechanism refuses the connection, and the `com.sun.jdmk.comm.UnauthorizedSecurityException` is raised by the connector server.

4. **Start the manager again, this time with a valid identity:**

   ```
   $ java -classpath classpath ContextClient -ident jack jill
   ```

   The connection is established and the output from management operation is displayed in both windows.

5. **Leave both applications running for the next example.**

23.2 Context Checking

Context checking is a more advanced security mechanism that can perform selective filtering of incoming requests. The context is an arbitrary object provided by the client and used by the server to decide whether or not to allow the request.

Filtering and context checking are performed in between the communicator server and the MBean server. The mechanism relies on two objects called the `MBeanServerForwarder` and the `MBeanServerChecker`.

23.2.1 Filter Mechanism

The `MBeanServerForwarder` allows for the principle of *stackable MBean servers*. An `MBeanServerForwarder` implements the `MBeanServer` interface and one extra method called `setMBeanServer`. Its function is to receive requests and forward them to the designated MBean server.

The `setMBeanServer` method of a communicator server object enables you to specify the MBean server that fulfills its requests. By chaining one or more `MBeanServerForwarder` objects between a communicator server and the actual MBean server, the agent application creates a stack of objects that can process the requests before they reach the MBean server.

The `MBeanServerChecker` is an extension of the forwarder that forces each request to call a *–checker* method. By extending the `MBeanServerChecker` class and providing an implementation of the checker methods, you can define a policy for filtering requests before they reach the MBean server. Table 23–1 shows the checker methods that apply to groups of `MBeanServer` methods.

TABLE 23–1 Filter Method Granularity for Context Checking

Filter Method	MBean Server Operations Filtered
`checkAny`	Every method of the `MBeanServer` interface
`checkCreate`	All forms of the `create` and `registerMBean` methods
`checkDelete`	The `unregisterMBean` method
`checkInstantiate`	All forms of the `instantiate` method
`checkInvoke`	The `invoke` method that handles all operation invocations

TABLE 23–1 Filter Method Granularity for Context Checking *(Continued)*

Filter Method	MBean Server Operations Filtered
`checkNotification`	Both `addNotificationListener` and `removeNotificationListener`
`checkQuery`	Both `queryMBeans` and `queryNames`
`checkRead`	All methods that access but do not change the state of the agent: `getAttribute`, `getAttributes`, `getObjectInstance`, `isRegistered`, `getMBeanCount`, `getDefaultDomain`, `getMBeanInfo`, and `isInstanceOf`
`checkWrite`	The `setAttribute` and `setAttributes` methods

As a request passes through a stack of MBean servers, the checker methods are called to determine if the request is allowed. In order to identify the manager that issued a request, the checker can access the *operation context* of the request.

The operation context, or just context, is an object defined by the manager which seeks access through a context checker. It usually contains some description of the manager's identity. The only restriction on the context object is that it must implement the `OperationContext` interface. The context object is passed from the connector client to the connector server and is then associated with the execution of a request. Conceptually, this object is stored in the user accessible context of the thread that executes the request.

All methods in the `MBeanServerChecker` class can access the context object by calling the protected `getOperationContext` method. The methods of the context checker then implement some policy to filter requests based on the context object, the nature of the request, and the data provided in the request, such as the attribute or operation name.

Figure 23–1 shows the paths of two requests through a stack of MBean server implementations, one of which is stopped by the context checker because it does not provide the correct context.

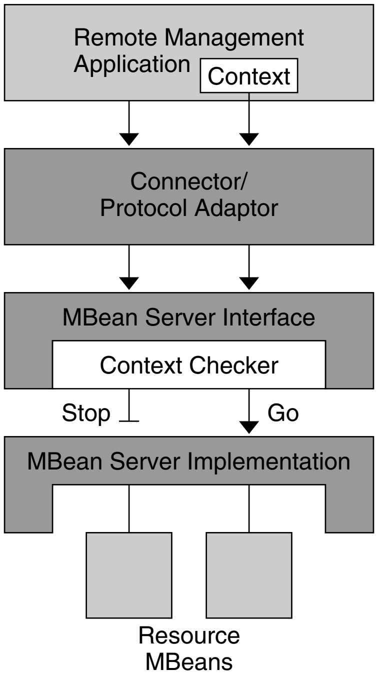

Remote Management Application
Context
Connector/ Protocol Adaptor
MBean Server Interface
Context Checker
Stop
Go
MBean Server Implementation
Resource MBeans

FIGURE 23–1 Context Checking in Stackable MBean Servers

Only connectors fully support the context mechanism. Their connector clients expose the methods that allow the manager to specify the context object. Existing protocol adaptors have no way to specify a context. Their requests can be filtered and checked, but their context object will always be `null`.

This functionality can still be used to implement a filtering policy, but without a context object, straightforward manager identification is not possible. However, a proprietary protocol adaptor could define some mapping to determine a context object that could be accepted by the filters.

23.2.2 Context Implementation

An agent wanting to implement context checking on a legacy connector first needs to extend the `MBeanServerChecker` class. This class retrieves the context object and determines whether any given operation is allowed.

EXAMPLE 23–3 Implementation of the Context Checker

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.QueryExp;
```

EXAMPLE 23–3 Implementation of the Context Checker *(Continued)*

```
import com.sun.jdmk.MBeanServerChecker;
import com.sun.jdmk.OperationContext;

public class ContextChecker extends MBeanServerChecker {

    // Constructor
    public ContextChecker(MBeanServer mbs) {
        super(mbs);
    }

    // Implementation of the abstract methods of the
    // MBeanServerChecker class: for each of the specific
    // checks, we just print out a trace of being called.
    [...]

    protected void checkWrite( String methodName,
                               ObjectName objectName) {
        System.out.println("checkWrite(\"" + methodName +
                           "\", " + objectName + ")");
    }

    protected void checkQuery( String methodName,
                               ObjectName name,
                               QueryExp query) {
        System.out.println("checkQuery(\"" + methodName +
                           "\", " + name + ", " + query + ")");
    }

    [...]

