Java Dynamic Management Kit 5.1
Getting Started Guide

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

Part No: 816–7607
June 4th, 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.

Federal Acquisitions: Commercial Software–Government Users Subject to Standard License Terms and Conditions.

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

040518@8606

Contents

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.

The *Java Dynamic Management Kit 5.1 Getting Started Guide* presents the architecture of the Java DMK, introducing the key components of the product and the development process for management applications.

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 are now compatible with the latest JMX 1.2 Maintenance Release.
- Secure and interoperable remote access is now compatible with the new JMX Remote API 1.0 Specification, including support for both the RMI-based and JMXMP-based standard connectors.
- Flexible authentication and privacy based on the Simple Authentication and Security Layer (SASL) 1.1 Specification and TLS.

- SASL mechanisms providing authentication, namely SASL-PLAIN, DIGEST-MD5, CRAM-MD5, and GSSAPI/Kerberos.
- SASL mechanisms providing connection privacy, namely DIGEST-MD5, GSSAPI/Kerberos.
- Fine-grained access control based on an authenticated client.
- 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.
- Enhanced Cascading service, supporting both the JMX Remote API connectors and the legacy Java DMK connectors.
- Enhanced Discovery service, allowing the discovery of Java DMK based applications using legacy connectors as well as applications using the new connectors.

Who Should Use This Book

This book is aimed at anyone who requires an introduction to the concepts and components of Java DMK.

You should be familiar with Java programming and the JavaBeans™ component model. You should also be familiar with the JMX specification, the JMX Remote API specification, and the Simple Network Management Protocol (SNMP).

This book is not intended to be an exhaustive reference. For more information about each of the management levels and how they interact, see the *Java Dynamic Management Kit 5.1 Tutorial*, and the API documentation generated by the Javadoc™ tool and included in the online documentation package.

After understanding of the concepts of the Java DMK, you should familiarize yourself with the tools for developing management applications. Then, through the lessons of the *Java Dynamic Management Kit 5.1 Tutorial*, learn how to instrument new or existing resources, write intelligent agent applications, and access these applications from remote managers written in the Java programming language. You can then design and develop your own Java dynamic management solution.

How This Book Is Organized

This book explains the key concepts of Java DMK, introduces the main components of the product, provides an overview of the development process and outlines the tools you need to use Java DMK. It is divided into the following chapters:

- Chapter 1 *"Java Dynamic Management Kit Overview"*
- Chapter 2 *"Architectural Components"*
- Chapter 3 *"The Development Process"*

Before You Read This Book

To build and run the sample programs or use the tool commands provided in Java DMK, you must have a complete installation of the product on your machine. Refer to the *Java Dynamic Management Kit 5.1 Installation README* for instructions on how to install the product components and configure your environment.

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

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	`#`

CHAPTER 1

Overview of the Java Dynamic Management Kit

The Java Dynamic Management Kit (Java DMK) is a Java application programming interface (API) and a set of development tools for designing and implementing a new generation of management applications. As an implementation of Java Management Extensions (JMX), the product provides a framework for the management of Java objects through Java technology-based applications. Because the product is also an implementation of the JMX Remote API, it allows your management applications to monitor and manage resources through your network.

The Java DMK provides a complete architecture for designing distributed management systems. A Java technology-based solution can embed management intelligence into your agents, can provide an abstraction of your communication layer, and can be upgraded and extended dynamically. Your management applications can also take advantage of other Java APIs such as Swing components for user interfaces and the JDBC™ API for database access.

In addition, the Java DMK provides a complete toolkit for the simple network management protocol (SNMP), the most widespread legacy architecture for network and device management. This gives you the advantages of developing both Java dynamic management agents and managers that can interoperate with existing management systems.

This chapter contains the following sections:

- "1.1 Introduction to the Java DMK" on page 16 gives an overview of the product architecture and functionality.
- "1.2 Key Concepts" on page 22 describes the main components of the Java DMK.
- "1.3 Benefits of a Java Dynamic Management Solution" on page 23 highlights the benefits of the product for designers and developers.
- "1.4 Overview of the Product Documentation" on page 26 describes the product documentation delivered with the Java DMK.

1.1 Introduction to the Java DMK

This section addresses these fundamental questions about the Java DMK:

- Why use Java dynamic management technology?
- What is the Java Dynamic Management Kit?
- How is a Java dynamic management solution developed?

If this is your first contact with the product, the answers to these questions should help you understand how your management needs can be solved using Java dynamic management technology.

1.1.1 Why Use Java Dynamic Management Technology?

In the past, network management was usually performed by large, centralized management applications. These management applications monitored and modified their network by tightly controlling their agents. In addition, agents were usually situated in or near the network elements they controlled, which meant that these agents were limited in nature. The agents usually contained little management intelligence and could perform only basic network management operations.

A Java dynamic management agent exposes its resources in a standard way and provides management services directly at the resource level. These services provide the intelligence that enables agent applications to perform management tasks autonomously. This frees the management application from routine tasks such as polling and thus reduces the network load as well.

When you implement Java dynamic management technology, the interface to resources is standardized, meaning your management applications can use any technology you want. As long as management applications communicate through a Java dynamic management agent, they can access any resource.

The same flexibility applies to the management services that are deployed in the agents. Because the management services can control resources through standard interfaces, they are dynamically interchangeable. When new services become available, these services can be downloaded and be plugged in dynamically to upgrade the capabilities of a smart agent. Finally, the Java DMK provides a distributed model that is protocol independent. Management applications rely on the API, not on any one protocol.

The Java DMK brings new solutions to the management domain through the following advantages.

- Compliance with the JMX specification and the JMX Remote API specification, for managing Java objects through Java applications, as developed through the Java Community Process^{SM}(JCP^{SM}).
- A single suite of components that provides uniform instrumentation for managing systems, applications, and networks, and that provides universal access to these resources.
- A flexible architecture that distributes the management load. This architecture can also be upgraded in real time for the service-driven network.

The service-driven network is a new approach to network computing that concentrates on the services you want to provide. These range from the low-level services that manage relationships between network devices to the value-added services you provide to end users. These services *drive* your network and management needs. In addition, autonomous agent functionality makes it possible to manage a very large installed base.

With the Java dynamic management architecture, services can be incorporated directly into agents. Agents are given the intelligence to perform management tasks themselves, enabling management logic to be distributed throughout the whole network. New services can be downloaded from a web server at runtime using a dynamic pull mechanism. Services are not only implemented inside devices, but they can also be network-based. If your services are network-based, you can download them through simple web pages in the same way as Java technology-based applets.

You can connect to your agents remotely, using the connector protocols that were standardized in JMX Remote API. Using the remote method invocation (RMI) and JMX messaging protocol (JMXMP) connectors, you can access agents across a network with the connectors remaining completely invisible to either end of the connection. These connections can be secured using the Secure Sockets Layer (SSL) security mechanism with the RMI connectors, and with the more advanced Simple Authentication and Security Layer (SASL) protocol with the JMXMP connector.

This dynamic, on-demand paradigm means that it is no longer necessary to know what will need to be configured, managed, and monitored in the future or in advance of network deployment. Services are created, enhanced and deployed as needed. This unique combination of features gives the Java DMK a wide domain of application as it integrates the current and future management standards.

1.1.2 What Is the Java DMK?

The Java DMK is a Java API that includes all its class and interface objects, development tools that speed up the development process, and a complete set of documentation. The Java DMK is a compliant implementation of the following specifications:

- The JMX specification
- The JMX Remote API specification

The programmatic components of the Java DMK include the following.

- New standard communication modules – Version 5.1 of Java DMK defines APIs for accessing JMX agents remotely. This version includes new standard communication modules that based on RMI and JMXMP protocols, as defined by the JMX Remote API specification. The JMXMP connector is a custom connector that has been created especially for the JMX Remote API. JMXMP is based on Java serialization over transmission control protocol (TCP) connections.
- RMI, hypertext transport protocol (HTTP), and secure HTTP (HTTPS) communication modules. You can still use the *legacy* communication modules based on the RMI, HTTP, and HTTPS protocols that were included in previous versions of the product. These legacy communication modules are deprecated in Java DMK 5.1.
- HTML adaptor. The Java DMK includes an HTML adaptor, which supports access to an agent from a web browser.
- Agent services. The library of supplied services includes monitoring, scheduling, dynamic loading, defining relations, new and legacy systems for cascading agent hierarchies, dynamic agent discovery, and components for implementing security mechanisms.
- SNMP API. Applications that rely on the SNMP APIs can integrate into existing network management systems and can help these systems migrate towards a more dynamic, service-based approach to network management.
- SNMPv3 compliance. Java DMK 5.1 provides an implementation of SNMPv3 security to protect your systems from outside interference.
- Security mechanisms. Java DMK 5.1 allows you to choose the level of security you require. For example, for an RMI connector over JRMP, you can use an RMI socket factory, so that the connection between client and server uses the Secure Socket Layer (SSL). A more advanced level of security is available with the standard JMXMP connector, which is based on the Java Secure Socket Extension (JSSE), the Java Authentication and Authorization Service (JAAS), and the Simple Authentication and Security Layer (SASL).

The development tools are implemented as two standalone applications:

- `mibgen` – This tool is used when developing SNMP agents. A management information base (MIB) represents the management interface of resources in an SNMP agent, and `mibgen` generates the corresponding Java objects.
- `proxygen` – This tool is a proxy object generator for use with legacy connectors. The `proxygen` tool simplifies the development of Java technology-based management applications. Proxy objects make the communication layer transparent to the manager application. Note that if you require proxies for standard connectors, you should use the dynamic proxies provided by J2SE (`java.lang.reflect.Proxy`), not the `proxygen` tool. The `proxygen` tool is deprecated in Java DMK 5.1.

Finally, the Java DMK includes complete documentation for developers:

- The full description of all classes, interfaces and methods in the APIs, generated by the Javadoc utility.
- The source code for programming examples, which demonstrate various aspects of the functionality of the Java DMK.
- A tutorial that explains the programming examples and a reference guide for the standalone tools.
- Both online HTML and PDF file formats for all documents. The HTML format complies with the accessibility standards for electronic and information technology covered by section 508 of the Rehabilitation Act Amendments of 1998.

1.1.3 How is a Java Dynamic Management Solution Developed?

The instrumentation level of the JMX specification describes how to represent a resource as a Java object. The JMX agent level describes how resources interact with an agent. The management level defined by the JMX Remote API specification describes how to use standard connectors to access agents remotely, and how to implement the associated security aspects. Using the Java DMK, you can design and develop a distributed management solution relying on all three levels, and compliant with both specifications.

1.1.3.1 Instrument Your Resources as MBeans

A resource can be any entity, physical or virtual, that you want to make available and control through your network. Physical resources can be devices such as network elements or printers. Virtual resources include applications and computational power that are available on some host. A resource is seen through its *management interface*, that is, the set of attributes, operations, and notifications that a management application can access.

To instrument a resource is to develop the Java object that represents the resource's management interface. The JMX specification defines how to instrument a resource according to a certain design pattern. These patterns resemble those of the JavaBeans™ component model. An attribute has getters and setters, operations are represented by their Java methods, and notifications rely on the Java event model.

A *managed bean*, or *MBean*, is the instrumentation of a resource in compliance with the JMX design patterns. If the resource itself is a Java application, it can be its own MBean. Otherwise, an MBean is a Java wrapper for native resources or a Java representation of a device. MBeans can be distant from the managed resource, as long as they accurately represent its attributes and operations. The MBean developer determines what attributes and operations are available through the MBean.

Device manufacturers and application vendors can provide the MBeans that plug into their customer's existing agents. Management solution integrators can develop the MBeans for resources that have not been previously instrumented. Because MBeans follow the JMX specification, they can be instantiated in any agent that is compliant with the JMX specification. This compliance makes the MBeans portable and independent of any proprietary management architecture.

1.1.3.2 Expose Your MBeans in a Smart Agent

A Java dynamic management agent follows the client-server model. The agent responds to the management requests from any number of client applications that want to access the resources that the agent contains. The agent centralizes all requests, dispatches the requests to the target MBeans, and returns any responses. The agent, rather than the MBeans, handles the communication issues involved with receiving and sending data.

The central component of an agent is the *MBean server*. The MBean server is a registry for MBean instances, that exposes a generic interface through which clients can issue requests on specific MBeans. Clients can ask for the description of an MBean's management interface, to find out what resource is exposed through that MBean. Using this information, the manager can then formulate a request to the MBean server to get or set attributes, invoke operations, or register for notifications.

MBeans are accessible only through requests to the MBean server. Manager applications never have the direct reference of an MBean, only a symbolic object name which identifies the MBean in the agent. This preserves the client-server model and is essential to the implementation of query and security features.

The MBean server also provides the framework that allows *agent services* to interact with MBeans. Services are themselves implemented as MBeans, which interact with resource MBeans to perform some task. For example, a manager could decide to monitor an MBean attribute. The manager instantiates the monitoring service MBean, configures the threshold, and registers to receive the alarms that might occur. The manager no longer needs to poll the agent, but will automatically be notified whenever the attribute exceeds the threshold.

The library of services contains the logic that is necessary for implementing advanced management policies, such as the following.

- Scheduling events
- Monitoring attributes
- Establishing and enforcing relations
- Discovering other agents
- Creating subagent hierarchies
- Downloading of new MBean objects

You can also develop your own service MBeans to meet your management needs, such as logging and persistence services, which are typically platform-dependent.

1.1.3.3 Access Your Agents Remotely

Finally, the Java DMK enables you to access agents and their resources easily from a *remote* application. All components for handling the communication are provided, both in the agent and for the client application. The same API that is exposed by the MBean server in the agent is also available remotely to the manager. This symmetry effectively makes the communication layer transparent.

Management applications perform requests by getting or setting attributes or invoking operations on an MBean identified by its symbolic name. Proxy objects provide a further level of abstraction by representing an MBean remotely and handling all communication. The manager can be designed and developed as if all resources were local. The communication components also handle notification forwarding, so that remote managers can register to receive notifications from broadcasting MBeans.

