



Getting Started with the Java Dynamic Management Kit 5.0

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

Part No: 816-4176-10
June 2002

Copyright 2002 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in this product. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and other countries.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook, AnswerBook2, Java, Java Coffee Cup logo, JDK, JavaBeans, JDBC, Java Community Process, JavaScript, J2SE, JMX and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

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.

Copyright 2002 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans ce produit. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à <http://www.sun.com/patents> et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats - Unis et les autres pays.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, docs.sun.com, AnswerBook, AnswerBook2, Java, le logo Java Coffee Cup, JDK, JavaBeans, JDBC, Java Community Process, JavaScript, J2SE, JMX et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE 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.



020610@4333



Contents

Preface	7
1 Java Dynamic Management Kit Overview	11
Introduction to the Java DMK	12
Why Use Java Dynamic Management Technology?	12
What Is the Java DMK ?	13
How Do I Develop a Java Dynamic Management Solution?	14
Key Concepts	17
Benefits of a Java Dynamic Management Solution	18
Simplified Design and Development	19
Protocol Independence	20
Dynamic Extensibility and Scalability	20
SNMPv3 Security	21
SNMP Master Agent	21
Overview of the Product Documentation	22
Online HTML Files	22
Printable Documents	22
Programming Examples	22
Javadoc Utility	23
2 Architectural Components	25
MBeans	25
Standard MBeans	26
Dynamic MBeans	27
Model MBeans	27

Open MBeans	28
The MBean Server	29
Communication Components	29
Connectors	30
Protocol Adaptors	33
The Notification Model	34
Local Notification Listeners	34
Remote Notification Listeners	35
Agent Services	36
Querying and Filtering	37
Dynamic Loading	37
Monitoring	38
Scheduling	39
Cascading	39
Discovering Agents	40
Defining Relations	41
Security	43
Password Protection	43
Context Checking	44
Data Encryption	45
Secure Dynamic Loading	46
The SNMP Toolkit	46
Developing an SNMP Agent	47
SNMP MIB Compiler – mibgen	48
SNMP Manager API	48
SNMPv1 and SNMPv2 Security	49
SNMPv3 Security	50
3 Development Process	55
Instrumenting Resources	56
Designing an Agent Application	57
Generating Proxy MBeans	58
Designing a Management Application	59
Defining Input and Output	59
Specific Versus Generic	60
Index	61

Figures

FIGURE 1-1	Key Concepts of the Java DMK	17
FIGURE 2-1	Binding Proxy MBeans to Local and Remote Servers	31
FIGURE 2-2	Adding Local Listeners on the Agent Side	34
FIGURE 2-3	Adding Remote Listeners on the Manager Side	35
FIGURE 2-4	The Discovery Search Service	40
FIGURE 2-5	The Relation Model Defined by the JMX Specification	42
FIGURE 2-6	Context Checking Using Stackable MBean Server Objects	45
FIGURE 3-1	Development Process	55

Preface

The Java™ Dynamic Management Kit (DMK) 5.0 provides a set of Java classes and tools for developing management solutions. This product conforms to the Java Management Extensions (the JMX™ Specification), v1.1 Maintenance Release, which defines a three-level architecture:

- Resource instrumentation
- Dynamic agents
- Remote management applications

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

The *Getting Started with the Java Dynamic Management Kit 5.0* guide presents the architecture of the Java DMK introducing the key components of the product and the development process for management applications.

Who Should Use This Book

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

You should be familiar with Java programming and the JavaBeans™ component model. You should also be familiar with the JMX specification and the simple network management protocol (SNMP).

This book is not intended to be an exhaustive reference. Management tutorials demonstrating each of the management levels and how they interact are covered in the *Java Dynamic Management Kit 5.0 Tutorial*. The complete Javadoc™ API definitions are provided in the online documentation package.

How This Book Is Organized

This book explains the key concepts of the Java DMK, introduces the main components of the product, provides an overview of the development process and outlines the tools you need to use the 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 the Java DMK, you must have a complete installation of the product on your machine. Refer to the *Java Dynamic Management Kit 5.0 Installation Guide* for instructions on how to install the product components and configure your environment.

After familiarizing yourself with 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 5.0 Tutorial*, you will 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.

Related Documentation

The following books are part of the product documentation set:

TABLE P-1 Related Documentation

Book Title	Part Number
<i>Java Dynamic Management Kit 5.0 Tools Reference</i>	816-4177-10
<i>Java Dynamic Management Kit 5.0 Tutorial</i>	816-4178-10
<i>Java Dynamic Management Kit 5.0 Installation Guide</i>	816-4179-10

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

Operating Environment	Homepage Location
Solaris	<i>installDir</i> /SUNWjdkm/jdkm5.0/index.html
Windows 2000	<i>installDir</i> \SUNWjdkm\jdkm5.0\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:

- /opt on the Solaris platform
- 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 the JMX specification. The specification document, *Java Management Extensions Instrumentation and Agent Specification, v1.1* (Maintenance Release, March 2002), is provided in the product documentation package, under the file name `jmx_instr_agent.pdf`.