    /**
     * This is where we implement the check that requires every
     * operation to be called with an OperationContext whose
     * toString() method returns the string "nice".
     */
    protected void checkAny( String methodName,
                             ObjectName objectName ) {

        System.out.println("checkAny(\"" + methodName + "\", " +
                           objectName);
        OperationContext context = getOperationContext();
        System.out.println("  OperationContext: " + context);

        if (context == null || !context.toString().equals("nice")) {
            RuntimeException ex =
                new SecurityException(" Bad context: " + context);
            ex.printStackTrace();
            throw ex;
        }
    }

}
```

The agent application then instantiates its context checkers and stacks them between the communicator servers and the MBean server. Each communicator server has its own stack, although filters and context checkers can be shared. The agent performs the stacking inside a synchronized block because other threads can try to do stacking simultaneously.

EXAMPLE 23–4 Stacking MBean Server and Context Checkers

```
// Create MBeanServer
//
MBeanServer server = MBeanServerFactory.createMBeanServer();

/* Create context checker.  The argument to the constructor is
 * our MBean server to which all requests will be forwarded
 */
ContextChecker contextChecker = new ContextChecker( server );


[...] // Create HTTP connector server

/* Add the context checker to this HTTP connector server.
 * We point it at the context checker which already points
 * to the actual MBean server.
 * It is good policy to check that we are not sidetracking
 * an existing stack of MBean servers before setting ours.
 */
synchronized (http) {
    if (http.getMBeanServer() != server) {
        System.err.println("After registering connector MBean, " +
           "http.getMBeanServer() != " + "our MBeanServer");
        System.exit(1);
    }
    http.setMBeanServer(contextChecker);
}
```

Finally, the manager operation defines a context object class and then provides a context object instance through its connector client.

EXAMPLE 23–5 Setting the Context in the Connector Client

```
/* In this example, the agent checks the OperationContext of
   each operation by examining its toString() method, so we
   define a simple implementation of OperationContext whose
   toString() is a constant string supplied in the constructor
 */
class StringOperationContext
          implements OperationContext, Cloneable {

    private String s;

    StringOperationContext(String s) {
        this.s = s;
    }
```

EXAMPLE 23–5 Setting the Context in the Connector Client *(Continued)*

```
    public String toString() {
        return s;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// the contextName must be provided on the command line
OperationContext context =
    new StringOperationContext(contextName);

[...]

// The context is set for all requests issued through
// the connector client; it can be changed at any time
connector.setOperationContext(context);
```

23.2.3 Running the Legacy Security Example With Context Checking

The `ContextClient` and `ContextAgent` applications in the *examplesDir*`/legacy/Context` directory also demonstrate the use of stackable MBean servers and context checking through the legacy HTTP connector.

If you have not done so already, compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, type:

```
$ cd examplesDir/legacy/Context/
$ javac -classpath classpath *.java
```

▼ To Run the Legacy Security Example With Context Checking

1. **If the agent and client applications are not already running from the previous example, type the following commands in separate windows:**

   ```
   $ java -classpath classpath ContextAgent
   ```

   ```
   $ java -classpath classpath ContextClient
   ```

   The *classpath* should include the current directory (`.`) for both applications because they rely on classes that were compiled in this directory.

2. **Press `Enter` in the client application to trigger another set of requests.**

The agent window displays the output of the `ContextChecker` class. We can see that the `checkAny` method verifies the "`nice`" context of every request and that the other checkers just print out their names, providing a trace of the request.

3. **Stop both applications by pressing `Control-C` in each of the windows.**

4. **Restart both applications, but specify a different context string for the client:**