Management applications developed in the Java programming language use *connectors* to make the communication layer transparent. Connectors for the RMI, RMI/IIOP and JMXMP protocols are provided, as defined by the JMX Remote API, all with the same API for interchangeability. The legacy RMI and HTTP-based connectors from previous versions of Java DMK are retained for reasons of backwards compatibility, but are deprecated in version 5.1. Wherever possible, you should migrate your remote agents to the standard JMX Remote API connectors.

Adaptors provide a view of an agent through other protocols for management applications which are not based on Java technology. For example, the HTML adaptor represents MBeans as web pages that can be viewed in any web browser. The SNMP adaptor can expose special MBeans that represent an SNMP MIB and respond to requests in the SNMP protocols. It is possible to use the SNMP adaptor without registering the MIB in the MBean server.

All connectors and adaptors are implemented as MBeans. Management applications can therefore create, configure and remove communication resources dynamically, according to network conditions or available protocols. Each protocol can have its own built-in security mechanisms, for example SSL, SASL, or SNMPv3 security. Security aspects linked to each protocol are therefore handled at the connector or adaptor layer, making them transparent to the MBean developer.

The flexibility of communicator MBeans and the availability of connectors for multiple protocols make it possible to deploy management solutions in heterogeneous network environments. The adaptors create a bridge between agents that are based on the JMX architecture and existing management systems. You can also create your own connectors and adaptors to accommodate proprietary protocols and future management needs.

1.2 Key Concepts

Figure 1–1 illustrates the key concepts of the Java DMK and shows how the components relate to each other.

In this example, the MBeans for two resources are registered with the agent's MBean server. An agent service such as monitoring is registered as another MBean. The agent contains a connector server for one of either the RMI or JMXMP connector protocols. The agent also contains a protocol adaptor, either for SNMP or HTML. An agent can have any number of communicator components, one for each of the protocols, and one for each of the ports through which it communicates.

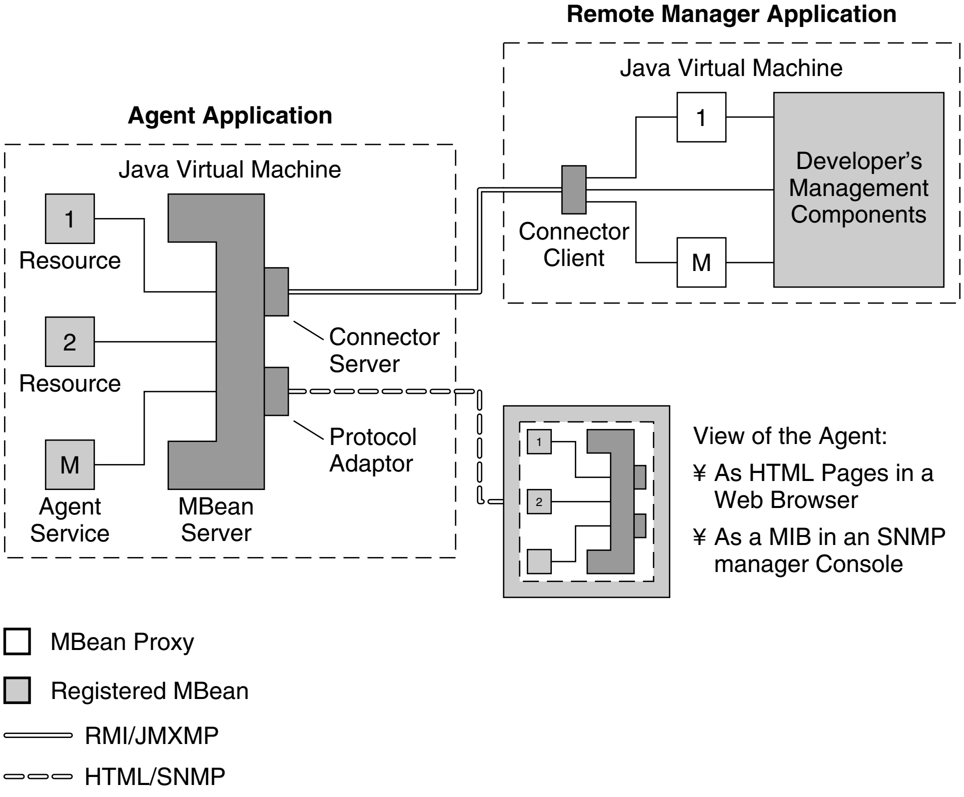

FIGURE 1–1 Key Concepts of the Java DMK

The remote manager is a Java application running on a distant host. The manager contains the connector client for the chosen protocol and proxy MBeans representing the two resources. When the connector client establishes the connection with the agent's connector server, the other components of the application can issue management requests to the agent. For example, the connector client can call the proxy objects to invoke an operation on the first resource and configure the monitoring service to poll the second resource.

With the HTML adaptor, you can view the agent through a web browser, which provides a simple user interface. Each MBean is represented as a separate HTML page, from which you can interact with text fields to set attributes and click buttons to invoke operations. The HTML adaptor also provides an administration page for creating or removing MBeans from the MBean server.

Each of these concepts is further defined in Chapter 2.

1.3 Benefits of a Java Dynamic Management Solution

To summarize, the benefits of the Java DMK include the following.

- Simplified design and development of instrumentation, smart agents, and remote managers
- Deployment flexibility through protocol independence and SNMP compatibility
- Dynamic extensibility and scalability
- Secure SNMPv3 access
- Secure standard communication modules with remote managers

1.3.1 Simplified Design and Development

The JMX architecture standardizes the elements of a management system. All three levels, instrumentation, agent, and manager, are isolated and their interaction is defined through the API. This design makes it possible to have modular development, in which each level is designed and implemented independently. Also, component reuse is possible. Services developed for one JMX agent will work in all JMX agents.

At the instrumentation level:

- MBeans need only to define their management interface and map the variables and methods of their resource to the attributes and operations of the interface.
- MBeans can be instantiated into any agent that is compliant with the JMX specification.
- MBeans do not need to know anything about communication with the outside world.

At the agent level:

- The MBean server handles the task of registering MBeans and transmitting management requests to the designated MBean.
- Any MBean compliant with the JMX specification can be registered and be exposed for management.
- Any of the provided communication components can be used to respond to remote requests, and you can develop new adaptors and new connectors to respond to proprietary requests.
- The library of agent services provides management intelligence in the agent, such as autonomous operation in the case of a network failure.

At the manager level:

- All management requests on an MBean server are available remotely through a connector.
- Notification forwarding is already implemented for you.
- Proxies provide an abstraction of the communication layer and simplify the design of the management application.
- Basic management tasks are implemented in the agent by the agent services.

At all three levels, the modularity also means the simple designs can be implemented rapidly, and then additional functionality can be added as needed. You can have a prototype running after your first day of development, because of the programming examples provided in the product.

1.3.2 Protocol Independence

The design of MBeans, agents, and managers does not depend in any way on the protocol that an agent uses for communicating with external applications. All interactions with MBeans are handled by the MBean server and are thus defined by the JMX APIs.

The provided connectors rely on the API and do not expose any communication details. A connector server, connector client pair can be replaced by another pair without loss of functionality, assuming both protocols are in the network environment. Applications can thus switch protocols according to real-time conditions. For example, if a manager must access an agent behind a firewall, it can instantiate and use an HTTP connector.

Because MBeans and agents are protocol independent, they can be accessed simultaneously through any number of protocols. Connector servers and protocol adaptors can handle multiple connections, so your agent needs only one of them for each protocol to which it responds. The MBean server also supports simultaneous requests, although MBeans are responsible for their own synchronization issues.

New connectors for new protocols can be developed and be used without rewriting existing MBeans or external applications. All that is required is that the new connector client expose the remote API.

The Java DMK 5.1 supports multihome interfaces, allowing you to work in environments where multiple network protocols are available. The multihome interface service means that Java DMK 5.1 offers complete support of the internet protocol version 6 (IPv6), provided it is running on a platform that is IPv6 compatible, namely JDK™ version 1.4 and higher.

1.3.3 Dynamic Extensibility and Scalability

By definition, all agents and manager applications developed with the Java DMK 5.1 are extensible and scalable. The library of agent services is always available. Managers can instantiate new services when needed and later remove them to minimize memory usage. This is especially useful for running agents on small footprint devices.

In the same way, MBeans can be registered and be unregistered with the MBean server in an agent while the agent is running. This is useful to represent application resources that can come and go on a given host. The scalability enables an agent to adapt to the size and complexity of its managed resources, without having to be restarted or be reinstalled.

The dynamic loading service can download and instantiate MBeans from an arbitrary location. Therefore, you can extend the functionality of a running agent by making new classes available at an arbitrary location and requesting that the agent load and instantiate them. This is effectively a push mechanism that can be used to deploy services and applications to customers.

In addition, open MBeans contribute to the flexibility and scalability of management systems by enabling management applications to use new managed objects as the objects are created.

Finally, conformance to the JMX specification ensures that all components that are compatible with the JMX specification can be incorporated into Java dynamic management agents, whether they are manageable resources, new services, or new communication components.

1.3.4 SNMPv3 Protocol

Java DMK 5.1 provides an implementation of the SNMPv3 protocol. This means that Java DMK benefits from the security and administration services offered by SNMPv3.

Java DMK supports SNMPv1 and v2 fully, and implements much of SNMPv3. A single agent can respond to requests from any version of SNMP.

For more information about security using the SNMPv3 protocol, see "2.6 Security" on page 50.

1.3.5 SNMP Master Agent

The SNMP support in Java DMK 5.1 allows you to build a *master agent*. An SNMP master agent groups together several SNMP subagents and exports their information through a single point of access. The master agent performs the following two main functions.

- Registers subagents to handle a MIB or a part of a MIB. A subagent can provide a local implementation of the MIB, in the form of the usual Java DMK `SnmpMibAgent` class. The master agent can also be a proxy, representing a remote SNMP agent, to which the request will be forwarded.
- Converts requests from the SNMP version supported by the manager into the version supported by the subagent. The master agent also converts the responses back, and converts the traps sent by the subagent into the version used by the manager.

1.4 Overview of the Product Documentation

Java DMK includes both printable and online documentation, as well as a set of programming examples.

1.4.1 Online HTML Files

You can view HTML documentation after installation of the product. On the machine where you installed the product, open one of the following URLs in any browser:

installDir`/SUNWjdmk/5.1/doc/index.html`

The page contains links to all the product documentation that is supplied online with Java DMK.

1.4.2 Printable Documents

Complete PDF versions of the books listed in the preceding section are also supplied with the CD—ROM release of Java DMK 5.1. These files are also located in the `/doc` directory on the CD-ROM.

The documents are formatted for U.S. Letter paper size (8.5 × 11 inches), but they can be loaded by any appropriate document viewer or printed directly to any printer. The text area on each page fits on most standard paper sizes.

1.4.3 Programming Examples

Sample applications that demonstrate most of the functionality of the Java DMK are provided in the examples package of the product. If you installed this package, the Java source files and `README` text files for these applications are located in subdirectories:

- *installDir*`/SUNWjdmk/5.1/examples/current`
- *installDir*`/SUNWjdmk/5.1/examples/legacy`

The examples in the `/legacy` subdirectory demonstrate the features that have been deprecated in Java DMK version 5.1, but retained for reasons of backwards compatibility.

Note – In the Solaris operating environment, you need to be root user to write to this directory. To compile the example programs, users should copy the examples hierarchy to a more accessible location.

The `README` file for each example gives a brief explanation of the source files and the instructions for running its application. Further explanation for most examples is available in the *Java Dynamic Management Kit 5.1 Tutorial*.

1.4.4 API Documentation

The API documentation generated by the Javadoc utility provides the full description of all classes, interfaces, and methods in the Java DMK APIs.

The generated API documentation for the Java DMK is found in the following location after installation:

installDir`/SUNWjdmk/5.1/doc/index.html`

CHAPTER 2

Architectural Components

This chapter presents the components of the Java Dynamic Management Kit (Java DMK) and explains how you can use them in a complete management solution. Some features are implemented in a slightly different way for the deprecated legacy connectors. For further information on features specific to legacy connectors, see Chapter 4.

This chapter contains the following sections:

- "2.3.1.1 RMI Connectors" on page 35 describes the three ways to instrument a resource so that it is manageable.
- "2.2 The MBean Server" on page 33 explains how a JMX agent exposes the MBeans it contains.
- "2.3 Communication Components" on page 34 presents the components that establish connections between agents and managers.
- "2.4 The Notification Model" on page 40 explains how resources and agents can signal events and how events are forwarded to remote listeners.
- "2.5 Agent Services" on page 42 briefly explains each agent service.
- "2.6 Security" on page 50 describes the security features built into the communication components of the Java DMK.
- "2.7 The SNMP Toolkit" on page 54 explains how to develop Java applications for SNMP agents and managers.

2.1 Instrumenting Resources as MBeans

The instrumentation level of the JMX specification defines standards for making resources manageable in the Java programming language. The instrumentation of a manageable resource is provided by one or more management beans, or MBeans. An MBean is a Java object that exposes attributes and operations for management. These attributes and operations enable any Java dynamic management agent to recognize and manage the MBean.

The design patterns for MBeans give the developer explicit control over how a resource, device, or application is managed. For example, attribute patterns enable you to distinguish between a read-only and a read-write property in an MBean. The set of all attributes and operations exposed to management through the design patterns is called the *management interface* of an MBean.

Any resource that you want to make accessible through an agent can be represented as an MBean. Both the agent application and remote managers can access MBeans in an agent. MBeans can generate notification events that are sent to all local or remote listeners. For more information about managing MBeans remotely, see "2.3 Communication Components" on page 34.

You can make resources accessible through Java DMK agents even if they are not instrumented as MBeans, by using MBean Interceptors. See "2.6.1.4 Server authentication" on page 52 for details.