Accessing Sun Documentation

You can view and print a broad selection of Sun™ documentation, including localized versions, at:

<http://www.sun.com/documentation/>

You can also purchase printed copies of select Sun documentation from iUniverse, the Sun documentation provider, at:

<http://corppub.iuniverse.com/marketplace/sun/>

Typographic Conventions

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

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name%</code> you have mail.
AaBbCc123	What you type, contrasted with on-screen computer output	<code>machine-name% su</code> Password:
<i>AaBbCc123</i>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
<i>AaBbCc123</i>	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You must be <i>root</i> 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	<code>machine-name%</code>
C shell superuser prompt	<code>machine-name#</code>
Bourne shell and Korn shell prompt	<code>\$</code>
Bourne shell and Korn shell superuser prompt	<code>#</code>

Java Dynamic Management Kit Overview

The Java Dynamic Management Kit (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 the Java Management Extensions (the JMX specification), the product provides a framework for the management of Java objects through Java technology-based applications.

The Java DMK provides a complete architecture for designing distributed management systems. A Java technology-based solution can embed management intelligence into your agents, provide an abstraction of your communication layer, and 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:

- “Introduction to the Java DMK ” on page 12 gives an overview of the product architecture and functionality.
- “Key Concepts” on page 17 describes the main components of the Java Dynamic Management Kit.
- “Benefits of a Java Dynamic Management Solution” on page 18 highlights the benefits of the product for designers and developers.
- “Overview of the Product Documentation” on page 22 describes the product documentation delivered with the Java DMK.

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 do I develop a Java dynamic management solution?

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.

Why Use Java Dynamic Management Technology?

Old Way – Network management is usually performed by large, centralized management applications. These management applications monitor and modify their network by tightly controlling their agents. The agents act as relays for the network resources they represent, translating commands and collecting raw data and status information. Agents are usually situated in or near the network elements they control, which means that these agents are limited in nature. They usually contain little management intelligence and can perform only basic network management operations.

New Way – 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.

Old Way – From a wider perspective, existing management systems for networks and applications are implemented with diverse protocols and technologies. Developers must choose a single management technology for a portion of the target market. In some cases, developers might need to implement multiple management technologies to provide more complete coverage of their potential markets. Due to the limitations of both approaches, vendors frequently choose not to implement any management technology.

New Way – The interface to resources is standardized, meaning that device vendors and application developers can finally agree: they can use any technology they 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 they can control resources through standard interfaces, they are dynamically interchangeable. In order to upgrade the capabilities of a smart agent,

new services can be downloaded and plugged in dynamically when they become available. 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:

- Compliance to the JMX specification, for managing Java objects through Java applications, as developed through the Java Community Process^(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 and that can 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, meaning that you can download them through simple web pages in the same way as Java technology-based applets.

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 will be 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.

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 programmatic components of the Java DMK include:

- A management architecture – The architecture is a conforming implementation of the JMX specification API, both the instrumentation and the agent levels.
- Communication modules – The Java DMK defines APIs for accessing JMX agent remotely. The product includes communication modules based on the RMI, HTTP, and HTTPS protocols. It also 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, cascading agent hierarchies, dynamic agent discovery, and components for implementing security mechanisms.
- SNMP APIs – Applications that rely on the SNMP APIs can integrate into existing network management systems and help these systems migrate towards a more dynamic, service-based approach to network management.
- SNMPv3 compliance – Java DMK 5.0 implements SNMPv3 security to protect your systems from outside interference.

The development tools are implemented as two standalone applications:

- `proxygen` – This tool is a proxy object generator that simplifies the development of Java technology-based management applications. Proxy objects make the communication layer transparent to the manager application.
- `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.

Finally, the Java DMK contains 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 all functionality of the product.
- A tutorial that explains the programming examples and a reference guide for the standalone tools.
- Both online HTML and printable file formats for all documents.
- The complete JMX specifications document: the Java DMK 5.0 implements the JMX Specification v1.1.

How Do I Develop a Java Dynamic Management Solution?

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 Java DMK extends the agent services and defines the distributed management features for accessing agents remotely. A distributed management solution relies on all three levels.

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 may 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.

Therefore, 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.

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 it contains. The agent centralizes all requests, dispatches them to the target MBeans, and returns any responses. The agent handles the communication issues involved with receiving and sending data, so that the MBeans don't have to.