```
$ java -classpath classpath ContextAgent
```

```
$ java -classpath classpath ContextClient -context BadToTheBone
```

This time the context is not recognized. The agent raises a `java.lang.SecurityException` that is propagated to the client, which then exits.

5. **Press `Control-C` in the agent window to stop the `ContextAgent` application.**

23.3 Legacy HTTPS Connector

The legacy HTTPS connector provides data encryption and certificate-based security through a Secure Socket Layer (SSL). The Java Secure Socket Extension (JSSE) provides a implementation of secure sockets for the Java 2 platform, Standard Edition 1.4.

The web site for the JSSE is `http://java.sun.com/products/jsse`. For further information and details regarding the use of the secure sockets, refer to the JSSE documentation.

The legacy HTTPS connector exposes the same interfaces as all other legacy connectors and has exactly the same behavior. The development of a management application that relies on the HTTPS connector is no different from that of any other Java dynamic manager. See "21.2 Legacy Connector Clients" on page 423 for details about programming with the `RemoteMBeanServer` API.

This section covers the steps that are required to establish a secure connection between your agent and manager applications. These instructions do not guarantee total security. They just explain the programmatic steps needed to ensure data security between two remote Java applications.

Before performing these steps, run each of your manager and agent applications on a separate host, and ensure that each host has its own installation of the Java platform (not a shared network installation).

▼ To Establish a Secure HTTPS Connection

1. **Generate public and private keys.**

   Repeat this step on all agent and manager hosts.

   Generate a key pair (a public key and associated private key).

   Wrap the public key into an X.509 v1 self-signed certificate, which is stored as a single-element certificate chain. This certificate chain and the private key are stored in a new keystore entry identified by *alias*.

   In the following command, the –dname parameters designates the X.500 Distinguished Name for the host where you are generating the certificates. The *commonName* field must be the host name.

   ```
   $ keytool -genkey -alias alias -keyalg RSA -keysize 1024 -sigalg MD5withRSA
             -dname "CN=commonName, OU=orgUnit, O=org, L=location,
   S=state, C=country"
             -keypass passPhrase -storetype jks -keystore yourHome/.keystore
             -storepass passPhrase
   ```

2. **Export a local certificate**

   Repeat this step on all agent and manager hosts.

   Read the certificate that is associated with your *alias* from the keystore and store it in a *hostCertFile*:

   ```
   $ keytool -export -alias alias -file hostCertFile -storetype jks
             -keystore yourHome/.keystore -storepass passPhrase -rfc
   ```

   When you are done with this step, you will have a certificate for each of your hosts.

3. **Import all remote certificates**

   Repeat this step on both the agent and manager hosts for all pairs of agent-managers in your management architecture.

   In this step, agent and manager pairs must exchange their certificates. The manager imports the agent's *hostCertFile* and the agent imports the manager's *hostCertFile*. If a manager has two agents, it will import two certificates and each agent will import a copy of the manager's certificate.

   Import the certificate into the file containing the trusted Certificate Authorities (CA) certificates. This will add our self-signed certificate as a trusted CA certificate to the `cacerts` file so that the server and the client will be able to authenticate each other.

   ```
   $ keytool -import -alias alias -file hostCertFile -noprompt -trustcacerts
             -storetype jks -keystore JAVAhome/jre/lib/security/cacerts
             -storepass changeit
   ```

   This command modifies the *JAVAhome*`/jre/lib/security/cacerts` that will affect all applications running on that installation. If you do not want to modify this file, you can create a file named `jssecacerts` and use it instead. The default location of this file is either *JAVAhome*`/lib/security/jssecacerts` or if that does not exist, then *JAVAhome*`/lib/security/cacerts`.

4. **Run your Java dynamic management agent**

   Start your agent applications with the following properties:

   ```
   $ java -Djavax.net.ssl.keyStore=yourHome/.keystore
          -Djavax.net.ssl.keyStoreType=jks
          -Djavax.net.ssl.keyStorePassword=passPhrase
          AgentClass
   ```

   If you are using the notification push mechanism, add the following property definition to the above command line:

   ```
   -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
   ```

5. **Run your management application**

   Start your management applications with the following properties:

   ```
   $ java -Djavax.net.ssl.keyStore=yourHome/.keystore
          -Djavax.net.ssl.keyStoreType=jks
          -Djavax.net.ssl.keyStorePassword=passPhrase
          -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
          ManagerClass
   ```

CHAPTER 24

Legacy Proxy Mechanism

As of version 5.1 of Java Dynamic Management Kit (Java DMK), proxies are generated using the `java.lang.reflect.Proxy` interface defined by the Java 2 Platform, Standard Edition (J2SE). However, if you want to implement proxies with the legacy connectors, you must use the legacy proxy mechanism described in this chapter. This information is still valid, and is provided for backwards compatibility.

In *Protocol Adaptors*, we saw how to access a remote agent and interact with its MBeans through connectors. Java DMK provides additional functionality that makes the remoteness of an agent and the communication layer even more transparent: *proxy objects* for registered MBeans.

A proxy is an object instance that represents an MBean, that mirrors the methods of the MBean, and whose methods are called directly by the calling process. The proxy transmits requests to the MBean, through the MBean server, possibly through a connector, and returns any responses to the calling process. Proxy objects can also register listeners for notifications that the MBean might emit.

The advantage of a proxy object is that it enables applications to have an instance of an object that represents an MBean, instead of accessing the MBean's management interface through methods of the MBean server or through a connector client. This can simplify both the conceptual design of a management system and the source code needed to implement that system.

The code samples in this chapter are from the files in the `legacy/SimpleClients` example directory located in the main *examplesDir* (see "Directories and Classpath" in the Preface).

This chapter covers the following topics:

- "24.1 Legacy Proxy Mechanism" on page 470 describes how legacy proxy objects are implemented and the interfaces on which they rely.
- "24.2 Standard MBean Proxies" on page 473 shows how to generate legacy proxy objects for standard MBeans and run the corresponding example.

- "24.3 Legacy Generic Proxies" on page 478 describes the legacy proxy objects for dynamic MBeans and shows how to run the corresponding example.
- Finally "24.4 Legacy Proxies for Java DMK Components" on page 482, explains how to use the pre-generated legacy proxy objects provided with the product.

24.1 Legacy Proxy Mechanism

Proxy objects simplify the interactions between an application and the MBeans it wants to manage. The purpose of a proxy is to call the methods that access the attributes and operations of an MBean, through its MBean server. These method calls can be rather tedious to construct at every invocation, so the proxy performs this task for the calling process.

For example to call an operation on an MBean, an application must call the `invoke` method of the MBean server and provide the MBean's object name, the operation name string, an array of parameter objects, and a signature array. The proxy is a class that codes this whole sequence, meaning that the application can call the `reset` method directly on the proxy instance.

Conceptually, a proxy instance makes the MBean server and a protocol connector completely transparent. Except for MBean registration and connector connection phases, all management requests on MBeans can be fully served through proxies, with identical results. However, all functionality of the Java DMK is available without using proxies, so their usage is never mandatory.

By definition, a proxy object has the same interface as its MBean: the proxy can be manipulated as if it were the MBean instance, except that all requests are transmitted through the MBean server to the actual MBean instance for processing. A standard MBean proxy exposes getters, setters, and operation methods. It can also register listeners for all notifications broadcast by their corresponding MBean. A dynamic MBean proxy, also known as a *generic proxy*, exposes generic methods that are identical to those of the `DynamicMBean` interface.

In addition, standard proxies are commonly called *proxy MBeans*, because they are themselves MBeans. They are generated with an MBean interface, and can therefore be registered as standard MBeans in an MBean server. This feature enables one agent to expose resources whose MBeans are actually located in another agent. An equivalent functionality is described in Chapter 14.

24.1.1 Legacy Local and Remote Proxies

Proxies can also be bound to objects called *handlers* that necessarily implement the `ProxyHandler` interface. The methods of this interface are a subset of MBean server methods, as listed in "21.2.2 `RemoteMBeanServer` Interface" on page 424. These are the only methods that a proxy needs to access its corresponding MBean and fulfill all management requests.

In versions of Java DMK prior to 5.1, the `RemoteMBeanServer` interface extends the `ProxyHandler` interface, meaning that proxy objects can be bound to any of the legacy connector clients. These are called *remote proxies*, because they are instantiated in an application that is distant from the agent and its MBean instances.

The implementation of the `MBeanServer` interface also implements the `ProxyHandler` interface, so that proxy objects can be bound to the MBean server itself. These are called *local proxies* because they are located in the same application as their corresponding MBeans. Local proxies help preserve the management architecture by providing the simplicity of performing direct method calls on MBeans, while still routing all operations through the MBean server.

The symmetry of remote and local proxies complements the symmetry that enables management components to execute either in an agent or in a management application. Provided that all proxy classes are available, management components that use MBean proxies can be instantiated in an agent and rely on the MBean server or can be instantiated in a remote manager where they interact with a connector client. Except for communication delays, the results of all operations are identical, and the same notifications are received, whether obtained through a local proxy or a remote proxy (see "22.1.3 Adding a Listener Through the Connector" on page 442).

Figure 24–1 shows local proxies that are instantiated in an agent and bound to the MBean server, and the same classes instantiated as remote proxies in a management application and bound to the connector client. Management components located in either the agent or the management application can interact with the local or remote proxies, respectively. Management components can also access the MBean server or the connector client directly, regardless of whether proxies are being used.

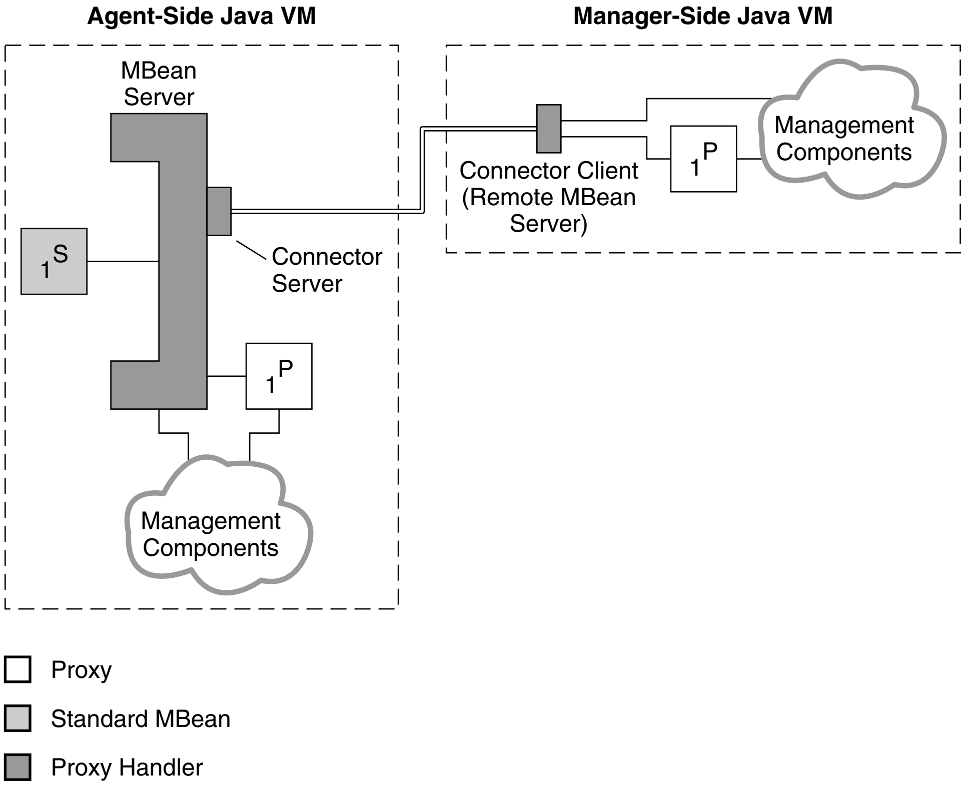


FIGURE 24–1 Interacting with Local and Remote Proxies

This diagram shows all possible relations between management components, proxies and their MBeans. Standard proxies can only represent a specific standard MBean, and generic proxies can represent any standard or dynamic MBean. In a typical management scenario, the management components are located in only one application, and for simplicity, they rarely instantiate more than one type of proxy.

Throughout the rest of this chapter, we do not distinguish between local and remote proxies. A proxy object, either standard or generic, is used in exactly the same way regardless of whether it is instantiated locally or remotely.

24.1.2 Legacy `Proxy` Interface

In addition to the methods for accessing the attributes and operations of its MBean, all proxies implement the methods of the `Proxy` interface. This interface contains the methods for binding the proxy instance to the proxy handler that can fulfill its requests. The `setServer` method binds the proxy to the handler object. Setting the server to `null` effectively unbinds the proxy object. The result of the `getServer` method can indicate whether or not a proxy is bound and if so, it will return a reference to the handler object.

The following list provides the functionality of the `Proxy` interface methods. See the Javadoc API of the `Proxy` interface for details.

`ObjectInstance`	`getMBeanObjectInstance()`
`ProxyHandler`	`getServer()`
`void`	`setServer(ProxyHandler server)`

Standard proxies can also implement the `NotificationBroadcasterProxy` interface if their corresponding MBean is a notification proxy. This interface contains the same `addNotificationListener` and `removeNotificationListener` methods that the MBean implements from the `NotificationBroadcaster` interface.

Applications that use proxies therefore have two ways to detect notification broadcasters. The first way relies on the implementation of the `NotificationBroadcasterProxy` interface that can be detected in the proxy's class inheritance. The second and more standard way is to look at the notifications listed in the MBean's metadata obtained by the `getMBeanInfo` method either from the server or through the proxy.

Generic proxies do not implement the `NotificationBroadcasterProxy` interface, so calling processes must use the MBean metadata for detecting broadcasters. In addition, generic proxies cannot register notification listeners, calling processes must do this directly through the server.

24.2 Standard MBean Proxies

A standard MBean proxy class is specific to its corresponding MBean class. Furthermore, a proxy instance is always bound to the same MBean instance, as determined by the object name passed to the proxy's constructor. Binding the proxy to its `ProxyHandler` object can be done through a constructor or set dynamically through the methods of the `Proxy` interface.

The methods of a standard proxy have exactly the same signature as those of the corresponding standard MBean. Their only task is to construct the complete management request, that necessarily includes the object name, and to transmit it to the MBean server or connector client. They also return any result directly to the caller.

Because the contents of all proxy methods are determined by the management interface of the MBean, the proxy classes can be generated automatically.

24.2.1 Generating Legacy Proxies for Standard MBeans

The deprecated `proxygen` tool provided with the Java DMK takes the class files of an MBean and its MBean interface and generates the Java source code files of its corresponding proxy object and proxy MBean interface. You then need to compile the two proxy files with the `javac` command and include the resulting class files in your application's classpath.

The `proxygen` compiler is fully documented in the *Java Dynamic Management Kit 5.1 Tools Reference Guide* and in the Javadoc API for the `ProxyGen` class. Its command—line options enable you to generate read-only proxies where all setter methods are suppressed and to define a package name for your proxies. For the purpose of the examples, we generate the default proxies without any of these options. See the section on "24.2.3 Running the Legacy Standard Proxy Example" on page 477.

Example 24–1 shows part of the code generated for the `SimpleStandard` MBean used in the `SimpleClients` examples.

EXAMPLE 24–1 Code Generated for the Legacy `SimpleStandardProxy` Class