You can also download MBeans from a web server and plug them into an agent at any time, in response to a demand from the management application. This is called *dynamic class loading* and means that future services and applications can be loaded on the fly and without any downtime. For example, dynamic class loading can be used to provide rapid, low-cost delivery of end-user applications across very large bases of Java technology enabled devices, such as desktop PCs or Web phones.

There are four types of MBean:

- Standard MBeans
- Dynamic MBeans
- Model MBeans, which are an extension of dynamic MBeans
- Open MBeans, which are an extension of dynamic MBeans

2.1.1 Standard MBeans

Standard MBeans are Java objects that conform to certain design patterns that are derived from the JavaBeans component model. Standard MBeans allow you to define your management interface straightforwardly in a Java interface. The method names of this interface determine getters and setters for attributes and the names of operations. The class implementation of this interface contains the equivalent methods for reading and writing the MBean's attributes and for invoking its operations, respectively.

The management interface of a standard MBean is static, and this interface is exposed statically. Standard MBeans are static because the management interface is defined by the source code of the Java interface. Attribute and operation names are determined at compilation time and cannot be altered at runtime. Changes to the interface must be recompiled.

Standard MBeans are the quickest and easiest type of MBeans to implement. They are suited to creating MBeans for new manageable resources and for data structures that are defined in advance and that are unlikely to change often.

2.1.2 Dynamic MBeans

Dynamic MBeans do not have getter and setter methods for each attribute and operation. Instead, dynamic MBeans have generic methods for getting or setting an attribute by name, and for invoking operations by name. These methods are common to all dynamic MBeans and are defined by the `DynamicMBean` interface.

The management interface is determined by the set of attribute and operation names to which these methods respond. The `getMBeanInfo` method of the `DynamicMBean` interface must also return a data structure that describes the management interface. This metadata contains the attribute and operation names, their types, and the notifications that can be sent if the MBean is a broadcaster.

Dynamic MBeans provide a simple way to wrap existing Java objects that do not follow the design patterns for standard MBeans. You can also implement dynamic MBeans to access resources that are not based on Java technology by using the Java Native Interface (JNI).

The management interface of a dynamic MBean is static, but this interface is exposed dynamically when the MBean server calls its `getMBeanInfo` method. The implementation of a dynamic MBean can be complex, for example, if the MBean determines its own management interface based on existing conditions when it is instantiated.

2.1.3 Model MBeans

A model MBean is a generic, configurable, dynamic MBean that you can use to instrument a resource at runtime. A model MBean is an MBean template. The caller tells the model MBean what management interface to expose. The caller also determines how attributes and operations are implemented, by designating a target object on which attribute access and operation invocation are actually performed.

The model MBean implementation class is mandated by the JMX specification and therefore is always available for instantiation in an agent. Management applications can use model MBeans to instrument resources on the fly.

To instrument a resource and expose it dynamically, you need to:

- Instantiate the `javax.management.modelmbean.RequiredModelMBean` class in a JMX agent
- Set the model MBean's management interface
- Designate the target object that implements the management interface
- Register the model MBean in the MBean server

The management interface of a model MBean is dynamic, and it is also exposed dynamically. The application that configures a model MBean can modify its management interface at any time. The application can also change its implementation by designating a new target object.

Management applications access all types of MBeans in the same manner, and most applications are not aware of the various MBean types. However, if a manager understands model MBeans, it can obtain additional management information about the managed resource. This information includes behavioral and runtime metadata that is specific to model MBeans.

2.1.4 Open MBeans

Open MBeans allow management applications and the users to understand and use new managed objects as the objects are discovered at runtime. These MBeans are called *open* because they rely on a small, predefined set of universal Java types and they advertise their functionality.

Management applications and open MBeans are thus able to share and use management data and operations at runtime without requiring the recompiling, reassembly, or expensive dynamic linking of management applications. In the same way, human operators can use the newly discovered managed object intelligently without having to consult additional documentation.

To provide its own description to management applications, an open MBean must be a dynamic MBean. Beyond the `DynamicMBean` interface, no corresponding open interface exists that must be implemented. Instead, an MBean earns its openness by providing a descriptively rich metadata and by using only certain predefined data types in its management interface.

An open MBean has attributes, operations, constructors, and possibly notifications like any other MBeans. An open MBean is a dynamic MBean with the same behavior and all of the same functionality. An open MBean also has the responsibility of providing its own description. However, all of the object types that the MBean manipulates, its attribute types, its operation parameters and return types, and its constructor parameters, must belong to a defined set of basic data types. It is the developer's responsibility to implement the open MBean fully by using these data types only.

An MBean indicates whether it is open or not through the `MBeanInfo` object it returns. Open MBeans return an `OpenMBeanInfo` interface. This interface is implemented by `OpenMBeanInfoSupport`, which inherits from `MBeanInfo`. If an MBean is marked as open in this manner, it is a guarantee that a management application compliant with the JMX specification can immediately make use of all attributes and operations without requiring additional classes.

Since open MBeans are also dynamic MBeans and they provide their own description, the MBean server does not check the accuracy of the `OpenMBeanInfo` object. The developer of an open MBean must guarantee that the management interface relies on the basic data types and provides a rich, human-readable description. As a rule, the description provided by the various parts of an open MBean must be suitable for displaying to a user through a graphical user interface (GUI).

2.2 The MBean Server

The MBean server is a registry for JMX manageable resources, which it exposes to management requests. The MBean server provides a protocol-independent and information model independent framework with services for manipulating JMX manageable resources.

If you choose to register a resource's MBean with the MBean server, the MBean becomes visible to management applications and is exposed to management requests. The MBean server makes no distinction between the types of MBeans. Standard, dynamic, model and open MBeans are managed in exactly the same manner.

You can register objects in the MBean server through the following.

- The other objects in the agent application.
- A remote management application, through a connector or a protocol adaptor.

The MBean server responds to the following management requests on registered MBeans.

- Listing and filtering MBeans by their symbolic name
- Discovery and publication of the management interface of MBeans
- Accessing MBean attributes for reading and writing
- Invoking operations defined in the management interface of MBeans
- Registering and unregistering listeners for MBean notifications

The MBean server never provides the programmatic reference of its MBeans. The MBean server treats an MBean as an abstraction of a management entity, not as a programmatic object. All management requests are handled by the MBean server, which dispatches them to the appropriate MBean, thus ensuring the coherence in an agent.

An MBean is identified by a unique symbolic name, that is called its *object name*. The object name can be assigned either by the entity registering the MBean or by the MBean itself, if its implementation has been designed to provide one. Managers give this object name to designate the target of their management requests. Unless specified otherwise, object names are local to a specific MBean server. You can however make object names *global* if you want to implement the cascading service (see "2.5.5 Cascading" on page 45).

It is possible to have multiple MBean servers within the same Java virtual machine, with each MBean server managing a set of resources.

2.3 Communication Components

Connectors and protocol adaptors interact with the Java communication objects such as sockets to establish connections and respond to requests from other host machines. Connectors and protocol adaptors enable agents to be accessed and be managed by remote management applications.

An agent can contain any number of connectors or protocol adaptors, enabling the agent to be managed simultaneously by several managers, through different protocols. The agent application is responsible for coordinating all the ports on which it intends to receive requests.

2.3.1 Connectors

Connectors establish a point-to-point connection between an agent and a management application, each running in a separate Java virtual machine.

A connector is composed of two parts:

- A connector server, which interacts with the MBean server in an agent
- A connector client, which exposes a manager-side interface that is identical to the MBean server interface

Therefore, a Java application that instantiates a connector client can perform all management operations that are available through the agent's MBean server.

The Java DMK 5.1 provides the three standard connectors that are defined by the JMX Remote API 1.0 (JSR 160) specification. The standard connectors are based on RMI, RMI/IIOP, and JMX Messaging Protocol (JMXMP) which uses Java serialization over TCP.

The Java DMK 5.1 also provides connectors for the HTTP/TCP, HTTP/SSL, and RMI protocols. These "legacy" connectors were included in previous versions of the product, and are deprecated in version 5.1, now that standard connectors are available. Java DMK 5.1 provides a mechanism to wrap legacy connectors so that they can be created in the same way as the new connectors. Use of the legacy connectors is discouraged, although the use of the legacy HTTP(S) connector is acceptable as long as no native standard HTTP connectors are present.

Every standard connector provides the same remote API, which frees management applications from protocol dependencies. The API provided by standard connectors is similar to that provided by legacy connectors, but they are not interchangeable. In particular, the address format used by standard connectors is different from that used by legacy connectors.

The addresses of the new connectors are instances of the JMX Remote API interface `JMXServiceURL`. The address must adhere to the following syntax.

`service:jmx:`*protocol*`:`*sap*

Here, *protocol* is the transport protocol to be used to connect to the connector server, and *sap* is the address at which the connector server is found. The following is an example JMX service URL.

```
service:jmx:jmxmp://localhost:5555
```

In this example, the transport protocol is JMXMP and it is found at port 5555 of the local host. See the API documentation for the `JmxServiceURL` interface for more information about the syntax of these addresses.

2.3.1.1 RMI Connectors

The RMI connector is specified by the JMX Remote API. User code uses the same generic factories to create a client or server as it would to create any other kind of connector. Every connection to an RMI connector is implemented by means of a separate RMI object, which is destroyed when the connection is closed. Communication is always from client to server, even for notifications, which simplifies configuration and firewall use.

The RMI connector works over both JRMP, (the native Java transport for RMI) and IIOP. The same remote interfaces are used in both cases. Only the way in which these interfaces are exported differs. Because of class-loading issues, the RMI/IIOP connector cannot easily be used to interact with clients or servers that are not written in the Java language.

2.3.1.2 JMXMP Connectors

The JMX Remote API specifies an optional connector using a custom protocol called JMXMP, that is based on Java serialization running over TCP connections. JMXMP can optional use one or both of SSL and the Simple Authentication and Security Layer (SASL) for security.

Communication between a given client and the server happens over a single TCP connection. Every message is a serialized Java object. Coomunication is conceptually in two independent streams of messages, from client to server and from server to client. Thus, there can be many concurrent client requests over the connection at any given time. Replies do not have to arrive in the same order as the corresponding requests.

Notifications are handled in the same way as for the RMI connector. A message from client to server asks for notifications, and a reply message from server to client supplies them.

2.3.1.3 Monitoring Standard Connectors Using the Heartbeat Mechanism

All connectors provided in the Java DMK implement a heartbeat mechanism. This is true for both legacy connectors and the new standard connectors. This section outlines the heartbeat mechanism used for standard connectors. For information on the heartbeat mechanism used for legacy connectors, see “4.1.2 Monitoring Legacy Connectors Using the Heartbeat Mechanism” on page 70.

The heartbeat mechanism monitors the connection between a manager and an agent, and automates the cleanup procedure when the connection is lost. This allows both the manager and the agent to free 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 and servers send notifications that the manager application can receive to be aware of changes in the status of a connection.

The connector client generates a periodic heartbeat, by performing an innocuous operation on the server. If any beat fails because of a communication failure, the connection is considered to be dead. The server does not need to know that the heartbeat exists. The innocuous beat operations look like any other client operation.

Furthermore, client death is not detected by heartbeat. Instead, connectors are defined in such a way that there is no permanent client state on the server. Thus, the server can close client connections after a specified idle time. If the client is still alive, it will establish another connection. This is essentially an implicit lease mechanism.

2.3.1.4 Generating Proxies

Java DMK supports the notion of proxies. Proxies simplify the interactions between an application and the MBeans the application wants to manage. The purpose of a proxy is to invoke the methods that access the attributes and operations of an MBean, through its MBean server. The proxy performs the task of constructing the method calls at every invocation, on behalf of the caller:

- Getting or setting attributes
- Invoking operations
- Registering or unregistering for notifications

Conceptually, a proxy instance makes the MBean server and, optionally, a protocol connector completely transparent. With the exception of MBean registration and connector connection phases, all management requests on MBeans can be fully served through proxies, with identical results, apart from some details concerning exceptions. However, all functionality of the Java DMK is available without using proxies, so their use is never mandatory.

Figure 2–1 shows management components interacting with an MBean through a proxy.

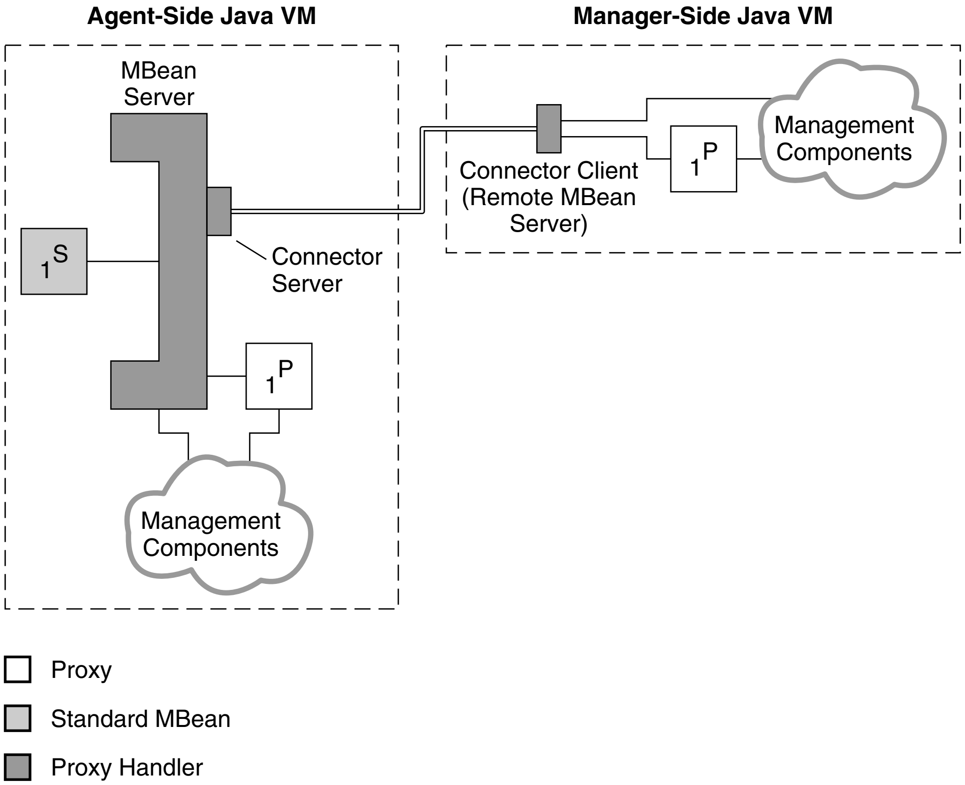

FIGURE 2–1 Binding Proxy MBeans to Local and Remote Servers

Figure 2–1 also shows that proxies can be instantiated either locally in the agent or remotely in the manager. Since the MBean server and the connector client have the same API, management requests to either of them are identical. This creates a symmetry so that the same management components can be instantiated either in the agent or in the manager application. This feature contributes to the scalability of Java dynamic management applications.