The central component of an agent is the *MBean server*. It is a registry for MBean instances, and it 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 some MBean attribute. The manager instantiates the monitoring service MBean, configures the threshold, and registers to receive the alarms that may 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:

- 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.

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, HTTP/TCP and HTTP/SSL protocols are provided, all with the same API for interchangeability.

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 HTTPS, 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 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.

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 the following protocols: RMI, HTTP, or HTTPS. It 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.

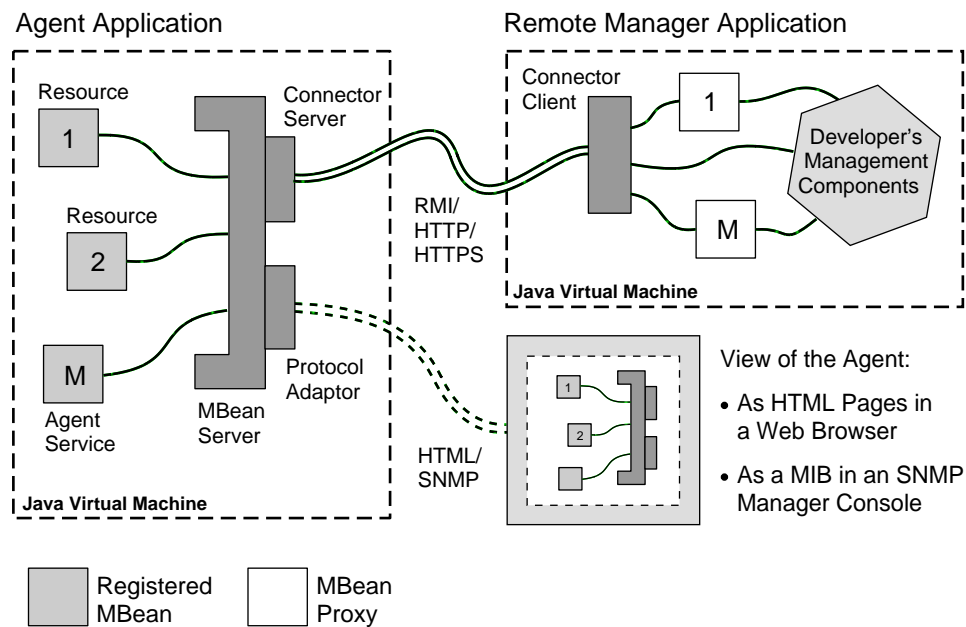


FIGURE 1-1 Key Concepts of the Java DMK

The remote manager is a Java application running on a distant host. It 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, it 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. There is also an administration page for creating or removing MBeans from the MBean server.

Each of these concepts is further defined in Chapter 2.

Benefits of a Java Dynamic Management Solution

To summarize, the benefits of the Java DMK include:

- 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

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 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 exposed for management.
- Any of the provided communication components can be used to respond to remote requests, and you can develop new adaptors and 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 it is needed. You can have a prototype running after your first day of development, because of the programming examples provided in the product.

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 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 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 5.0 supports multihome interfaces, allowing you to work in environments where there are multiple network protocols available. The multihome interface service means that Java DMK 5.0 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.

Dynamic Extensibility and Scalability

By definition, all agents and manager applications developed with the Java DMK 5.0 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 unregistered with the MBean server in an agent while it 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 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 they 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.

SNMPv3 Security

The Java DMK 5.0 extends the SNMP support of previous Java DMK releases to include the SNMPv3 protocol. This means that Java DMK benefits from the security and administration services offered by SNMPv3.

The 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 “Security” on page 43.

SNMP Master Agent

The SNMP support in the Java DMK 5.0 allows you to build a *master agent* that groups together several SNMP subagents and exports their information through a single point of access. The master agent performs two main functions:

- It 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. It can also be a proxy, representing a remote SNMP agent, to which the request will be forwarded
- It converts requests from the SNMP version supported by the manager into the version supported by the subagent. It also converts the responses back, and converts the traps sent by the sub agent into the version used by the manager.

Overview of the Product Documentation

The Java DMK product includes both printable and online documentation, as well as an exhaustive set of programming examples.

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:

In the Solaris operating environment: `file:/opt/SUNWjdmk/jdmk5.0/index.html`

On the Windows 2000 operating environment:
`file:/Program Files/SUNWjdmk/jdmk5.0/index.html`

The page contains links to all the product documentation supplied online with the Java DMK, including:

- *Getting Started with the Java Dynamic Management Kit 5.0*
- *Java Dynamic Management Kit 5.0 Tools Reference*
- *Java Dynamic Management Kit 5.0 Tutorial*
- *Java Dynamic Management Kit 5.0 Installation Guide*
- The *Java Management Extensions 1.1* specification document.