```
public java.lang.String getState()
  throws InstanceNotFoundException, AttributeNotFoundException,
  ReflectionException, MBeanException {

    return ((java.lang.String)server.getAttribute(
                objectInstance.getObjectName(), "State"));
  }

public  void setState(java.lang.String value)
  throws InstanceNotFoundException, ReflectionException,
  AttributeNotFoundException,InvalidAttributeValueException,
  MBeanException {

    server.setAttribute(objectInstance.getObjectName(),
                        new Attribute("State",value));
}

public void reset()
  throws InstanceNotFoundException, ReflectionException,
  MBeanException {

  Object result;
  result= server.invoke(objectInstance.getObjectName(), "reset",
                        null, null);
```

EXAMPLE 24–1 Code Generated for the Legacy `SimpleStandardProxy` Class *(Continued)*

```
   }
```

You can modify the generated code if you want to customize the behavior of your proxies. However, customization is not recommended if your MBean interfaces are still evolving, because all modifications will need to be redone every time the proxies are generated.

24.2.2 Using Legacy Standard MBean Proxies

Once the proxies are generated and available in your application's classpath, their usage is straightforward. For each of the proxy objects it wants to use, the application needs to instantiate its proxy class and then bind it to a `ProxyHandler` object. The application is responsible for creating and binding all of the proxies that it wants to use, and it must unbind and free them when they are no longer needed.

Note – The binding methods in the `Proxy` interface in previous releases of Java DMK are deprecated in favor of the new `setServer` and `getServer` methods. This change is necessary so that proxies can be bound to any `ProxyHandler` object, to enable for both local and remote proxies.

All parameters for binding the proxy can be given in its constructor, which makes it very simple to instantiate and bind a proxy in one step.

EXAMPLE 24–2 Instantiating and Binding a Legacy Proxy in One Step

```
String mbeanName = "SimpleStandard";

// build the MBean ObjectName instance
ObjectName mbeanObjectName = null;
String domain = connectorClient.getDefaultDomain();
mbeanObjectName = new ObjectName( domain + ":type=" + mbeanName );

// create the MBean in the MBeanServer of the agent
String mbeanClassName = mbeanName;
ObjectInstance mbeanObjectInstance =
    connectorClient.createMBean( mbeanClassName, mbeanObjectName );

// create and bind a proxy MBean on the client side
// that corresponds to the MBean just created in the agent
Proxy mbeanProxy = new SimpleStandardProxy(
                              mbeanObjectInstance, connectorClient );
```

EXAMPLE 24–2 Instantiating and Binding a Legacy Proxy in One Step *(Continued)*

```
echo("\tPROXY CLASS NAME  = " +
      mbeanProxy.getClass().getName() );
echo("\tMBEAN OBJECT NAME = " +
      mbeanProxy.getMBeanObjectInstance().getObjectName() );
echo("\tCONNECTOR CLIENT  = " +
      mbeanProxy.getServer().getClass().getName() );
```

If the class name of your proxy is not known at compile time, you will have to instantiate its class dynamically. In the following code, we obtain the proxy class name that corresponds to an MBean, and we call its first constructor. This must be the constructor that takes an `ObjectInstance` identifying the MBean, and we must dynamically build the call to this constructor. We then call the `setServer` method to bind the new proxy instance.

EXAMPLE 24–3 Instantiating and Binding a Legacy Proxy Class Dynamically