In Java DMK 5.0, proxies were generated using the `proxygen` tool, supplied with Java DMK 5.1 but now deprecated. This tool is still needed if proxies are required for legacy connectors. However, Java DMK 5.1 provides a useful enchancement for generating proxies. Because Java DMK 5.1 is an implementation of JMX 1.2, you can generate a proxy object at runtime given just its Java interface, using the dynamic proxies provided with the J2SE platform (`java.lang.reflect.Proxy`). These proxies cannot be used with the legacy connectors.

2.3.2 MBean Server Interceptors

As stated previously, the Java DMK does not require every MBean in a Java DMK agent to be represented by a Java object in that agent. MBean Server *interceptors* enable you to intercept operations on MBeans and handle them arbitrarily. Handling the operations can involve handing the request to other interceptors, possibly after logging or authenticating them for security. Alternatively, handling can involve processing the request directly. For example, with very volatile MBeans, direct handling avoids having to keep up with the creation and deletion of objects. Instead, the managed object is effectively synthesized when there is a request on it, which for volatile objects happens much less often than creation and deletion.

Note – In Java DMK version 5.1, it is necessary to use the new `JdmkMBeanServerBuilder` class to add interceptor functionality. This can be done by specifying a Java system property. See the *Java Dynamic Management Kit 5.1 Tutorial* for details.

2.3.3 Protocol Adaptors

Protocol adaptors have only a server component and provide a view of an agent and its MBeans through a different protocol. Protocol adaptors can also translate requests that are formulated in this protocol into management requests on the JMX agent. The view of the agent and the range of possible requests depends upon the given protocol.

For example, Java DMK provides an HTML adaptor that presents the agent and its MBeans as HTML pages that are viewable in any web browser. Because the HTML protocol is text based, only data types that have a string representation can be viewed through the HTML adaptor. However, this is sufficient to access most MBeans, view their attributes, and invoke their operations.

Due to limitations of the chosen protocol, adaptors have the following limitations.

- Not all data types are necessarily supported
- Not all management requests are necessarily supported, because some requests might rely on unsupported data types
- Notifications from a broadcaster MBean might not be supported
- A given protocol adaptor might require private data structures or helper MBeans for responding to requests

The SNMP adaptor provided in the Java DMK is limited by the constraints of SNMP. The richness of the JMX architecture cannot be translated into SNMP, but all the operations of SNMP can be imitated by methods of the MBean server. This translation requires a structure of MBeans that imitates the MIB. While an SNMP manager cannot access the full potential of the JMX agent, the MBeans representing the MIB are available for other managers to access and incorporate into their management systems.

In general, a protocol adaptor tries to map the elements of the JMX architecture into the structures provided by the given protocol. However, the completeness or accuracy of this mapping is not guaranteed.

2.4 The Notification Model

The JMX architecture defines a notification model that enables MBeans to broadcast notifications. Management applications and other objects register as listeners with the broadcaster MBean. In this way, MBeans can signal asynchronous events to any interested parties.

The JMX notification model enables a listener to register only once and still receive all the various notifications that an MBean can broadcast. A listener object can also register with any number of broadcasters, but it must then sort all notifications it receives according to their source.

2.4.1 Local Notification Listeners

In the simplest case, listeners are objects in the same application as the broadcaster MBean. The listener is registered by calling the `addNotificationListener` method on the MBean. The MBean server exposes the same method so that listeners can also be added to an MBean identified by its symbolic name.

In Figure 2–2, one listener has registered directly with the MBean and another has registered through the MBean server. The end result is the same, and both listeners receive the same notifications directly from the broadcaster MBean.

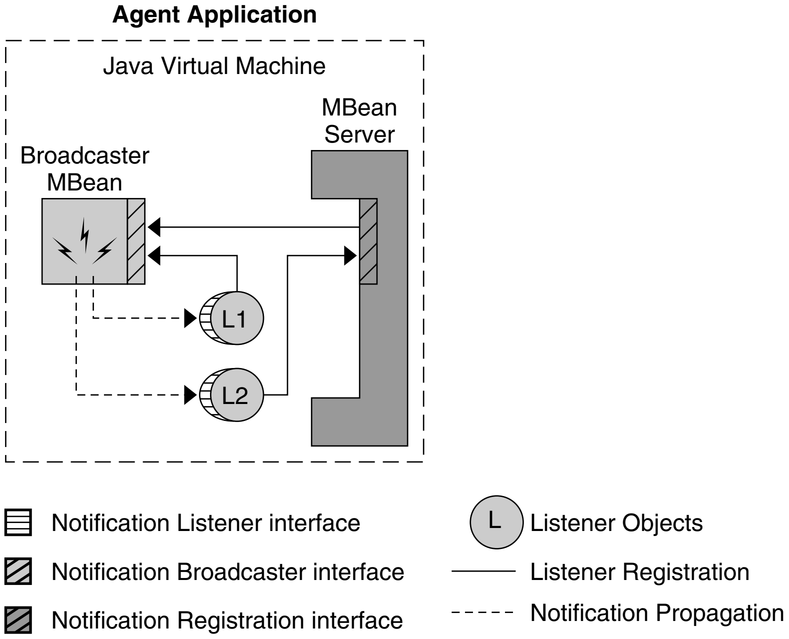

FIGURE 2–2 Adding Local Listeners on the Agent Side

2.4.2 Remote Notification Listeners

The connector client interface also exposes the `addNotificationListener` method so that notifications can be received in remote management applications. Standard proxies expose this method as well and transmit any listener registrations through the connector client.

Listeners do not need to be aware that they are remote. The connector transmits registration requests and forwards notifications back to the listeners. The whole process is transparent to the listener and to the management components.

As shown in Figure 2–3, the connector components implement a complex mechanism for registering remote listeners and forwarding notifications. Because notifications are based on the Java event model, broadcasters cannot send notifications outside their Java virtual machine. So, the connector server instantiates local listeners that receive all notifications and places them in a cache buffer, to wait to be sent to the manager application. This enables the connector to avoid saturating the communication layer in case of a burst of notifications.

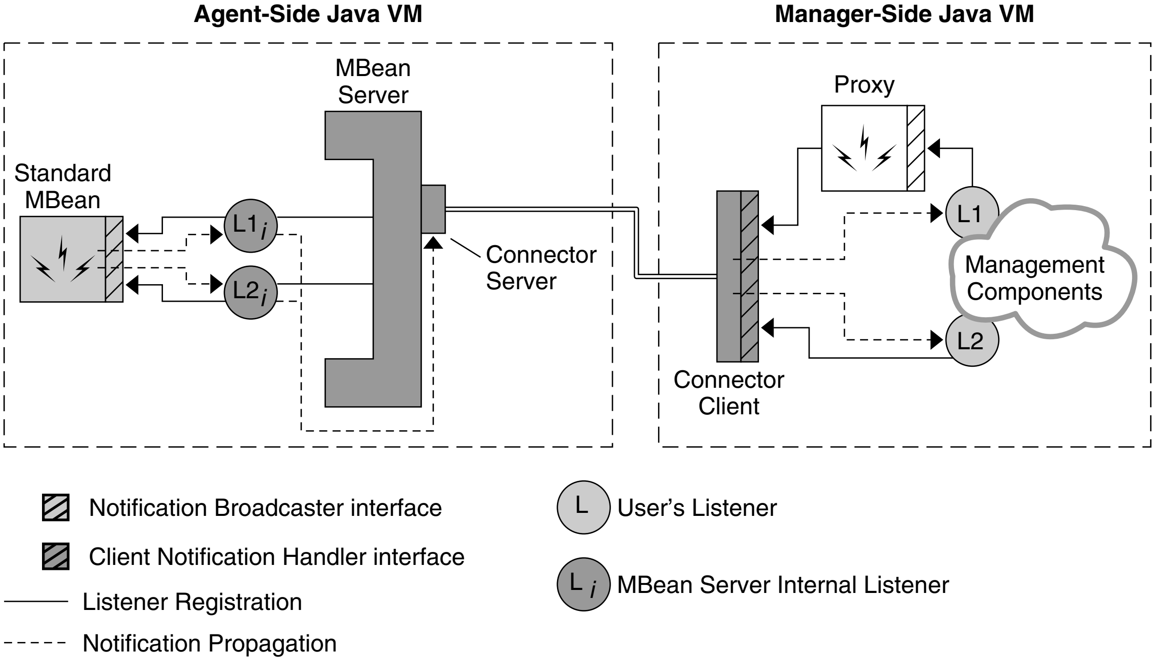

FIGURE 2–3 Adding Remote Listeners on the Manager Side

Notifications in the new RMI and JMXMP connectors are pulled periodically at the client's request. The pull mechanism is used to group notifications and reduce bandwidth usage. The connector client acts as a broadcaster and sends the notifications to their intended listeners.

Notifications in the legacy RMI and HTTP connectors are pulled in the same way as the new connectors. However, the legacy connector notifications can also can be pushed from the agent to the connector client as they are received.

2.5 Agent Services

To simplify the development of agents for network, system, application, and service management, the Java DMK supplies a set of agent services. These services are implemented as MBeans that perform some operations on the other MBeans in an agent. All the provided agent services are briefly explained in this section.

2.5.1 Querying and Filtering

Querying and filtering are performed by the MBean server, not by a separate MBean. This ensures that such critical services are always available. Queries and filters are performed in a single operation, whose goal is to select the MBeans on which management operations are performed.

Usually, a management application performs a query to find the MBeans that are the target of its management requests. To select MBeans, applications can specify the following.

- *An object name filter*, which is a possibly incomplete object name that the MBean server tries to match with the object names of all registered MBeans. All MBeans whose names match the filter pattern are selected. Filters can contain wildcards to select sets of MBeans, or a filter can be a complete object name that must be matched exactly. Filter rules are explained in detail in the JMX specification.
- *A query expression*, which is an object that represents a set of constraints applied to the attribute of an MBean. For each MBean that passes the filter, the MBean server determines if the current state of the MBean satisfies the query expression. Queries usually test for attribute values or MBean class names.

For example, a filter could select all the MBeans whose object names contain `MyMBeans` and for which the attribute named `color` is currently equal to `red`.

The result of a query operation is a list of MBean object names, which can then be used in other management requests.

2.5.2 Dynamic Loading

Dynamic class loading is performed by loading management applets or *m-lets* containing MBeans. This service loads classes from an arbitrary network location and creates the MBeans that they represent. The m-let service is defined by the JMX specification. The m-let service makes it possible to create dynamically extensible agents.

A management applet is an HTML-like tag called `<MLET>` that specifies information about the MBeans to be loaded. It resembles the `<APPLET>` tag, except that it loads only MBean classes. The tag contains information for downloading the class, such as the classname and the location of its class file. You can also specify any arguments to the constructor that is used to instantiate the MBean.

The m-let service loads a URL that identifies the file containing `<MLET>` tags, one for each MBean to be instantiated. The service uses a class loader to load the class files into the application's Java virtual machine. It then instantiates these classes and registers them as MBeans in the MBean server.

The m-let service is implemented as an MBean and instantiated and registered in the MBean server. Thus, the m-let service can be used either by other MBeans or by management applications. For example, an application could make new MBean classes available at a location, generate the m-let file, and instruct the m-let service in an agent to load the new MBeans.

Dynamic loading effectively pushes new functionality into agents, allowing management applications to deploy upgrades and to implement new resources in their agents.

2.5.3 Monitoring

The monitoring service complies with the JMX specification and provides a polling mechanism based on the value of MBean attributes. The monitoring service contains three monitor MBeans, one MBean for counter attributes, another MBean for gauge-like attributes, and a third MBean for strings. These monitors send notifications when the observed attribute meets certain conditions, mainly equaling or exceeding a threshold.

Monitor MBeans observe the variation of an MBean attribute's value over time. All monitors have a configurable granularity period that determines how often the attribute is polled. Each monitor has specific settings for the type of the observed attribute, as follows.

- *Counter monitor*, which observes an attribute of the integer type (`byte`, `integer`, `short` or `long`) that is monotonically increasing. The counter monitor has a threshold value and an offset value to detect counting intervals. The counter monitor resets the threshold if the counter rolls over.
- *Gauge monitor*, which observes an attribute of integer (`byte`, `integer`, `short`, or `long`) or floating-point (`float` or `double`) types that fluctuates within a given range. The gauge monitor has both a high and low threshold, each of which can trigger a distinct notification. The two thresholds can also be used to avoid repeated triggering when an attribute oscillates around a threshold.
- *String monitor* – Observes an attribute of type `String`. The string monitor performs a full string comparison between the observed attribute and its match string. A string monitor sends notifications both when the string matches and when it differs at the observation time. Repeated notifications are not sent, meaning that only one notification is sent the first time the string matches or differs.