Printable Documents

Complete PDF versions of the books listed in the preceding section are supplied on the CD-ROM of the software. These files are located in the `docs` directory at the common root of 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, regardless of the default paper size. The text area on each page fits on all standard paper sizes.

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 at:

`installDir/SUNWjdmk/jdmk5.0/examples`

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 5.0 Tutorial*.

The `examples` directory also contains the `JdmkProxyMBeans` subdirectory, which provides proxy MBeans for all of the Java DMK components that support them. These are generated by the `proxygen` tool and must be compiled in the normal way before use. You can use them to provide proxy objects in your manager applications for the agent-side service MBeans.

Javadoc Utility

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

The Java DMK Javadoc is found in the following location after installation:

`installDir/SUNWjdmk/jdmk5.0/docs/locale/C/api/index.html`

Architectural Components

This chapter presents the components of the Java Dynamic Management Kit (DMK) and explains how you can use them in a complete management solution.

This chapter contains the following sections:

- “MBeans” on page 25 describes the three ways to instrument a resource so that it is manageable.
- “The MBean Server” on page 29 explains how a JMX agent exposes the MBeans it contains.
- “Communication Components” on page 29 presents the components that establish connections between agents and managers.
- “The Notification Model” on page 34 explains how resources and agents can signal events and how events are forwarded to remote listeners.
- “Agent Services” on page 36 briefly explains each agent service.
- “Security” on page 43 describes the security features built into the communication components of the Java DMK.
- “The SNMP Toolkit” on page 46 explains how to develop Java applications for SNMP agents and managers.

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 “Communication Components” on page 29.

You can make resource accessible through Java DMK agents even if they are not instrumented as MBeans, by using MBean Interceptors. See “MBean Interceptors” on page 32 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 MBeans:

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

Standard MBeans

Standard MBeans are Java objects that conform to certain design patterns 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 are unlikely to change often.

Dynamic MBeans

Dynamic MBeans do not have getter and setter methods for each attribute and operation. Instead, they 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 them 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 it determines its own management interface based on existing conditions when it is instantiated.

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. It 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.

Open MBeans

Open MBeans allow management applications and the users to understand and use new managed objects as they 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, there is no corresponding open interface 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. It is a dynamic MBean with the same behavior and all of the same functionality. It 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).

The MBean Server

The MBean server is a registry for JMX manageable resources, which it exposes to management requests. It 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, it becomes visible to management applications and 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 other objects in the agent application itself
- 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
- Discovering and publicizing 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. It 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, 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.

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

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 managed by remote management applications.

An agent can contain any number of connectors or protocol adaptors, enabling it 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.

Connectors

Connectors establish a point-to-point connection between an agent and a management application, each running in a separate Java virtual machine. The Java DMK provides connectors for the HTTP/TCP, HTTP/SSL, and RMI protocols. Every connector provides the same remote API, which frees management applications from any protocol dependency.

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.

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.

Connector Heartbeat

All connectors provided in the Java DMK implement a heartbeat mechanism. 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.

Proxy MBeans

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

Because dynamic MBeans expose their management interface only at runtime, they cannot have a specific proxy MBean. Instead they have a generic proxy whose methods have an extra parameter to specify the attribute or operation name. Figure 2-1 shows management components interacting with both standard and dynamic MBeans through both standard and generic proxies. Notice that a generic proxy can also represent a standard MBean.

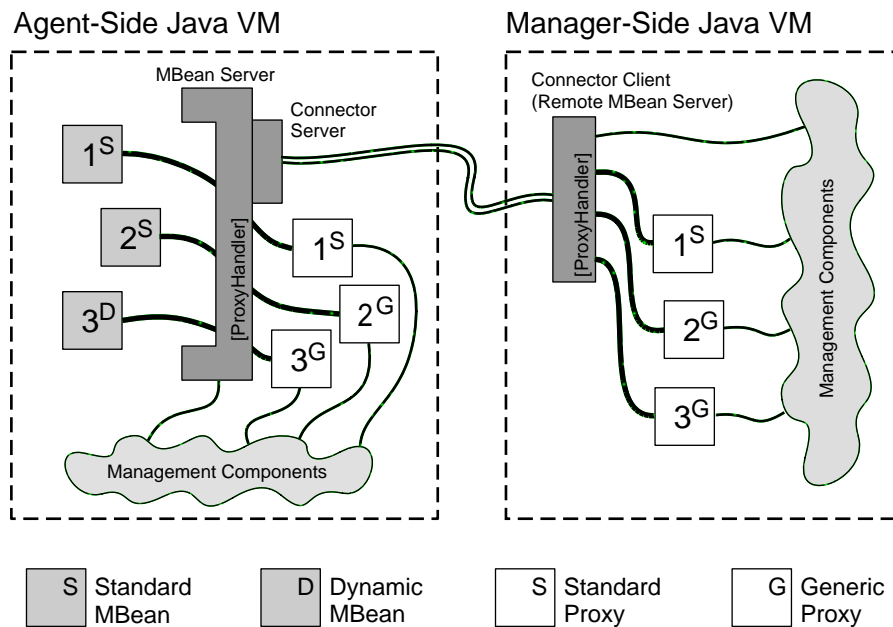


FIGURE 2-1 Binding Proxy MBeans to Local and Remote Servers

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.