```
// Get the class name of the MBean's proxy
Class proxyClass = Class.forName(
    connectorClient.getClassForProxyMBean( mbeanObjectInstance ));

// Find the constructor with takes an ObjectInstance parameter
Class[] signature = new Class[1];
signature[0] = Class.forName("javax.management.ObjectInstance");
Constructor proxyConstr = proxyClass.getConstructor( signature );

// Call the constructor to instantiate the proxy object
Object[] initargs = new Object[1];
initargs[0] = mbeanObjectInstance;
Proxy proxy2 = (Proxy) proxyConstr.newInstance( initargs );

// Bind the proxy
proxy2.setServer( connectorClient );

echo("\tPROXY CLASS NAME  = " +
      proxy2.getClass().getName());
echo("\tMBEAN OBJECT NAME = " +
      proxy2.getMBeanObjectInstance().getObjectName());
echo("\tCONNECTOR CLIENT  = " +
      proxy2.getServer().getClass().getName());

// We no longer need proxy2, so we unbind it
proxy2.setServer( null );
```

Once a proxy is bound, you can access the attributes and operations of its MBean through direct calls to the proxy object, as shown in the following example.

EXAMPLE 24–4 Accessing a Standard MBean Through Its Legacy Proxy

```
try {
    // cast mbeanProxy to SimpleStandardProxy, so we can
    // call its MBean specific methods
    SimpleStandardProxy simpleStandardProxy =
        (SimpleStandardProxy) mbeanProxy;

    [...]

    // Change the "State" attribute
    simpleStandardProxy.setState("new state from client");

    // Get and display the new attribute values
    echo("\tState     = \"" + simpleStandardProxy.getState() + "\"");
    echo("\tNbChanges = " + simpleStandardProxy.getNbChanges());

    // Invoke the "reset" operation
    simpleStandardProxy.reset();
    [...]

    // We are done with the MBean, so we
    // unbind the proxy and unregister the MBean
    simpleStandardProxy.setServer( null );
    connectorClient.unregisterMBean( mbeanObjectName );

} catch (Exception e) {
    echo("\t!!! Error accessing proxy for " +
          mbeanProxy.getMBeanObjectInstance().getObjectName() );
    e.printStackTrace();
}
```

24.2.3 Running the Legacy Standard Proxy Example

The *examplesDir*/legacy/SimpleClients directory contains all of the files for the ClientMBeanProxy application that demonstrates the use of standard MBean proxies.

▼ To Run the Legacy Standard Proxy Example

1. **Compile all files in this directory with the javac command, if you have not done so already.**

   For example, on the Solaris platform with the Korn shell, type:

   ```
   $ cd examplesDir/legacy/SimpleClients/
   $ javac -classpath classpath *.java
   ```

2. **Generate the proxy MBeans classes and compile them. To do this, from the same directory as above, type the following commands:**

   ```
   $ installDir/SUNWjdmk/jdmk5.0/proxygen SimpleStandard
   $ javac -classpath classpath SimpleStandardProxy.java
   ```

3. **Start the base agent in a terminal window with the following command:**

   ```
   $ java -classpath classpath BaseAgent
   ```

   The agent creates the RMI connector server to which the client application will establish a connection, and then it waits for management operations.

4. **Wait for the agent to be completely initialized, then, in another window on the same host, start the management application with the following command:**

   ```
   $ java -classpath classpath ClientMBeanProxy
   ```

5. **Press `Enter` in the manager window to step through the example.**

   As seen in the code examples, the client application instantiates the proxy objects to access the MBean it has created in the base agent.

6. **Press `Enter` one last time to exit the manager application, but leave the base agent running for the next example.**

24.3 Legacy Generic Proxies

Because dynamic MBeans only expose their management at runtime, it is impossible to generate a specific proxy object for them. Instead, we use the `GenericProxy` object which can be bound to any dynamic MBean, and whose generic methods take the name of the attribute or operation being accessed. Therefore, to access a dynamic MBean through generic proxy you call exactly the same methods as those of the `DynamicMBean` interface.

Just as the MBean server's generic methods can access both standard and dynamic MBeans, generic proxies can also be bound to standard MBeans. You lose the specificity and simplicity of a standard proxy, but a generic proxy is always available in any Java dynamic management application, and it never needs regenerating.

The management application in this example shows how generic proxies can be used to access both standard and dynamic MBeans. The application contains the following subroutine that takes the class name of an MBean, creates that MBean in the agent, and instantiates a generic proxy to access the MBean.

In fact, the subroutine instantiates two generic proxies for the MBean, one using the GenericProxy class constructor that also binds the proxy, the other bound in a second, separate call to its setServer method. This demonstrates that it is possible to have two distinct proxy instances coexisting simultaneously for the same MBean.

EXAMPLE 24–5 Accessing Standard and Dynamic MBeans Using Legacy Generic Proxies

```
private void doGenericProxyExample( String mbeanName ) {

    try {
        // build the MBean ObjectName instance
        ObjectName mbeanObjectName = null;
        String domain = connectorClient.getDefaultDomain();
        mbeanObjectName = new ObjectName( domain +
                                          ":type=" + mbeanName);

        // create the MBean in the MBeanServer of the agent
        String mbeanClassName = mbeanName;
        ObjectInstance mbeanObjectInstance =
            connectorClient.createMBean( mbeanClassName, mbeanObjectName );

        // create and bind a generic proxy instance for the MBean
        Proxy proxy = new GenericProxy(
                              mbeanObjectInstance, connectorClient );

        echo("\tPROXY CLASS NAME  = " +
              proxy.getClass().getName());
        echo("\tMBEAN OBJECT NAME = " +
              proxy.getMBeanObjectInstance().getObjectName());
        echo("\tCONNECTOR CLIENT  = " +
              proxy.getServer().getClass().getName());

        // An alternate way is to first instantiate the generic proxy,
        // and then to bind it to the connector client:
        Proxy proxy2 = new GenericProxy( mbeanObjectInstance );
        proxy2.setServer( connectorClient );

        echo("\tPROXY CLASS NAME  = " +
              proxy2.getClass().getName());
        echo("\tMBEAN OBJECT NAME = " +
              proxy2.getMBeanObjectInstance().getObjectName());
        echo("\tCONNECTOR CLIENT  = " +
              proxy2.getServer().getClass().getName());

        // we no longer need proxy2, so we unbind it
        proxy2.setServer(null);

        [...] // Accessing the MBean through its generic proxy (see below)

        // When done with the MBean, we unbind the proxy
        // and unregister the MBean
        //
        proxy.setServer(null);
        connectorClient.unregisterMBean( mbeanObjectName );
```

EXAMPLE 24–5 Accessing Standard and Dynamic MBeans Using Legacy Generic Proxies *(Continued)*