Monitor notifications contain the name of the observed MBean, the name of the observed attribute, and the value that triggered the event, as well as the previous value for comparison. This information allows listeners to know which MBean triggered an event. The listeners do not need to access the MBean before taking the appropriate action.

Monitor MBeans can also send notifications when certain error cases are encountered during an observation.

2.5.4 Scheduling

The timer service is a notification broadcaster that sends notifications at specific dates and times. The timer service provides a scheduling mechanism that can be used to trigger actions in the listeners. Timer notifications can be single events, repeated events, or indefinitely repeating events. The timer notifications are sent to all of the service's listeners when a timer event occurs.

The timer service manages a list of dated notifications, each of which has its own schedule. Users can add or remove scheduled notifications from this list at any time. When adding a notification, users provide its schedule, defined by the trigger date and repetition policy, and information that identifies the notification to its listeners. The timer service uses a single Java thread to trigger all notifications at their designated time.

You can stop the timer service to prevent it from sending notifications. When you start it again, notifications that could not be sent while the timer was stopped are either sent immediately or discarded, as determined by the configuration of the service.

Like all other agent services, the timer is implemented as an MBean so that it can be registered in an agent and configured by remote applications. However, the timer MBean can also be used as a stand-alone object in any application that needs a simple scheduling service.

For more information about the timer service, see the JMX specification document.

2.5.5 Cascading

Cascading is the term used to describe a hierarchy of agents, where management requests are passed from a master agent to one of its subagents. A master agent connects to other agents, possibly remotely, through their connector server components, much like a manager connects to an agent. In a set of cascading agents, all MBeans in a subagent are visible as if they are registered in their master agent. The master agent hides the physical location of subagents and provides client applications with a centralized access point.

In Java DMK 5.1, the cascading service is implemented over JMX Remote API connectors. The `CascadingServiceMBean` makes it possible to mount *source MBean servers*, that are possibly located in subagents, into a *target MBean server*, that is located in the master agent, in a manner that is somewhat analogous to a File System *mount* operation.

2.5.5.1 Object Names and Domain Paths

The Java DMK cascading API introduces the notion of a *domain path*. An `ObjectName` is thus decomposed into three parts, as follows.

domain-path/*domain-base-name*:*key-property-list*

The domain path is a hierarchical name similar to a File System path name, using the character '/' as a separator.

2.5.5.2 File System Analogy

The `CascadingServiceMBean` provided in the Java DMK 5.1 makes it possible to mount MBeans from a source MBean server under a target domain path in a target MBean server, in a similar way to a File System mount operation.

Although our API also allows you to implement different cascading schemes, we recommend that applications 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 naming 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.

Our implementation does not enforce those rules, but applications which are concerned with naming consistency and coherency should make sure to respect them. See the package description in the API documentation for the `com.sun.jdmk.remote.cascading` package for details.

2.5.5.3 `CascadingServiceMBean`

The cascading service proposed in the `com.sun.jdmk.remote.cascading` package is based on a simple MBean class, the `CascadingServiceMBean`.

- The `CascadingServiceMBean` provides methods that make it possible to mount MBeans from a source MBean server in a target MBean server under a target domain path. Usually the target MBean server is the MBean server in which the `CascadingService` is registered.

  There should be only one `CascadingServiceMBean` per target MBean server.

- The CascadingServiceMBean.mount method mounts a partial view of a source MBean server known by its JMXServiceURL in the target MBean server of the CascadingServiceMBean.
- The CascadingServiceMBean.unmount method cancels a previous mount operation. The unmount operation will close the connection that was opened by the mount operation.

The default CascadingService implementation provided in the Java DMK 5.1 relies on proxy-based cascading and implements the mount operation by instantiating a ProxyCascadingAgent behind the scenes. Although the ProxyCascadingAgent offers a public API, you should not use it directly. Applications should use the CascadingServiceMBean instead.

CascadingServiceMBeans are also notification emitters, which emit notifications when mountpoints are unmounted, as a result of a an unmount operation, or because the underlying connection with the source MBean server has been closed or failed.

2.5.5.4 Cascading over Java DMK legacy connectors

There are two possibilities to implement cascading over Java DMK legacy connectors.

- Use the legacy cascading service that was provided in earlier versions of the Java DMK. This legacy cascading service is still provided for reasons of backwards compatibility, but it is now deprecated. This alternative is therefore not recommended.
- Create a JMXServiceURL from which a JMXConnector wrapping a legacy Java DMK ConnectorClient can be obtained from the JMXConnectorFactory. You can then use this JMXServiceURL with the new CascadingServiceMBean API. See the *Java Dynamic Management Kit 5.1 Tutorial* for more information about wrapping legacy connectors.

2.5.6 Discovering Agents

You can use the discovery service to discover Java dynamic management agents in a network. Only agents that have a discovery responder registered in their MBean server can be discovered when you use this service.

The discovery service for legacy connectors can be functionally divided into two parts:

- The discovery *search* service which actively finds other agents
- The discovery *support* service which listens for other agents to be activated

2.5.7 Discovery Search Service

In a discovery search operation, the discovery client sends a discovery request to a multicast group and waits for responses. To be found by the discovery service, the agents must have a `DiscoveryResponder` registered in their MBean server. All discovery responders that receive the discovery request send a response that contains information about the connectors and the protocol adaptor that are available in their agent.

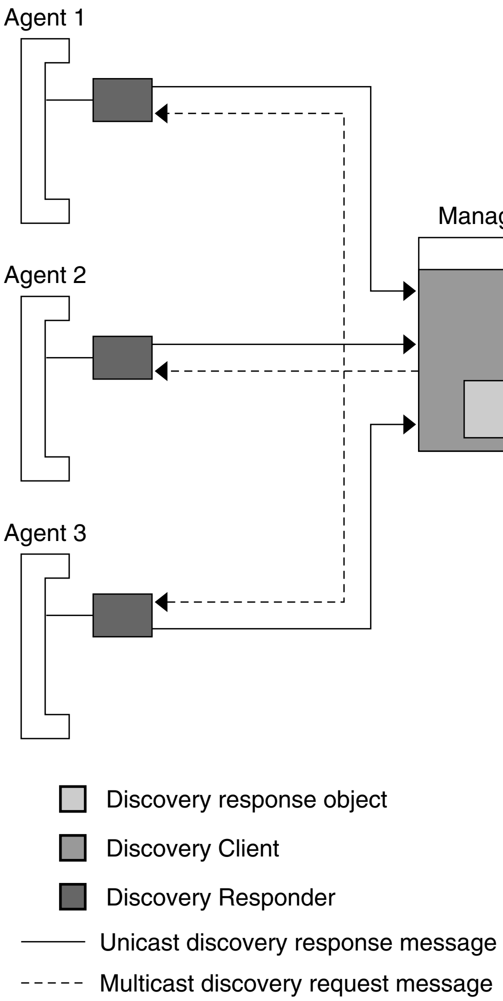
Agent 1
Agent 2
Agent 3
Manager
Discovery response object
Discovery Client
Discovery Responder
Unicast discovery response message
Multicast discovery request message

FIGURE 2–4 The Discovery Search Service

A manager application might use the discovery search service during its initialization phase, to determine all agents that are accessible in its network environment.

2.5.8 Discovery Support Service

The discovery support service passively monitors discovery responders in a multicast group. When discovery responders are activated or deactivated, indicating that their agent is starting or stopping, they send a multicast message about their new state. A *discovery monitor* object listens for discovery responder objects starting or stopping in the multicast group.

By registering listeners with the discovery monitor, a management application knows when agents become available or unavailable. The discovery support message for an agent that is being started also lists its connector and protocol adaptor.

A management application can use the discovery monitor to maintain a list of active agents and the protocols they support.

2.5.9 Defining Relations

The Relation Service is used to record relationships between MBeans in an MBean server. The Relation Service is itself an MBean. More than one instance of a Relation Service MBean can be registered in an MBean server.

A *relation type* defines a relationship between MBeans. A relation type contains roles that the MBeans play in the relationship. Usually, there are at least two roles in a relation type.

A *relation* is a named instance of a relation type, where specific MBeans appear in the roles, represented by their object names.

For example, suppose `Module` MBeans represent the modules within an application. A `DependsOn` relation type could express the relationship that some modules depend on others, which could be used to determine the order in which the modules are started or stopped. The `DependsOn` relation type would have two roles, `dependent` and `dependedOn`.

Every role is *typed*, meaning that an MBean that appears in that role must be an instance of the role's type. In the `DependsOn` example, both roles would be of type `Module`.

Every role has a *cardinality*, which provides lower and upper bounds on the number of MBeans that can appear in that role in a given relation instance. Usually, the lower and upper bounds are both 1, with exactly one MBean appearing in the role. The cardinality only limits the number of MBeans in the role per relation instance. The same MBean can appear in the same role in any number of instances of a relation type. In the `DependsOn` example, a given module can depend on many other modules, and be depended on by many others, but any given relation instance links exactly one dependent module with exactly one `dependedOn` module.

A relation type can be created explicitly, as an object implementing the RelationType interface, typically a RelationTypeSupport. Alternatively, a relation type can be created implicitly using the Relation Service's createRelationType method. A relation instance can be created explicitly, as an object implementing the Relation interface, typically a RelationSupport. A RelationSupport is itself a valid MBean, so it can be registered in the MBean server, though this is not required. Alternatively, a relation instance can be created implicitly using the Relation Service's createRelation method.

Through the relation service, users can create relation types and then create, access, and delete instances of a relation. All MBeans are referenced by their object name, so that a relation can be accessed from a remote application. An MBean does not need to know what relations it participates in. New kinds of relations can be added to an agent without having to modify the code of the MBeans that they link.

The relation service provides query mechanisms to retrieve MBeans that are related to each other. The relation service is notified when MBeans in a relation are unregistered, and it verifies that any relation involving that MBean still has the required cardinality.

The relation service can represent a relation instance either internally or externally. If the user defines a relation instance through the API of the relation service, the relation is represented by internal structures that are not accessible to the user. This is the simplest way to define relations, because the relation service handles all coherence issues through its internal structures.

A relation instance can also be a separate MBean object that fulfills certain requirements. The user instantiates and registers these MBeans, ensures that they represent a coherent relationship, and places these MBeans under the control of the relation service. This process places the responsibility of maintaining coherency on the user, but external relations have certain advantages. They can implement operations on a relation instance. Because external relations are MBeans, these extended operations are available to remote management applications.

2.6 Security

The Java DMK provides several security mechanisms to protect your agent applications. As is always the case, simple security that enforces management privileges is relatively easy to implement. However, full security against mischievous attacks requires a more sophisticated implementation and deployment scheme. However, in all cases the security mechanisms preserve the Java dynamic management architecture and management model.

New connector protocols were brought into Java DMK in version 5.1, with the integration of the JMX Remote API. These new connectors implement new security mechanisms.

The following sections give an overview of the new security features that are provided through components of the Java DMK.

2.6.1 Security for Standard Connectors

Three main aspects to connector security exist in Java DMK.

- *Privacy*, ensuring that attackers cannot see potentially sensitive information or operations over the connection, or inject new operations into it.
- *Authentication*, determining the identity of the client, and optionally of the server.
- *Authorization*, limiting the operations that a client can do based on its authenticated identity.

2.6.1.1 Privacy

The JMXMP connector negotiates security parameters during the initial handshake of a connection. In particular, the JMXMP connector can negotiate that the connection use transport layer security (TLS), which is basically the same as SSL 3.0. The server can require that only connections with TLS are accepted.

The SASL mechanisms DIGEST-MD5 and GSSAPI also provide connection privacy. See "2.6.1.2 Client Authentication in the JMXMP connector" on page 51.

Privacy can be assured for the RMI connector by using an RMI *socket factory* to cause connections to be created using TLS. Java DMK 5.1 includes a socket factory that does this. The connector server imposes the socket factory, which is serialized into client stubs so that all clients automatically use it.

2.6.1.2 Client Authentication in the JMXMP connector

Authentication in the JMXMP connector is based on SASL. The handshake phase of a JMXMP connection can negotiate the SASL mechanism to use. The connector server can mandate a list of mechanisms, and reject connections that do not negotiate one of them. When a SASL mechanism successfully completes, it has authenticated a client identity, which is used to derive the Subject for the connection.

The SASL mechanisms DIGEST-MD5 and GSSAPI also provide connection privacy. For these mechanisms, a TLS connection is superfluous.

TLS also supports client authentication. The JMXMP connector can exploit this to accept only clients that can authenticate themselves, but in this case it does not currently support authorization based on the authenticated identity.

2.6.1.3 Client Authentication in the RMI connector

The RMI connector provides a simple way to add authentication. This mechanism is unambitious, but is powerful enough to build real solutions. However, where security is a major concern, users should consider using the JMXMP connector instead.

An RMI connector server can supply a `JMXAuthenticator`. This is a Java object with a method that takes an arbitrary *credentials* object and either returns a Java `Subject` if the credentials are accepted, or throws an exception if they are not. When a connection is made, if the authenticator accepts the credentials then subsequent operations over the connection are performed as the returned `Subject`. If the authenticator does not accept the credentials, then the connection is refused.

Challenge-response mechanisms can be introduced into this scheme by having the authenticator throw a specific exception containing a challenge. The client responds with new credentials that respond to the challenge.

A simple `JMXAuthenticator` is included in Java DMK 5.1. This simple authenticator is also included in Sun's implementation of the J2SE platform, version 1.5. The credentials consist of two strings, a role name and a clear text password. The authenticator consults a text file to validate the credentials. In this file, blank lines and lines beginning with `#` are ignored. Other lines must contain two blank-separated fields, again a role name and a clear text password. If the credentials match one of these lines then the connection is authenticated with a `Subject` containing the role name.