A standard proxy is generated from a standard MBean using the `proxygen` compiler, 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.

MBean 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 *interceptors* enable you to intercept operations on MBeans and handle them arbitrarily. Handling them 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.

Protocol Adaptors

Protocol adaptors have only a server component and provide a view of an agent and its MBeans through a different protocol. They can also translate requests 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 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. There is no guarantee that this mapping is complete or fully accurate. However, specification efforts are currently under way to define fully and standardize the mappings between the JMX architecture and the most widespread management protocols such as SNMP, CORBA, TMN, and CIM/WBEM. For more details, see the Java Community ProcessSM web site at:

<http://java.sun.com/aboutJava/communityprocess/>

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.

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 will receive the same notifications directly from the broadcaster MBean.

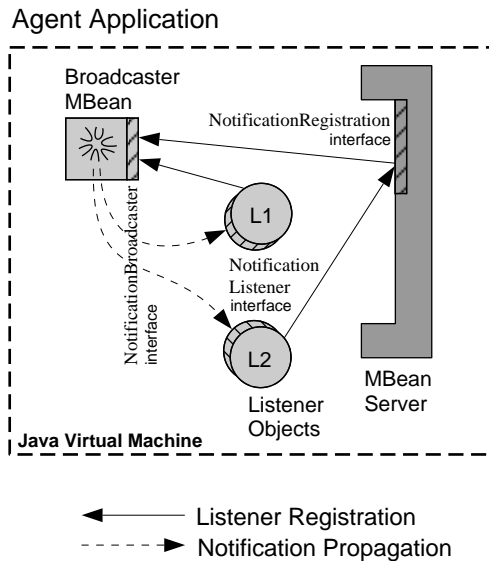


FIGURE 2-2 Adding Local Listeners on the Agent Side

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 place them in a cache buffer, waiting to be sent to the manager application. This enables the connector to avoid saturating the communication layer in case of a burst of notifications.

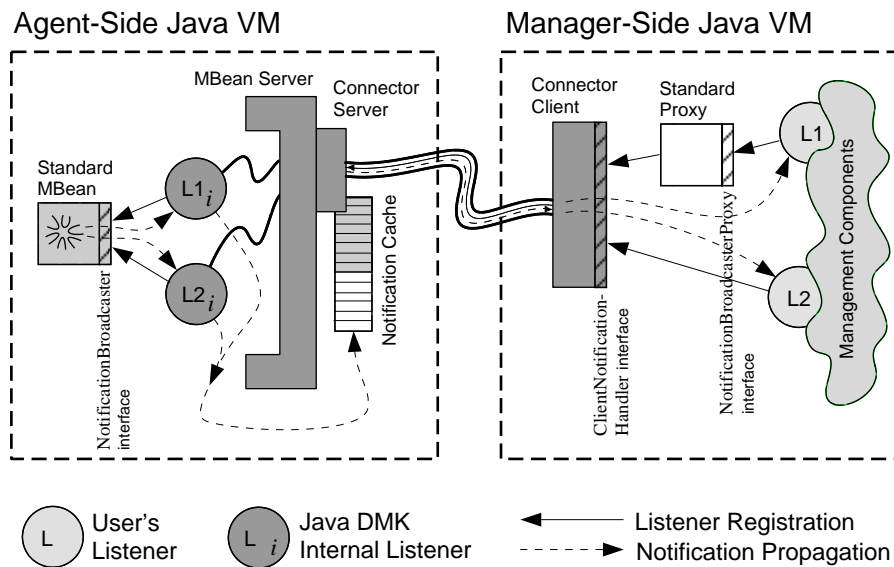


FIGURE 2-3 Adding Remote Listeners on the Manager Side

Notifications either can be pushed from the agent to the connector client as they are received, or they can be pulled periodically at the client's request. The pushing mechanism is the simplest, but the pull mechanism can be used to group notifications and reduce bandwidth usage. In either case the connector client then acts as a broadcaster and sends the notifications to their intended listeners.

The management application configures the forwarding mechanism through the connector client. The manager can choose either push or pull mode, and in pull mode it can set the pull interval and the caching policy. The manager should set these parameters according to the notification emission rate, to avoid saturating the communication layer and yet receive notifications in a timely manner.