```
    } catch (Exception e) {
        echo("\t!!! Error instantiating or binding proxy for " +
              mbeanName );
        e.printStackTrace();
    }
}
```

The standard and dynamic MBean classes used in this example have exactly the same management interface, and therefore, we can use the same code to access both of them. The manager application does this by calling the above subroutine twice, once with the class name of the standard MBean, once with that of the dynamic MBean:

```
manager.doGenericProxyExample("SimpleStandard");
manager.doGenericProxyExample("SimpleDynamic");
```

Because the two MBeans have the same behavior, they produce the same results when accessed through their proxy. The only difference is that the dynamic MBean can expose a description of its management interface in its `MBeanInfo` object. As expected, accessing a standard MBean through a generic proxy also produces the same result as when it is accessed through a standard proxy (compare the following with Example 24–5).

EXAMPLE 24–6 Accessing an MBean Through its Legacy Generic Proxy

```
try {

    // cast Proxy to GenericProxy
    GenericProxy genericProxy = (GenericProxy) proxy;

    // Get the MBean's metadata through the proxy
    MBeanInfo info = genericProxy.getMBeanInfo();

    // display content of the MBeanInfo object
    echo("\nCLASSNAME: \t"+ info.getClassName() );
    echo("\nDESCRIPTION: \t"+ info.getDescription() );
    [...] // extract all attribute and operation info

    // Change the "State" attribute
    Attribute stateAttr = new Attribute( "State", "new state from client");
    genericProxy.setAttribute( stateAttr );

    // Get and display the new attribute values
    String state =
        (String) genericProxy.getAttribute("State");
    Integer nbChanges =
        (Integer) genericProxy.getAttribute("NbChanges");
    echo("\tState     = \"" + state + "\"");
    echo("\tNbChanges = " + nbChanges);
    // Invoke the "reset" operation
    Object[] params = new Object[0];
    String[] signature = new String[0];
    genericProxy.invoke("reset", params, signature );

} catch (Exception e) {
    echo("\t!!! Error accessing proxy for " +
          proxy.getMBeanObjectInstance().getObjectName() );
    e.printStackTrace();
}
```

Example 24–6 shows how the generic methods are called with the names of attributes and operations, and how required parameters can be constructed.

24.3.1 Running the Legacy Generic Proxy Example

The `ClientGenericProxy` application, also located in the *examplesDir*`/legacy/SimpleClients` directory, demonstrates the use of generic proxies.

▼ To Run the Generic Proxy Example

1. **Compile all files in this directory with the `javac` command, if you have not done so already. For example, on the Solaris platform with the Korn shell, type:**

   ```
   $ cd examplesDir/legacy/SimpleClients/
   $ javac -classpath classpath *.java
   ```

   Because generic proxies do not need to be generated, this example does not need the `proxygen` tool. The `GenericProxy` class is available in the usual classpath for the product's runtime libraries.

2. **If it is not already running on your host, start the base agent in a terminal window with the following command:**

   ```
   $ java -classpath classpath BaseAgent
   ```

   The agent creates the RMI connector server to which the client application will establish a connection, and then it waits for management operations.

3. **Wait for the agent to be completely initialized, then start the management application in another window on the same host:**

   ```
   $ java -classpath classpath ClientGenericProxy
   ```

4. **Press `Enter` in the manager window to step through the example.**

As seen in the code examples, the client application instantiates generic proxy objects to access both a standard and dynamic MBean that it creates in the base agent. The only difference between the two is the user-provided information available in the dynamic MBean's metadata.

5. **Press `Enter` in both windows to exit the base agent and manager applications.**

24.4 Legacy Proxies for Java DMK Components

Most components of the Java DMK product are MBeans and can therefore also be managed through local or remote proxies. Nearly all are standard MBeans, so their corresponding standard proxies are provided with the product. The Java source code for all component proxy classes can be found in the `/legacy/JdmkProxyMBeans` directory located in the main *examplesDir* (see "Directories and Classpath" in the Preface).

Note – The HTML protocol adaptor is implemented as a dynamic MBean and therefore cannot have a standard proxy. You must use a generic proxy to access the HTML adaptor through a proxy object.

Of course, all other Java DMK MBean components can also be accessed through generic proxies, although their standard proxies provide more abstraction of the MBean server and a greater simplification of your application's code.

The proxy classes have been generated by the `proxygen` tool with full read-write access of all attributes. Refer to the chapter on the `proxygen` tool in the *Java Dynamic Management Kit 5.1 Tools Reference Guide*.

24.4.1 Legacy Proxy Packages

Like all other classes, proxies can contain a `package` statement. The package for a component proxy class depends upon the package of the component:

- The proxy classes for Java DMK components in the `javax.management` package and its `javax.management.*` subpackages do not have a package statement.

  Their class files can be located in any directory that can then be added to your application's classpath.

- The proxy classes for all other components belong to the same class as the component itself. For example, the proxy classes for the RmiConnectorServer component are declared in the com.sun.jdmk.comm package.

  Their class files are contained in the corresponding file hierarchy, whose root can be added to your application's classpath. Therefore, the class files of the RmiConnectorServerProxy are located in a directory called *packageRoot*/com/sun/jdmk/comm/.

24.4.2 Compiling the LegacyProxy Classes

To use the standard proxies for the product components, you must first compile them with the javac command and then place the classes you need in the classpath of your agent or management application. If you compile the proxy classes individually, be sure to compile the proxy's MBean interface before its corresponding proxy class.

Because of the package statement in proxy classes, use the following commands:

```
$ cd examplesDir/legacy/JdmkProxyMBeans/
$ javac -d packageRoot -classpath classpath *ProxyMBean.java *Proxy.java
```

In this command, the *classpath* must contain the current directory and the classpath of the Java DMK runtime libraries (jdmkrt.jar and legacysnmp.jar). See "Directories and Classpath" in the Preface). The -d option of the javac compiler creates the necessary directories in the given *packageRoot* for each class's package. The *packageRoot* is usually the current directory (.) or you can specify a target directory in your application's classpath directly.

CHAPTER 25

Legacy Cascading Agents

Due to the implementation of Java Management Extensions (JMX) Remote API in Java Dynamic Management Kit (Java DMK) 5.1, the cascading service has been superseded by a new implementation of cascading. The cascading service used with the legacy connectors is retained here for reasons of backwards compatibility.

The code samples in this topic are from the files in the `legacy/Cascading` directory located in the main *examplesDir* (see *Directories and Classpath* in the preface).

This chapter contains the following topics:

- "25.1 Legacy `CascadingAgent` MBean" on page 485 describes the main component of the cascading service that creates the connections between two agents.
- "25.2 Mirror MBeans in the Legacy Cascading Service Master Agent" on page 488 gives more details about what you can and cannot do through the cascading service.
- "25.3 Running the Legacy Cascading Example" on page 491 lets you interact with the two cascading agents through their HTML protocol adaptors.

25.1 Legacy `CascadingAgent` MBean

You should create one `CascadingAgent` MBean for every subagent you want to manage through the master agent. Each connects to an agent and mirrors all of that agent's registered MBeans in the master agent's MBean server. No other classes are required or need to be generated in order to represent the MBeans.

The agent whose MBean server contains an active cascading service is called a *master agent* in relation to the other agent that is mirrored. The agent to which the cascading service is connected is called the *subagent* in relation to its master agent. We say that it creates *mirror MBeans* to represent the subagent's MBeans. See "25.2 Mirror MBeans in the Legacy Cascading Service Master Agent" on page 488 for a description of these objects.

A master agent can have any number of subagents, each controlled individually by a different `CascadingAgent` MBean. A subagent can itself contain cascading agents and mirror MBeans, all of which are mirrored again in the master agent. This effectively enables cascading hierarchies of arbitrary depth and width.

Two master agents can also connect to the same subagent. This is similar to the situation where two managers connect to the same agent and can access the same MBean. If the implementation of a management solution permits such a condition, it is the designer's responsibility to handle any synchronization issues in the MBean.

The connection between two agents resembles the connection between a manager and an agent. The cascading service MBean relies on a connector client, and the subagent must have the corresponding connector server. The subagent's connector server must already be instantiated, registered with its MBean server, and ready to receive connections.

In our example application, we use the legacy RMI connector client that we will connect to the legacy RMI connector server of the subagent on port 1099 of the local host. In fact, this is the same as the default values when instantiating a cascading agent MBean, but we also want to specify a pattern for selecting the MBeans to mirror. By default, all MBeans of the subagent are mirrored in the master agent; we provide an object name pattern to only select those in the subagent's `CascadedDomain`.

EXAMPLE 25–1 Connecting to a Subagent

```
ObjectName mbeanObjectName = null;
String domain = server.getDefaultDomain();
mbeanObjectName = new ObjectName(domain + ":type=CascadingAgent");
[...]

RmiConnectorAddress address = new RmiConnectorAddress(
    java.net.InetAddress.getLocalHost().getHostName(),
    1099,
    "name=RmiConnectorServer");
CascadingAgent remAgent = new CascadingAgent(
    address,
    "com.sun.jdmk.comm.RmiConnectorClient",
    new ObjectName("CascadedDomain:*"),
    null);
ObjectInstance remAgentInstance =
    server.registerMBean(remAgent, mbeanObjectName);

[...] // Output omitted
// Now we explicitly start the cascading agent
// as it is not started automatically
```

EXAMPLE 25–1 Connecting to a Subagent *(Continued)*

```
//
echo("\nStarting the cascading agent...");
[...]
server.invoke(mbeanObjectName, "start", null, null);
sleep(1000);
echo("\tIs ACTIVE = " + server.getAttribute(mbeanObjectName, "Active"));
echo("done");
```

Before the subagent’s MBeans are mirrored, the `CascadingAgent` MBean must be registered in the master agent’s MBean server, and its mirroring must be started. When you invoke the `start` operation of the legacy cascading service MBean, it will connect it to its designated subagent and create one mirror MBean to represent each MBean in the subagent. When its `Active` attribute becomes true, the cascading mechanism is ready to use.

The `CascadingAgent` MBean exposes two writable attributes:

- `Address` is a `ConnectorAddress` object whose subclass defines the protocol used for the connection and whose content gives the address or host name of the subagent, along with a port number.
- `ClientConnectorClassName` is a string that gives the class name of the connector client to be used in the connection; it must be compatible with the protocol defined by the class of the `Address` attribute.

Neither of these attributes can be modified when the legacy cascading service is active. You must first call the MBean’s `stop` operation: this will remove all of the mirror MBeans for the given subagent and disconnect from the subagent’s connector server. You can then modify the address or the class of the connector client. The new values will be used when you start the mirroring again: this lets you change subagents or even change protocols.

When the legacy cascading service is stopped or its MBean is removed from the master agent, all of its mirror MBeans are unregistered. The MBean server delegate in the master agent will send an unregistration notification for each mirror MBean as it is removed.

25.2 Mirror MBeans in the Legacy Cascading Service Master Agent

Once the legacy cascading service is active, you interact directly with the mirror MBeans representing the subagent's MBeans. You can access and manage a mirror MBean as if you are connected to the subagent and accessing or managing the original MBean. The mirror MBeans are actual MBean objects registered in the master agent's MBean server with the same object name.

All management operations that you can perform on the original MBean can be performed identically on its mirror MBean. You can modify attributes, invoke operations and add or remove listeners, all with exactly the same result as if the manager were connected to the subagent when performing the action.

The behavior of a mirror MBean is to transmit the action to the subagent's MBean and return with an answer or result. The actual computation is performed by the original MBean running in its own agent.

In our example, we know that there is a timer MBean that was created in the subagent. Once the cascading service is active for our subagent, we operate the timer through its local mirror MBean. We can never have the direct reference to a mirror MBean, so we always invoke operations through the master agent's MBean server.

EXAMPLE 25–2 Managing Mirrored MBeans

```
// Here we know the object name of the MBean we want to access, the
// object name of its mirrored MBean in the master agent will be identical.
ObjectName timerName = new ObjectName("CascadedDomain:type=timer");

echo("\n>>>  Ask the Timer MBean to send a notification every 5 seconds ");
java.util.Date currentDate = new java.util.Date();
Object params[] = {
    "Timer",
    "Message",
    new Integer(5),
    new java.util.Date (currentDate.getTime() + new Long (2).longValue()),
    new Long(1000) };

String signatures[]={ "java.lang.String",
                      "java.lang.String",
                      "java.lang.Object",
                      "java.util.Date",
                      "long"};

server.invoke(timerName, "addNotification", params, signatures);
server.invoke(timerName, "start", null, null);

echo("\n>>>  Add ourselves as a listener to the Timer MBean");
server.addNotificationListener(timerName, this, null, null);
```

EXAMPLE 25–2 Managing Mirrored MBeans *(Continued)*