Obviously, where clear text passwords are involved, considerable caution is necessary. Connection privacy must be established if there is any danger of snooping. We talk of role names rather than user names so as not to encourage naive users to put real user passwords in the password file. A template file is included in the relevant examples that warns in comments that the file must be read-protected, that valuable passwords should not be used, and that in environments with strong security requirements this solution is inappropriate. We include this scheme for simple uses and for getting started, but expect that most deployed systems will prefer a system that does not use clear text passwords and that integrates into an existing security infrastructure.

Again, TLS also supports client authentication. Using the socket factory mentioned in "2.2 The MBean Server" on page 33, the RMI connector can be configured to accept only clients that can authenticate themselves. However, in this case it does not currently support authorization based on the authenticated identity.

2.6.1.4 Server authentication

The Java DMK's security model is focused on ensuring that rogue clients cannot harm legitimate servers. However, a complete security solution must also ensure that, if a rogue server somehow substitutes itself for the legitimate server a client expects to find, the client is not compromised. For example, a rogue server could send bogus data to the client, or overload it, or receive sensitive information from it.

Server authentication can be done using TLS. The SASL mechanisms DIGEST-MD5 and GSSAPI also support server authentication.

2.6.1.5 Authorization

Authorization works in the same way with both connectors. The authentication step produces a Java Subject, which is a collection of `Principals`. The security mechanisms in the Java platform allow permissions to be associated with each `Principal`. When a remote operation is performed, the required permissions must be present, usually because they are associated with one of the authenticated `Principals` in the policy file.

A simplified authorization scheme is supported by Java DMK. This scheme is also used in Sun's implementation of the J2SE platform, version 1.5. In the simplified scheme, Java permissions are not involved. This removes the need to create policy files and to set a security manager, which are relatively complicated. Instead, there are just two access levels, `readonly` and `readwrite`.

The `readwrite` level gives access to all MBean server operations. The only exceptions are the creation of m-lets and the addition of URLs to existing m-lets. Since these operations could allow arbitrary code to be loaded into the MBean server and run, they are forbidden even at the `readwrite` level. When there is a security manager, running arbitrary downloaded code is acceptable because it will have no permissions by default. But the simplified scheme is specifically intended for the case where there is no security manager.

The `readonly` level gives access only to operations that do not change the state of the MBean server, such as reading attributes or querying existing MBeans.

A text file defines the access levels for different principals. In this file, blank lines and lines beginning with # are ignored. Other lines must contain two blank-separated fields. The first is an authenticated principal name, and the second is `readonly` or `readwrite`.

The mechanism that checks authorization in this simplified scheme is intended for simple uses and for getting started. It is expected that users with strong security requirements will eventually graduate to using the full Java security model, with permissions, policy files, and a security manager.

2.6.1.6 Subject Delegation

Java DMK 5.1 provides for subject delegation. The idea is that a single connection authenticated with a trusted identity, the *delegate*, can perform operations on behalf of other identities, without having to authenticate those identities explicitly or to establish a different connection per identity.

The delegate must have a specific permission to perform operations on behalf of each identity it assumes. This permission can be specified with a wildcard, to allow delegation from a set of identities. Unlike most permission checks, this one happens even if there is no Java security manager.

2.7 The SNMP Toolkit

The Java Dynamic Management Kit provides a toolkit for integrating SNMP management into a JMX architecture. SNMP management includes:

- Developing an SNMP agent with the SNMP protocol adaptor.
- Representing your SNMP management information base (MIB) as MBeans generated by the `mibgen` compiler.
- Developing an SNMP manager using the SNMP Manager API, if necessary.
- Different levels of SNMP security, if necessary.

For more information regarding the SNMP toolkit, refer to the *Java Dynamic Management Kit 5.1 Tools Reference Guide* and the *Java Dynamic Management Kit 5.1 Tutorial*.

2.7.1 SNMP Packaging in Java DMK 5.1

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. However, should you require them, a full set of SNMP examples that follow the package naming from Java DMK 5.0 have been retained in the `/examples/legacy/Snmp` directory.

2.7.2 Developing an SNMP Agent

An SNMP agent is an application that responds to SNMP requests formulated as `get`, `set`, `getnext`, and `getbulk` operations on variables defined in a MIB. This behavior can be fully mapped onto the MBean server and MBean resources of a Java dynamic management agent, provided that those MBeans specifically implement the MIB. An SNMP agent can be issued either with or independently from an MBean server.

There are two SNMP protocol adaptors: one that supports SNMPv1 and v2, and another introduced in the Java DMK 5.0 that supports SNMPv3 as well as the two previous versions. All features added in the Java DMK 5.0 therefore support SNMPv3 USM MIBs, providing user-based security, and scoped MIBs, that can be registered in the adaptor using a context name. The addition of multithread support in SNMP adaptors and timers in Java DMK 5.0 has also improved the performance of SNMP.

The SNMP protocol adaptors respond to requests in SNMP and translate the requests into management operations on the specific MIB MBeans. The SNMP adaptors also send traps, the equivalent of a JMX agent notification, in response to SNMP events or errors.

The SNMP protocol adaptors can manage an unlimited number of different MIBs. These MIBs can be loaded or unloaded dynamically, by registering and unregistering the corresponding MBeans. The adaptors attempt to respond to an SNMP request by accessing all loaded MIBs. However, MIBs are dynamic only through the agent application, and the SNMP protocol does not support requests for loading or unloading MIBs.

One advantage of the dual JMX–SNMP agent is that MIBs can be loaded dynamically in response to network conditions, or even in response to SNMP requests. Other Java dynamic management applications can also access the MIB through its MBean interface. For example, the value of a MIB variable might be computed in another application and written by a call to the MBean setter.

The SNMP protocol adaptors also send inform requests from an SNMP agent to an SNMP manager. The SNMP manager sends an inform response back to the SNMP agent.

2.7.3 SNMP MIB Compiler – `mibgen`

The `mibgen` tool takes as input a set of SNMP MIBs and generates standard MBeans that you can customize. MIBs can be expressed using either structure of management information (SMI) v1 or SMI v2 syntax.

A MIB is like a management interface. It defines what is exposed, but it does not define how to compute the exposed value. Therefore, MBeans generated by `mibgen` need to be customized to provide the definitive implementation. The MIB is implemented through Java objects, meaning that it has access to all Java runtime libraries and all features of the dynamic agent where it will be instantiated.

The `mibgen` compiler parses an SNMP MIB and generates the following:

- An MBean representing the whole MIB
- MBeans representing SNMP groups and table entries
- Classes representing SNMP tables
- Classes representing SNMP enumerated types
- A class mapping symbolic names with object identifiers

The resulting classes should be made accessible in the agent application. When the single MBean representing the whole MIB is registered in the MBean server, all the associated groups are automatically instantiated and registered as well.

The `mibgen` compiler supports all data structure of SMI v1 and v2 protocols, including:

- Tables with cross-references indexed across several MIBs
- MIBs that contain either SMI v1 or v2 definitions
- Nested groups
- Default value variables
- Row status variables

The Java DMK also provides an example program, showing how an agent can act as an SNMP master agent to access MIBs implemented remotely in subagents. This allows SNMP managers to access hierarchies of agents through a single master agent. In this way, some MIBs can be implemented by native devices and others can be implemented in JMX agents, yet this heterogeneous architecture is completely transparent to the manager issuing a request.

2.7.4 SNMP Manager API

The SNMP manager API simplifies the development of Java applications for managing SNMP agents. Its classes represent SNMP manager concepts such as sessions, parameters, and peers through Java objects. Using this API, you can develop an application that can issue requests to SNMP agents.

For example, you could create an SNMP resource using the SNMP manager API. You would define a management interface that corresponds to your resource's MIB, in which variables are easily mapped as MBean attributes. In response to calls on the attribute getters and setters, your MBean would construct and issue an SNMP request to the SNMP agent that represents the resource.

The SNMP manager API supports requests in the SNMP v1, v2 or v3 protocol, including inform requests for communicating between SNMP managers. The manager API is used to access any compliant SNMP agent, including those developed with the use of the Java DMK.

2.7.5 SNMPv1 and SNMPv2 Security

Because of backward compatibility, Java DMK 5.1 implements the security aspects of the SNMP protocol v1 and v2. However, you should implement the superior security mechanisms of SNMPv3, which are added in the Java DMK 5.1.

2.7.5.1 SNMPv1 and SNMPv2 Access Control

SNMPv1 and v2 define an access control mechanism similar to password authentication. Lists of authorized manager host names are defined in an *access control list* (ACL) stored in a file on the agent side, called the IP ACL file. There are no passwords, but logical community names (IP addresses) can be associated with authorized managers to define sets of allowed operations.

The SNMP adaptor performs access control if an ACL file is defined. Because SNMP is a connection—free protocol, the manager host and community are verified with every incoming request. By default, the file is not loaded and any SNMP manager can send requests.

The ACL file is the default access control mechanism in the SNMP protocol adaptor. However, you can replace this default implementation with your own mechanism. For example, if your agent runs on a device with no file system, you could implement access control lists through a simple Java class.

2.7.5.2 SNMPv1 and SNMPv2 Encoding

SNMP requests follow the standardized Basic Encoding Rules (BER) for translating management operations into data packets. At the communication level, an SNMP request is represented by an array of bytes in a UDP protocol packet. The SNMP components in the Java DMK provide access to the byte encoding of these packets.

Your applications can customize the encoding and decoding of SNMP requests, as follows:

- On the manager side, after the request is translated into bytes, your encoding can add signature strings and then perform encryption.
- On the agent side, the bytes can be decoded and the signature can be verified before the bytes are translated into the SNMP request.

  A decoded SNMP request contains the manager's hostname and community string, the operation, the target object, and any values to be written. Like the context checking mechanism, you can insert code to filter requests based on any of these criteria. However, inserting your own code would make the protocol proprietary.

2.7.6 SNMPv3 Security

The main addition to Java DMK 5.1 provided by SNMPv3 is the possibility of secure SNMP operation. The SNMPv3 security in Java Dynamic Management Kit 5.1 implements the following SNMP RFCs:

RFC 2571	Architecture
RFC 2572	Message Processing and Dispatching
RFC 2574	USM

The SNMPv3 protocol implementation provides:

- A dispatcher, the SNMP adaptor, for sending and receiving messages
- The SNMPv3 Message Processing Model (MPM), to prepare messages for sending and to extract data from messages received
- A User-based Security Model (USM), to provide authentication and privacy for SNMP operations
- A user-based Access Control Model (ACM), to control access to Java management agents
- A USM local configuration data file (LCD) that allows configured users persistency

Despite the differences between the previous versions of SNMP and SNMPv3, agents in Java DMK 5.1 can respond to requests from any version if the SNMPv3 protocol adaptor is used. SNMP v1 and v2 requests have greater security constraints than v3 requests in an agent compatible with SNMPv3.

The USM MIB is accessible remotely and is not registered to the SNMPv3 adaptor by default.

The USM MIB can be registered in an MBean server, thus making it accessible through the HTML adaptor. This is particularly useful when debugging, although it does create a security risk. Exposing the USM MIB through SNMP without the MBean server, however, is not insecure.

Users can also be configured into an agent by means of an ASCII text file that acts as an initial configuration template for all agents created.

2.7.6.1 SNMPv3 Authentication and Privacy

Inside SNMP domains, every SNMP entity is issued a unique identifier, the *engine ID*. Java DMK 5.1 provide a set of classes to allow you to generate engine IDs based on, amongst other identifiers, host names, internet protocol (IP) addresses, port numbers and Internet assigned numbers authority (IANA) numbers.

There are two types of SNMP entity:

- *Authoritative* entities
- *Nonauthoritative* entities

Authoritative agent entities receive `get`, `set`, `getnext`, and `getbulk` requests and send traps. Nonauthoritative agents send informs.

Authoritative manager entities receive informs. Nonauthoritative managers send `get`, `set`, `getnext` and `getbulk` requests and informs, and receive traps. The engine ID and the number of times the engine has booted can be stored and persisted in the SNMPv3 security file, so that the timeliness of the incoming requests can be verified.

Under SNMPv3 there are three levels of security:

- *No security:* Unsecured SNMP requests
- *Authenticated requests:* Confirmation of the sender's identity and of the timeliness of the request, with the content of the request visible to the network
- *Authenticated and encrypted requests:* Authentication, with the content of the request encrypted

Managers and agents are both configured with a *username*, allowing the manager specific access to that agent. The username has an associated password. Both the agent and the manager sides must be configured according to the desired security policy. For requests to be authenticated, the manager and the agent must share knowledge of the authentication password associated with the username. For requests to be encrypted, the manager and the agent must additionally share knowledge of the privacy password associated with the username.

Unsecured SNMP Requests

When an agent receives a request from a manager, it checks its LCD. If the user is found in the LCD, the request is granted. No timeliness checking is performed, and the content of the request is not encrypted.

Authenticated Requests

The agent checks the identity of the originator of the request as previously described and then checks the timeliness of the request to ensure that it has not been delayed or intercepted for improper purposes. To monitor the timeliness of the arrival of requests, both manager and agent maintain synchronized clocks, and the manager's local notion of the authoritative engine's time of sending is included in the request. If the difference between the time of sending included in the request and the time of receipt recorded by the agent exceeds 150 seconds, then the request is not considered timely and is rejected.

Once the timeliness of the request has been confirmed, the request is authenticated using either of the HMAC-MD5 or HMAC-SHA protocols. These protocols check that the message digest included in the message matches the one computed locally in the receiving agent.

Authenticated and Encrypted Requests

If privacy has been activated, the content of the request is encrypted, using the DES encryption protocol provided by the Java cryptography extension (JCE) from JDK 1.4. The secure hash algorithm (SHA) and MD5 encryption protocols provided in JDK 1.2 are also used. The requests are decrypted and forwarded once the identity of the sender and the timeliness of the request have been established.

Error Messages

If any of the preceding checks fail, one of the following errors will be generated:

`unknownUser`	Unregistered user
`unknownEngineID`	Unregistered SNMP entity
`encryptionFailed`	Encryption error
`unsupportedSecurityLevel`	Unsupported security level
`authentificationFailed`	Password error
`notInTimeWindow`	Timeliness error

Note – You can optionally implement alternative authentication and encryption algorithms. You cannot, however, plug in customized security or access control models in Java Dynamic Management Kit 5.1, although this will be possible in future versions.