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.

Querying and Filtering

Querying and filtering are performed by the MBean server itself, 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 for the purpose of selecting the MBeans on which management operations are performed.

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

- *An object name filter* – This 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* – A query 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.

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 create the MBeans that they represent. The m-let service is defined by the JMX specification and 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, it 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 implement new resources in their agents.

Monitoring

The monitoring service complies with the JMX specification and provides a polling mechanism based on the value of MBean attributes. There are three monitor MBeans, one for counter attributes, another for gauge-like attributes, and a third 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:

- *Counter monitor* – 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 will reset the threshold if the counter rolls over.
- *Gauge monitor* – 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. Using this information, listeners know which MBean triggered an event, and they 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.

Scheduling

The timer service is a notification broadcaster that sends notifications at specific dates and times. This 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 with 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 regarding the timer service, refer to the JMX specification document.

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 remote, 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 registered in their master agent. The master agent hides the physical location of subagents and provides client applications with a centralized access point.

The cascading service 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.

Discovering Agents

With the discovery service you can 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 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

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 containing information about the connectors and protocol adaptor that are available in their agent.

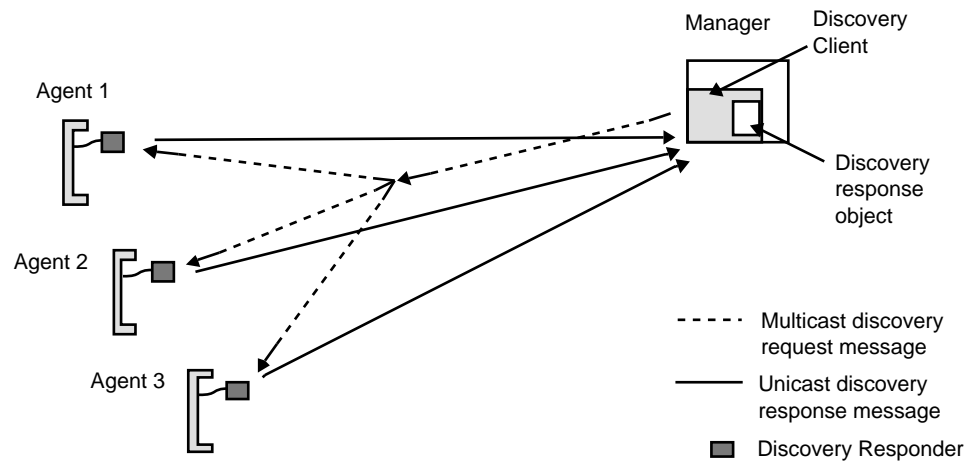


FIGURE 2-4 The Discovery Search Service

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

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.

Defining Relations

The relation service defines and maintains logical relations between registered MBeans. Users define the relation type and establish the relation instance which that associates any number of MBeans. The relation service provides query mechanisms to retrieve MBeans that are related to one another.

In the JMX architecture a relation type is defined by the class and cardinality of MBeans that it associates in named roles. For example, if `Books` and `Owner` are roles, `Books` represents any number of owned books of a given MBean class, and `Owner` is a single book owner of another MBean class. You could define a relation type containing these two roles and call it `Personal Library`. This represents the concept of book ownership.

Figure 2-5 compares this sample relation type to the unified modeling language (UML) model of its corresponding association.

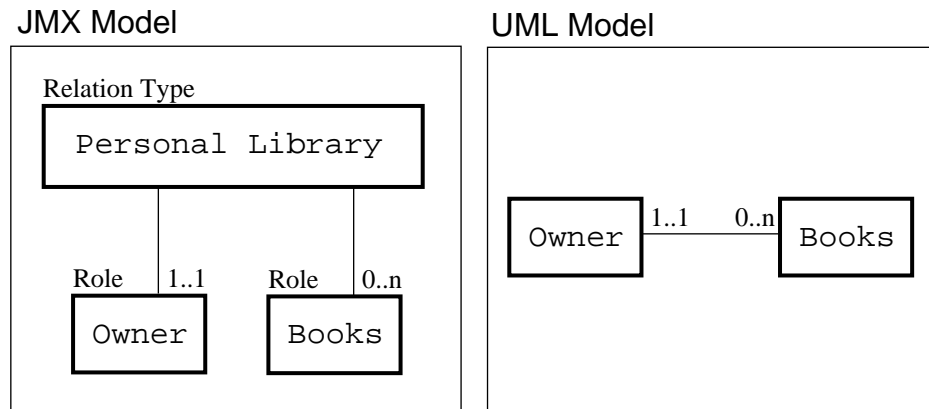


FIGURE 2-5 The Relation Model Defined by the JMX Specification

Through the relation service, users can create relation types and then create, access, and delete instances of a relation. In the preceding example, a management application can add `Book` MBeans to a `Personal Library` relation, or it can replace the MBean in the `Owner` role with another MBean of the same class. 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 they link.