```
echo("\nPress <Enter> to remove the listener from the Timer MBean ");
waitForEnterPressed();
server.removeNotificationListener(timerName, this);
```

For the managing application, the mirror MBean in the master agent *is* the MBean. Unregistering a mirror MBean in the master agent will unregister the mirrored MBean in the subagent. If you want to control the number of mirror objects without removing the originals, you must use filters and/or queries of the subagent's MBeans in the constructor of the legacy cascading service MBean.

The mirror and its mechanism make the cascading totally transparent: a manager has no direct way of knowing whether an object is a mirror or not. Neither does it have any direct information about the topology of the cascading hierarchy rooted at the agent that it accesses. If this information is necessary, the MBeans should expose some kind of identification through their attributes, operations, or object names.

25.2.1 Class of a Mirror MBean

Mirror MBeans are implemented as dynamic MBeans; they are instances of the `CascadeGenericProxy` class. The cascading service gives them the `MBeanInfo` object that they will expose and establishes their connection with the original MBean. The MBean information contains the class name of the original MBean, not their own class name. Exposing this borrowed class name guarantees that the cascading service is completely transparent.

The symmetry of the Java dynamic management architecture means that this cascading mechanism is scalable to any number of levels. The mirror object of a mirror object is again an instance of the `CascadeGenericProxy` class, and it borrows the same object name and class name. Any operation on the top mirror will be propagated to its subagent, where the intermediate mirror will send it its own subagent, and so forth. The cost of cascading is the cost of accessing the subagent: the depth of your cascading hierarchy should be adapted to your management solution.

Because the legacy cascading service MBean instantiates and controls all mirror MBeans, the `CascadeGenericProxy` class should never be instantiated through a management operation, nor by the code of the agent application. We have described it here only to provide an example application of dynamic MBeans.

25.2.2 Legacy Cascading Service Issues

In this section, we explain some of the design issues that are determined by the implementation of the legacy cascading service.

25.2.2.1 Dynamic Mirroring

Any changes in the subagent's MBeans are automatically applied to the set of mirror MBeans, to ensure that both master agent and subagent remain consistent.

When an MBean is unregistered from the subagent, the cascading service MBean removes its mirror MBean from the master agent. When a new MBean is registered in the subagent, the cascading service instantiates its mirror MBean, sets up its connection to the new MBean, and registers the mirror with the master agent's MBean server.

Both of these mechanisms scale to cascading hierarchies: adding or removing an MBean in the master agent will trigger a notification that any cascading service connected to the master agent will receive. This will start a chain reaction up to the top of the hierarchy. Removing an MBean from the middle of a hierarchy also triggers a similar reaction down to the original MBean that is finally removed.

Dynamic unregistration only applies to subagent MBeans that are actually mirrored. Dynamic registration is also subject to filtering, as described in the next section.

25.2.2.2 MBean Filtering

When the legacy cascading service MBean is instantiated, you can pass it an object name pattern and a query expression. These will be applied to the list of MBeans in the subagent to determine those that will be mirrored in the master agent. The filtering will be in effect for the life of the cascading service connected to this MBean.

Filtering the mirrored MBeans reduces the number of MBeans in the master agent. It also provides a way of identifying mirror MBeans in the master agent, as in our example where cascading is limited to MBeans in the `CascadedDomain`.

Both the object name pattern and query expression will be used to filter any new MBean that is registered in the subagent. If the new MBean meets the filter criteria, it will become visible and mirrored in the master agent. Since the query expression applies to attribute values of the MBean, you must be careful to initialize the new MBean before registering it so that its values are significant when the filter is applied by the cascading service.

25.2.2.3 Naming in Cascading Agents

Mirror MBeans are registered in the master agent with the same object name as the mirrored MBean in the subagent. If the registration fails in the master agent's MBean server, no error is raised and no action is taken: the corresponding MBean will simply not be mirrored in the master agent.

The most likely cause for registration to fail is that the object name already exists in the master agent. An MBean cannot be registered if its chosen object name already exists in the MBean server.

If your management solution has potential naming conflicts, you will need a design that is guaranteed to assign unique object names throughout the cascade hierarchy. You can set the default domain name in your subagents or use the `MBeanServerId` attribute of the delegate MBean to give MBeans a unique object name.

25.3 Running the Legacy Cascading Example

The *examplesDir*`/legacy/Cascading` directory contains all of the files for the two agent applications, along with a simple MBean.

▼ To Run the Legacy Cascading Example

1. **Compile all files in this directory with the `javac` command. For example, on the Solaris platform with the Korn shell, you would type:**

   ```
   $ cd examplesDir/legacy/Cascading/
   $ javac -classpath classpath *.java
   ```

2. **Start the subagent in another terminal window with the following command. Be sure that the classes for the `SimpleStandard` MBean can be found in its *classpath*.**

   ```
   $ java -classpath classpath SubAgent
   ```

3. **Wait for the agent to be completely initialized, then start the master agent with the following command:**

   ```
   $ java -classpath classpath MasterAgent
   ```

   When started, the master agent application first creates the `CascadingAgent` MBean and then sets up its connection to the subagent. The master agent then performs operations on the mirrored MBeans of the subagent. Press `Enter` to step through the example when the application pauses.

4. **You can also interact with the example through the HTML adaptor of the master agent and subagent. If you are still receiving timer notification on the master agent, press `Enter` once more to remove the listener, but leave both agent applications running.**

▼ How to Interact with a Legacy Cascade Hierarchy

1. **Open two browser windows side by side and load the following URLs:**

Subagent	`http://localhost:8082/`
Master Agent	`http://localhost:8084/`

In the subagent, you should see the timer MBean in the `CascadedDomain` and a `SimpleStandard` MBean in the `DefaultDomain`.

The master agent is recognizable by the cascading service MBean in the `DefaultDomain`. Otherwise it has an identical timer MBean registered in the `CascadedDomain`: this is the mirror for the timer in the subagent. The `SimpleStandard` MBean is not mirrored because our cascading service instance filters with the following object name pattern:

```
CascadedDomain:*
```

2. **Create four MBeans of the `SimpleStandard` class in following order:**

On the Master Agent:	`CascadedDomain:name=SimpleStandard,number=1`
On the Subagent:	`CascadedDomain:name=SimpleStandard,number=1`
	`CascadedDomain:name=SimpleStandard,number=2`
	`CascadedDomain:name=SimpleStandard,number=3`

3. **Reload the agent view on the master agent.**

   The mirror MBeans for the last two agents have been created automatically. Look at the MBean view of either of these mirror MBeans on the master agent. Their class name appears as `SimpleStandard`.

4. **In the master agent, set a new value for the `State` string attribute of all 3 of its `SimpleStandard` MBeans.**

   When you look at the corresponding MBeans in the subagent, you see that `number=2` and `number=3` were updated by their mirror MBean. However, `number=1` has not changed on the subagent. Because it was created first in the master agent, it is not mirrored and exists separately on each agent.

5. **In the subagent, invoke the `reset` operation of all 3 of its `SimpleStandard` MBeans.**

   When you inspect the MBeans in the master agent, the values for `number=2` and `number=3` were reset. Remember that the HTML adaptor must get the values of attributes for displaying them, so they were correctly retrieved from the mirrored MBeans that we reset.

6. **In the master agent, unregister MBeans `number=1` and `number=2`, then update the agent view of the subagent.**

   In the subagent, you should still see the local version of `number=1`, but `number=2` has been removed at the same time as its mirror MBean.

We are in a state where `number=1` is a valid MBean for mirroring but it is not currently mirrored. This incoherence results from the fact that we did not have unique object names throughout the cascading hierarchy. Only new MBeans are mirrored dynamically, following the notification that signals their creation. We would have to stop and restart the master agent's cascading service MBean to mirror `number=1`.

7. **In the subagent, unregister MBeans `number=1` and `number=3`, then update the agent view on the master agent.**

   The mirror MBean for `number=3` was automatically removed by the cascading service, so none of the MBeans we added now remain.

8. **Invoke the `stop` operation of the `CascadingAgent` MBean in the master agent.**

   The last mirror MBean for the timer is removed from the master agent. The two agents are no longer connected.

9. **If you have finished with the agents, press `Enter` in both of their terminal windows to exit the applications.**

Index