2.7.6.2 SNMPv3 Access Control

SNMPv3 access control differs from the access control defined by versions 1 and 2, in that it is based on contexts and user names, rather than on IP addresses and community strings. The configuration for SNMPv3 access control is located in a text file, called the user ACL file. See the *Java Dynamic Management Kit 5.1 Tutorial* for information about the user ACL file and how to configure it.

When managers send a requests to an agent, the agent authenticates and, if necessary, decrypts the request, as explained earlier. It then passes the request through SNMP context-checking filters to determine whether it is authorized.

2.7.6.3 SNMPv3 Security Configuration

The configuration for SNMPv3 user-based security is located in a text file, called the security file. Each SNMP engine has its own security file. See the *Java Dynamic Management Kit 5.1 Tutorial* for information about the user security file and how to configure it.

You can view examples of security files at:

installDir`/SUNWjdmk/5.1/examples/current/Snmp`

CHAPTER 3

Development Process

This chapter outlines the main tasks in developing management solutions using the Java Dynamic Management Kit (Java DMK).

This chapter is concerned mostly with design issues in the development process. For an explanation of how to write the code of management applications, see the programming examples in the *Java Dynamic Management Kit 5.1 Tutorial*.

The tasks are described in the following sections:

- "3.1 Instrumenting Resources" on page 64
- "3.2 Designing an Agent Application" on page 65
- "3.3 Designing a Management Application" on page 66

Figure 3–1 summarizes these tasks, from crafting MBeans in your factory to deploying them through the web.

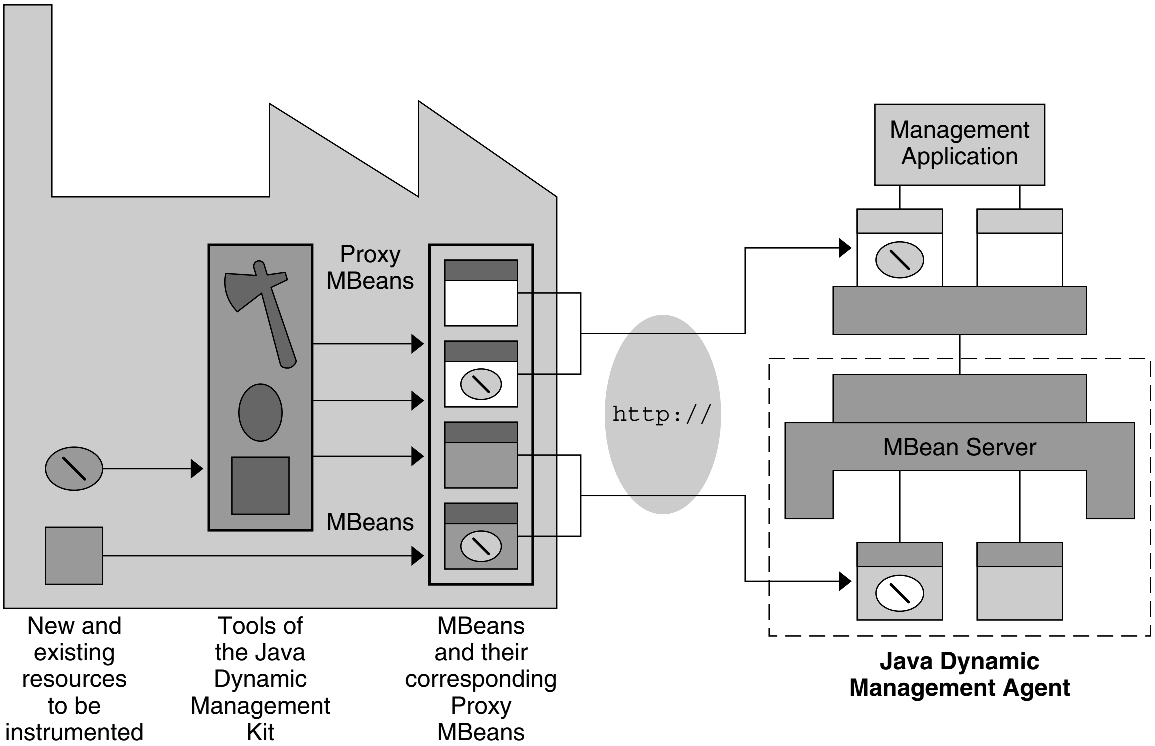
Proxy MBeans
MBeans
http://
Management Application
MBean Server
Java Dynamic Management Agent
New and existing resources to be instrumented
Tools of the Java Dynamic Management Kit
MBeans and their corresponding Proxy MBeans

FIGURE 3–1 Development Process

3.1 Instrumenting Resources

MBeans conform to the JMX specification, which standardizes the representation of the MBean's management interface. Therefore, the first task in the development process is to define the management interface of your resources.

If you are creating new resources, you must determine the granularity of the information about that resource. How many attributes need to be exposed for management? What operations will be useful when the resource is deployed? When should the resource send notifications? The answers to these questions determine the granularity of your MBean's management interface.

Consider an MBean that represents a printer. If your MBean is exposed to end users, it might need only to expose a state attribute, *ready* or *offline*, and perhaps an operation such as *switch paper trays*. However, if your MBean is intended for remote servicing, it must contain much more information. Operators need to know such information as the total print count, the toner level, and the location of a paper jam, and they might want to run self-diagnostics.

Sometimes resources are already manageable through another system. In this case you need only to translate their existing management interfaces into an MBean. Because the JMX architecture is rich, you can usually improve the existing management interface in the translation. Some operations might not be needed because they can be replaced by an agent service. New attributes might be added now that they can be computed dynamically.

As more vendors adopt the JMX specification, resources will be supplied with their instrumentation. Your task will then be to understand the management interface that is provided and to integrate the MBean classes into your application. In this case you will be integrating MBeans from various sources and ensuring that they interact as expected.

3.2 Designing an Agent Application

Given the set of resources you want to manage, you need only to register their corresponding MBeans in an agent, and they become manageable. However, designing an effective agent is more complex.

When designing your agents, you must keep in mind the nature of the management application that will access them. You must strike a balance between services that unburden your clients and making of your agent application too complex.

The simplest agent is one that contains an MBean server and a connector or protocol adaptor. The class for this agent can be written in 10 lines of code, yet this agent is fully manageable. Through the one communication component, a manager can instantiate agent services and dynamically load new resources. The minimalist agent can grow to contain as many MBeans as its memory can hold.

At the other extreme, your entire management solution could be located in the agent. All the policies and all resources you need could be managed locally. This application can become overburdened with its management tasks and does not take advantage of distributed management logic. You need to decide between how much management logic can be performed locally and how much is distributed across your whole management solution.

The functionality of your agents is most often determined by their environment. Some agents might be limited by their host machine. When memory or processing power is limited, an agent can be expected only to expose its MBeans and perhaps run a monitoring service.

An agent in a more powerful machine has the liberty to run more services and handle more MBeans. For example, the agent at the top of a cascading hierarchy might establish relations between MBeans in all the subagents. Desktop machines and workstations can easily handle agents with thousands of MBeans.

The hierarchical model is very appropriate, because management logic and power are concentrated toward the top of the hierarchy. The information from many small devices becomes concentrated on a few large servers where the management consoles are located. In between are medium-sized agents that perform some management tasks, such as filtering errors and computing averages across their subagents.

3.3 Designing a Management Application

This section focuses on developing a management application in the Java programming language. Java applications access agents through connectors which preserve the JMX technology-based architecture. All management requests are available through the connectors, making the communication layer transparent.

Beyond the specifics of establishing connections, accessing MBeans, and using proxies, there are more general programming issues to consider when implementing a management application.

Without going into the details, a list of features that managers might need to implement is given here. A full treatment of these topics would fill several books and several of these issues will probably remain research topics for years to come:

- Optimizing communications by dynamically configuring the connectors
- Deploying new services and upgrading agents dynamically
- Establishing and managing a hierarchy of agents
- Implementing lookup services to allow connector clients to find connector servers
- Cascading management requests through hierarchies of agents
- Handling errors and exceptions
- Recovery from crashes
- Total security

Do not let this list of complex issues scare you away. Not all of these features are needed by all managers. Only the largest management applications would implement full solutions to any one of these issues.

The modularity of the JMX architecture lets you start with a basic manager that is only concerned with accessing resources in an agent. As your needs evolve you can explore solutions to the issues listed above.

In parallel to the programming issues, there two major design issues to consider when developing a management application: the flow of information, and the specificity of the solution.

3.3.1 Defining Input and Output

A management application serves three purposes: to access resources in order to give or receive information, to perform some operation on this information, and to expose the result to others. The operation that a manager performs on its information might be some form of computation, a concentration of the data, or simply a translation from one representation to another.

For example, a manager for a network might collect bandwidth data from routers and calculate averages that are available through some API. The manager also monitors all data for abnormal values and triggers a notification when they occur. These could arguably be the tasks of a smart agent, but let us suppose it is an intermediate manger for very simple agents in the routers.

Now consider a second example: a graphical user interface for managing a pool of printers. Agents in the printers signal whenever there is an error, the manager reads other parameters to determine whether the problem is serious and displays a color-coded icon of the printer: red if the printer needs servicing, orange if it is only a paper problem, and green if the printer is now back online.

In both cases, the applications can have much more functionality, but each function can be broken down into its three facets. By identifying what data needs to be collected, how it needs to be processed and how it needs to be exposed, you can determine the agents that need to be accessed, the algorithms that need to be implemented, and the format of the output.

3.3.2 Specific Versus Generic

Another design choice is whether you need a specific manager or a generic management solution. The two examples above are applications designed for a specific task. Their inputs are known, their agents are listed in address tables, and they are programmed to provide a specific output for given inputs.

A generic management solution is much more complex. It takes advantage of all dynamic features in the JMX architecture. Agents and their resources are not known ahead of time, data formats are unknowable and the output is at best a set of guidelines. Generic managers do not implement a task, they implement a system for integrating new tasks.

Let us extend our printer management system to perform some generic management. First, we set a guideline of only managing printers whose agents contain discovery responders. That way, we can detect when printers are plugged in, we can connect to their agents, and we can add them to the management console automatically. Then we make a space in our user interface for a custom printer interface. If the printer's agent has a resource called `HTMLserver`, we will load the data from this server into the screen frame reserved for this printer.

Users of this management system can now install a server-enabled printer, and it will be managed automatically when it is plugged into the network. Of course, this system is only viable if you advertise the ways in which it is generic, so that printer manufacturers are encouraged to add Java dynamic management agents to their products.

Generic management systems are complex and perhaps difficult to design, but they are definitely in the range of possibilities offered through the JMX architecture and the Java Dynamic Management Kit.

CHAPTER **4**

Legacy Connectors and Related Features

Several features from previous releases of Java Dynamic Management Kit (Java DMK) have been deprecated in version 5.1. This is mostly due to the support of the Java Management Extensions specification (JMX) version 1.2 and the JMX Remote API that were added in Java DMK 5.1. Although these legacy features are now marked as “deprecated”, they have been retained for reasons of backwards compatibility.

4.1 Legacy Connectors

Java DMK 5.1 includes three legacy connectors, in addition to the standard connectors described in “2.3.1 Connectors” on page 34. The legacy connectors are deprecated in favor of the standard ones.

The legacy connector protocols are based on RMI, HTTP, and HTTP/S. The HTTP and HTTP/S connectors are identical except for the security details of connection establishment. For more information on security for legacy connectors, see “4.4 Security Mechanisms for Legacy Connectors” on page 74.

A legacy connector is composed of two parts:

- A connector server, which interacts with the MBean server in an agent
- A connector client, which exposes a manager-side interface that is identical to the MBean server interface

Therefore, a Java application that instantiates a connector client can perform all management operations that are available through the agent’s MBean server.

In the client-server model, it is the connector client that initiates all connections and all management requests. An agent is identified by an address that contains the agent's hostname and port number. The target agent must contain an active connector server for the desired protocol. The address object is protocol-specific and can contain additional information needed for a given protocol.

The connector client uses this address to establish a connection with its corresponding connector server. A connector client can establish only one connection at a time. This implies that a manager instantiates one connector client for each agent it contacts. The management application must wait for the connector to be online, meaning that a connection is established and ready to send requests.

Management applications can then invoke one of the methods of the connector client to issue a request. These methods have parameters that define the object name of the MBean and the attribute or operation name to which the request applies. If the request has a response, it will be returned to the caller.

A connector hides all the details of the protocol encoding from the Java applications. Agent and manager exchange management requests and responses based on the JMX architecture. The underlying encoding is hidden and is not accessible to the applications.

All legacy connectors provide the same remote API, which frees management applications from protocol dependencies. The API provided by legacy connectors is similar to that provided by standard connectors, but they are not interchangeable. In particular, the address format used by legacy connectors is different from that used by standard connectors.

4.1.1 Wrapping of 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.

4.1.2 Monitoring Legacy Connectors Using the Heartbeat Mechanism

All connectors provided in the Java DMK implement a heartbeat mechanism. This is true for both standard connectors and legacy connectors. This section describes the heartbeat mechanism used for legacy RMI, HTTP, and HTTPS connections. For information on the heartbeat mechanism used for standard connectors, see "2.3.1.3 Monitoring Standard Connectors Using the Heartbeat Mechanism" on page 36.

The heartbeat enables both the agent and manager applications to detect when a connection is lost, either because the communication channel is interrupted or because one of the applications has been stopped.

The connector client and connector server components exchange heartbeat messages periodically. When a heartbeat is not returned or an expected heartbeat is not received, both components begin a retry and timeout period. If the connection is not reestablished, both the connector client and the connector server free the resources allocated for that connection.