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. For example, if an `Owner` MBean were unregistered, the relation service would remove any `Personal Library` relations where that MBean was the designated owner.

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.

For example, a `Personal Library` relation could be implemented by an MBean with an operation called `Loan`. This operation would search the list of `book` MBeans for a title and implement some mechanism to mark that book as being on loan. And because external relations are MBeans, these extended operations are available to remote management applications.

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, whereas 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.

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

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 “SNMPv1 and SNMPv2 Security” on page 49 and “SNMPv3 Security” on page 50.

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.

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.

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.

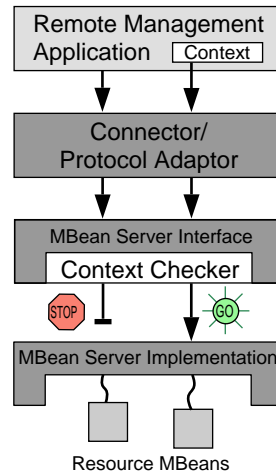


FIGURE 2-6 Context Checking Using Stackable MBean Server Objects

In Figure 2-6, 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.

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 “SNMPv1 and SNMPv2 Security” on page 49 and “SNMPv3 Security” on page 50.

HTTP/SSL

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.

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.

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 5.0 Tools Reference* and the *Java Dynamic Management Kit 5.0 Tutorial*.

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 exist 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 performance of SNMP has also been improved since Java DMK 5.0, by the addition of multithread support in SNMP adaptors and timers.

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.

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.

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.

SNMPv1 and SNMPv2 Security

Because of backward compatibility, Java DMK 5.0 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 5.0.

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.

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.

SNMPv3 Security

The main addition to Java DMK 5.0 provided by SNMPv3 is the possibility of secure SNMP operation. The SNMPv3 security in Java Dynamic Management Kit 5.0 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 5.0 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.

SNMPv3 Authentication and Privacy

Inside SNMP domains, every SNMP entity is issued a unique identifier, the *engine ID*. Java DMK 5.0 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:

<code>unknownUser</code>	Unregistered user
<code>unknownEngineID</code>	Unregistered SNMP entity
<code>encryptionFailed</code>	Encryption error
<code>unsupportedSecurityLevel</code>	Unsupported security level
<code>authenticationFailed</code>	Password error
<code>notInTimeWindow</code>	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 5.0, although this will be possible in future versions.

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 5.0 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.

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 5.0 Tutorial* for information about the user security file and how to configure it.

You can view examples of security files at:

`installDir/SUNWjdmk/jdmk5.0/examples/Snmp`

Development Process

This chapter outlines the main tasks in developing management solutions using the Java Dynamic Management Kit (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 5.0 Tutorial*.

The tasks are described in the following sections:

- “Instrumenting Resources” on page 56
- “Designing an Agent Application” on page 57
- “Generating Proxy MBeans” on page 58, an optional task
- “Designing a Management Application” on page 59

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

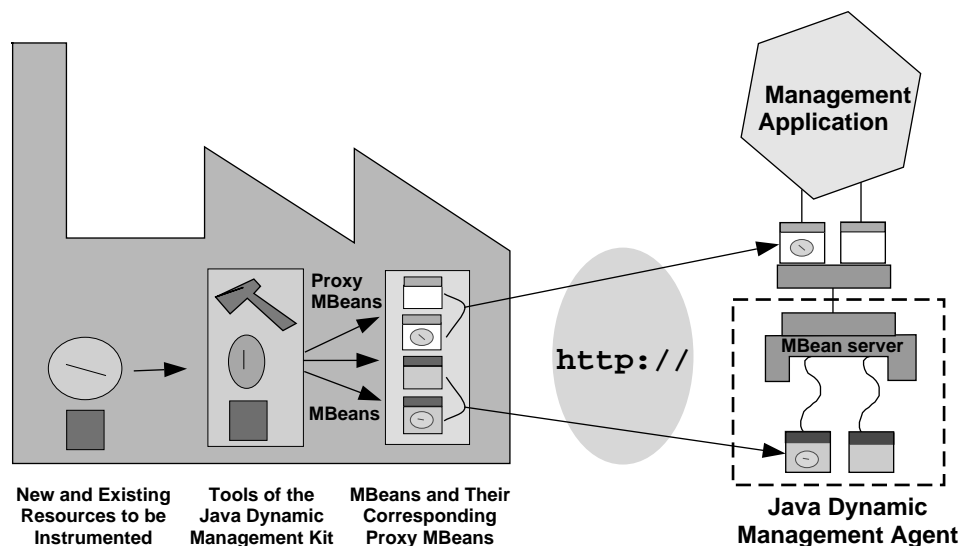


FIGURE 3-1 Development Process

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.