The heartbeat mechanism is only configurable on the manager side, the connector server simply replies to heartbeats. The manager application can set the retry policy as determined by the heartbeat period and the number of retries. The manager application can also register for heartbeat notifications that are sent whenever a connection is established, retried, reestablished, lost, or terminated.

4.1.3 Generating Proxies for Legacy Connectors

This section outlines how to generate proxies for MBeans accessed through legacy RMI, HTTP and HTTPS connectors.

A proxy MBean is an object that represents a specific MBean instance and that makes it easier to access that MBean. A management application instantiates a proxy so that it has a simple handle on a registered MBean, instead of needing to access the MBean server.

The manager can access MBeans by invoking the methods of their proxy object. The proxy formulates the corresponding management request to the MBean server. The operations are those that are possible on an MBean:

- Getting or setting attributes
- Invoking operations
- Registering or unregistering for notifications

Figure 2–1 shows management components interacting with an MBean through a proxy.

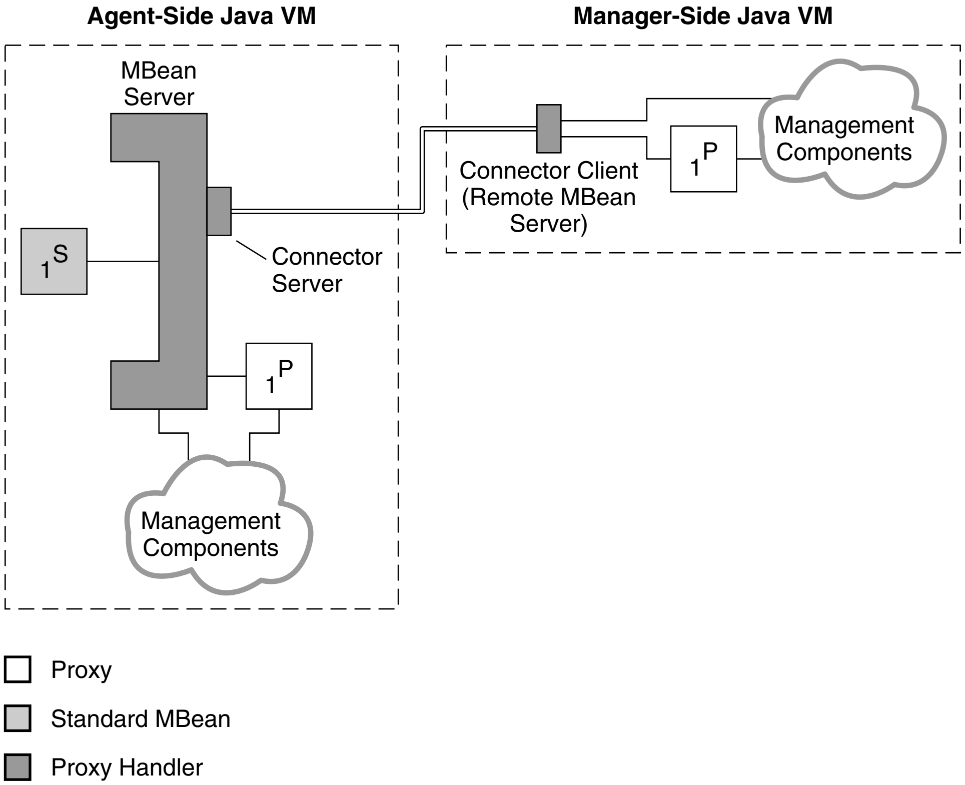

FIGURE 4–1 Binding Proxy MBeans to Local and Remote Servers

Figure 4–1 also shows that proxies can be instantiated either locally in the agent or remotely in the manager. Since the MBean server and the connector client have the same API, management requests to either of them are identical. This creates a symmetry so that the same management components can be instantiated either in the agent or in the manager application. This feature contributes to the scalability of Java dynamic management applications.

A standard proxy is generated from a standard MBean using the `proxygen` tool, supplied with the Java DMK. The resulting class then needs to be loaded wherever the proxy will be instantiated. Generic proxies provide less of an abstraction but do not need to be generated. They are part of the Java DMK libraries and are thus always available.

Note – The `proxygen` tool must only be used to create proxies for MBeans accessed through legacy Java DMK connectors. It is not for use with standard connectors.

4.2 Generating Proxy MBeans

Generating proxy objects for your MBeans is an optional step that depends on the design of your management application. As discussed earlier in this guide, a proxy is an object that represents an MBean in a remote agent. The manager accesses an MBean by performing operations on the proxy MBean.

Proxy objects simplify the design of your management application because they provide an abstraction of remote resources. Your architecture can assume that resources are local because they appear to be, even if they are not. Of course, proxies have greater response times than local resources, but the difference is usually negligible.

Using proxies also simplifies the code of your application. Through the connector client, the proxy object handles all communication details. Your code invokes a method that returns a value, and the complete mechanism of performing the remote management request is hidden. This object-oriented design of having a local object represent a remote resource is fully in the spirit of the Java programming language.

Assuming that a management application has already established the connection to an agent, the overhead of a proxy object is minimal, both in terms of resource usage and required setup. However, it is common sense to instantiate proxies only for resources that will be accessed often or that are long-lived.

The method used to generate proxies has changed in Java DMK 5.1. The proxygen tool is now marked as deprecated. Use the proxygen tool only if you require proxies for legacy remote method invocation (RMI), hypertext transfer protocol (HTTP), and secure HTTP (HTTP/S) connectors. For new RMI, RMI/IIOP and Java Management Extensions messaging protocol (JMXMP) connectors that comply with the JMX 1.2 and JMX Remote API 1.0 specifications, you can generate a proxy object at runtime, given just its Java interface, using the dynamic proxies defined by the Java 2 Platform, Standard Edition (J2SE) `java.lang.reflect.Proxy` interface. These dynamic proxies cannot be used with the legacy connectors.

In an advanced management solution where resources are discovered only at runtime, the proxy class can be loaded dynamically in the manager. For example, the resource might expose an attribute called `ProxyURL` from which a class loader can retrieve the proxy object.

4.3 Cascading Service for Legacy Connectors

The cascading service for legacy connectors is an MBean that establishes a connection to one subagent. For each of the subagent's MBeans, the cascading service instantiates a *mirror* MBean that is registered in the master agent. The cascading service also defines a filter and query expression that together determine the set of MBeans in the subagent that is mirrored.

The mirror MBean is a sort of proxy that is specific to the cascading service. A mirror MBean exposes the same management interface as its corresponding MBean. All attributes, operations, and notifications can be accessed through the mirror MBean, which forwards all management requests through the cascading service to the corresponding MBean in the subagent.

You can define hierarchies of agents of arbitrary complexity and depth. Because mirrored MBeans are registered MBeans, they can be mirrored again in a higher master agent. The cascading service is dynamic, meaning that mirrored MBeans are added or removed as MBeans in a subagent are added or removed.

The cascading mechanism works only in one direction. While master agents can manipulate objects in their subagents, subagents have no visibility of their master agent and are not even aware of their master agent.

The cascading service relies on connector components internally and can therefore be used with the RMI, HTTP, or HTTPS protocols. The user specifies the protocol and the subagent's address when configuring the cascading service.

4.4 Security Mechanisms for Legacy Connectors

The legacy RMI and HTTP-based connectors implemented different security mechanisms to those implemented by the new RMI and JMXMP connectors. These legacy mechanisms are now deprecated, and are presented here for reasons of backwards compatibility.

4.4.1 Password Protection

Password-based protection restricts client access to agent applications. All HTTP-based communication provides login- and password- based authentication, as does the SNMP protocol adaptor.

Password protection can be used to associate managers with a set of privileges that determine access right to agents. The user is free to implement whatever access policy is needed on top of the password authentication mechanism. The SNMP protocols also provide password protection to agent applications. See "2.7.5 SNMPv1 and SNMPv2 Security" on page 57 and "2.7.6 SNMPv3 Security" on page 58.

4.4.1.1 HTTP Connectors

Both HTTP and HTTPS connectors provide login and password-based authentication. The server component contains the list of allowed login identifiers and their passwords. Management applications must specify the login and password information in the address object when establishing a connection.

If the list of recognized clients is empty, the default behavior is to perform no authentication and grant access to all clients.

4.4.1.2 HTML Protocol Adaptor

Because the HTML protocol adaptor relies on HTTP messaging, it also implements password protection. The agent application specifies the list of allowed login identifiers and their passwords when creating the HTML adaptor. When password protection is enabled in HTML, the web browser usually displays a dialog box for users to enter their login and passwords.

In general, the security mechanisms of a protocol adaptor depend on the security features of the underlying protocol. The ability to use security mechanisms also depends on the functionality of the management console. If your web browser does not support the password dialog, you cannot access a password-protected HTML adaptor.

4.4.2 Context Checking

Whereas password protection grants all-or-nothing access, context checking enables the agent application to filter each management request individually. Context checking can be associated with password protection to provide multiple levels of security.

All management requests that arrive through a connector or HTML protocol adaptor are inspected by the agent application to determine if they are authorized. The management application filters requests based on the type of request, the MBean for which they are intended, or the values that are provided in the operation.

For example, context checking could allow an agent to implement a read-only policy that refuses attribute set operations, all operation invocation, and does not allow MBean registration or unregistration. A more selective filter could just ensure that the agent cannot be disconnected: it would disallow MBean unregistrations, `stop` operations, and invocations that contain null parameters, but only when applied to connector servers or protocol adaptor MBeans.

In addition, requests through connector clients can be filtered by an *operation context* field, which could be a password or any other identifying data. The context object is provided by the management application, and it will be sent to the connector server along with each request. The agent can verify this context and potentially reject the request if the context is considered invalid or inappropriate for the operation.

To make this context checking possible, the agent provides:

- *Stackable MBean server objects* – You can insert your own code to perform context checking and filtering between the communication component and the MBean server.
- *Thread contexts* – Your code can retrieve the remote application's context object that is stored in the thread object that handles the request. The context is an arbitrary object that your code can use to determine whether or not to allow the request.

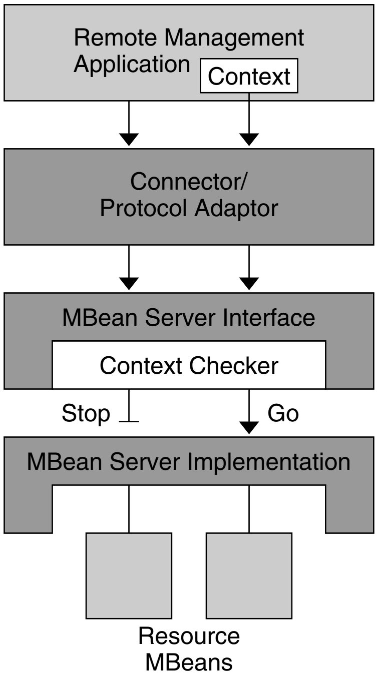
Remote Management Application
Context
Connector/ Protocol Adaptor
MBean Server Interface
Context Checker
Stop
Go
MBean Server Implementation
Resource MBeans

FIGURE 4–2 Context Checking Using Stackable MBean Server Objects

In Figure 4–2, a context checker object has been inserted between the connector and the MBean server. Because a context checker object implements the `MBeanServer` interface, the connector interacts with it in exactly the same way as it did with the MBean server. This stacked object retains a reference to the real MBean server, to which it forwards all requests that are allowed. The context checker can also perform any other action, such as log all filtered requests and trigger a notification when an invalid request is received.

For security reasons, only the agent application can insert or remove stackable MBean server objects. This operation is not exposed to management applications, which cannot even detect whether requests are being filtered. However, the context checker might respond with an exception message that explains why a request was denied.

4.4.3 Data Encryption

The last link in the security chain is the integrity of data that is exchanged between agent and managers. Two issues need to be considered simultaneously:

Authentication: Both agent and manager must be certain of the other's identity.

Privacy: The data of a management request should be tamper-proof and undecipherable to nontrusted parties.

These issues are usually resolved by a combination of electronic signatures and data encryption. Again, the implementation is protocol-dependent.

The SNMP protocols also provide password protection to agent applications. See "2.7.5 SNMPv1 and SNMPv2 Security" on page 57 and "2.7.6 SNMPv3 Security" on page 58.

The HTTPS connector enables Java managers to access a Java dynamic management agent using HTTP over Secure Socket Layer (SSL). SSL security is implemented in the Java 2 platform. The HTTP/SSL connector provides identity authentication based on the Challenge-Response Authentication Mechanism using MD5 (CRAM-MD5). The HTTPS connector server requires client identification by default.

The behavior of the HTTP/SSL connector is governed by the particular SSL implementation used in your applications. For data encryption, the default cipher suites of the SSL implementation are used. The SSL implementation must be compliant with the SSL Standard Extension API.

4.4.4 Secure Dynamic Loading

The m-let service downloads Java classes from arbitrary locations over the network. If you want to do so, you can enable code signing to ensure that only trusted classes can be downloaded. Secure loading relies on code signing.

On the Java 2 platform, the java.lang.SecurityManager property determines if code signing is enforced. When this security is enabled, again only class files signed by a trusted party will be loaded. On the Java 2 platform, users invoke the keytool, jarsigner, and policytool utilities to define their security policies.

4.5 Tracing

The implementation of Java DMK 5.1 has changed with regard to the production of tracing and debugging information:

- The source code of Java DMK 5.1 now uses directly the java.util.logging API to emit debug and trace messages.
- The com.sun.jdmk.TraceManager class, com.sun.jdmk.trace.* classes and com.sun.jdmk.Trace class (already deprecated in Java DMK 5.0) are all deprecated in Java DMK 5.1.
- Backward compatibility with Java DMK 5.0 system properties is preserved: -DLEVEL_* and -DINFO_* flags still activate the traces.

When java.util.logging is not present, it is still possible to activate the traces by specifying -DLEVEL_DEBUG or -DLEVEL_TRACE on the Java command line.

Index