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.

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 development cost of a proxy MBean is also minimal. Standard proxies are fully generated from their corresponding MBean by using the `proxygen` compiler supplied with the Java DMK. Generic proxies are part of the Java dynamic management runtime libraries and need only to be instantiated.

You can use of the `proxygen` tool options to modify the characteristics of the proxies you generate from an MBean. For example, the `read-only` option generates proxies whose setter methods return exceptions. By generating sets of proxies with different characteristics from the same MBean, you can develop a Java manager whose behavior is modified at runtime, depending on which set is available.

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.

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 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
- 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.

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 manager 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 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.

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.

Index

A

- access control
 - SNMPv3, 53
 - SNMPv1 and SNMPv2, 49
- accessing agents remotely, 16
- adaptors, 16
 - HTML, 33
 - limitations, 33
 - protocol, 33
- advantages of the Java DMK, 12, 18
- agent applications, designing, 57
- agent services, 36
 - cascading, 39
 - defining relations, 41
 - discovering agents, 40
 - dynamic loading, 37
 - filtering, 37
 - monitoring attributes, 38
 - querying, 37
 - scheduling notifications, 39
- attributes, monitoring, 38
- authentication
 - HTML protocol adaptor, 44
 - HTTP connectors, 43
 - HTTPS connectors, 46
 - SNMPv3, 51
 - SNMPv1 and SNMPv2, 49

C

- cascading, 39

class

- dynamic loading, 26, 37
- communication components, 29
- connectors, 16, 30
 - heartbeat mechanism, 31

D

- data encryption, 45
- developing a Java dynamic management solution, 14
- development tools
 - generating proxy MBeans, 58
 - mibgen tool, 48
 - proxygen tool, 58
 - SNMP MIB compiler, 48
 - summary, 14
- discovering agents, 40
 - discovery search service, 40
 - discovery support service, 41
- discovery monitor object, 41
- documentation, 22
 - HTML, 22
 - Javadoc, 23
 - printable, 22
 - programming examples, 22
 - summary, 14
- dynamic class loading, 26, 37
 - security, 46
- dynamic MBeans, 27

E

encryption of data, 45
exposing MBeans, 15

F

filtering, 37

G

generating proxy MBeans, 58

H

heartbeat mechanism, connectors, 31
HTML adaptor, 33
HTTP over SSL, 46

I

instrumenting
resources, 15, 25
interceptors, MBeans, 32
introduction to the Java DMK, 12
IPv6, 20

K

key concepts of the Java DMK, 17

M

m-let service, 37
managed beans, 25
management applets, 37
management applications
defining input, 59
defining output, 59
designing, 59
generic managers, 60
specific managers, 60

management interface, defining, 56
management solutions, steps to developing,
55

MBeans, 25
dynamic, 27
interceptors, 32
management interface, 25
mirror, 39
model, 27
monitor, 38
open, 28
proxy, 31
representing in a remote agent, 58
server, 29
standard, 26
types, 26
mibgen tool, 48
model, relation, 41
model MBeans, 27
monitoring attributes, 38

N

notification listeners
adding on the agent side, 34
adding on the manager side, 35
local, 34
remote, 35
notifications
broadcasting, 34
local listeners, 34
model, 34
monitor, 38
remote listeners, 35
scheduling, 39

O

open MBeans, 28

P

password protection, 43

- protection
 - See security mechanisms
- protocol adaptors, 33
 - limitations, 33
- protocols, IPv6, 20
- proxy MBeans, 31
 - binding to local and remote servers, 31
 - generating, 58
- proxygen tool, 58

SNMPv1 and SNMPv2 (Continued)

- encoding, 49
- security, 49
- standard MBeans, 26

T

- timer service, 39

Q

- querying, 37

R

- registry, 29
- relation model, 41
- relation service, 41
- resources
 - instrumenting, 15, 25, 56

S

- scheduling notifications, 39
- security mechanisms, 43
 - context checking, 44
 - data encryption, 45
 - password protection, 43
 - secure dynamic loading, 46
 - SNMPv3, 50
 - SNMPv1 and SNMPv2, 49
 - SNMPv3 security configuration, 53
- server, MBean, 29
- SNMP agents, developing, 47
- SNMP manager API, 48
- SNMP MIB compiler, 48
- SNMP toolkit, 46
- SNMPv3
 - access control, 53
 - authentication, 51
 - security, 50
 - security configuration, 53
- SNMPv1 and SNMPv2
 - access control list (ACL), 49

