Getting Started with the Java Dynamic Management Kit 4.2

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-6630–10
December 2000

Copyright 2000 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more additional patents or pending patent applications in the U.S. or other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of this product or of this documentation 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.

Sun, Sun Microsystems, the Sun Logo, Java, Java Dynamic Management, JMX, JavaBeans, JavaScript, Javadoc, JDK, PersonalJava, Java Community Process, the Java Coffee Cup logo, docs.sun.com, AnswerBook, AnswerBook2, and Solaris are trademarks or registered trademarks 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.

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

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.

Copyright 2000 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, Californie 94303-4900 Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuelle relatants à la technologie incorporée dans le produit décrit par ce document. En particulier, et sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs brevets, ou des applications de brevet en attente, aux Etats-Unis et dans d’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.

Sun, Sun Microsystems, le logo Sun, Java, Java Dynamic Management, JMX, JavaBeans, JavaScript, Javadoc, JDK, PersonalJava, Java Community Process, le logo Java Coffee Cup, docs.sun.com, AnswerBook, AnswerBook2 et Solaris sont des marques de fabrique ou des marques déposées 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’ETAT” ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L’APTITUDE A UNE UTILISATION PARTICULIERE OU A L’ABSENCE DE CONTREFAÇON.

Contents

Preface 5

1. **The Java Dynamic Management™ Kit** 11

Introduction 12

Why Use Java Dynamic Management Technology? 12

What Is the Java Dynamic Management Kit? 13

How Do I Develop a Java Dynamic Management Solution? 14

Key Concepts 17

Advantages of a Java Dynamic Management Solution 18

Simplified Design and Development 18

Protocol Independence 19

Dynamic Extensibility And Scalability 20

2. **Architectural Components** 21

MBeans 21

Standard MBeans 22

Dynamic MBeans 23

Model MBeans 23

The MBean Server 24

Communication Components 25

Connectors 25

Protocol Adaptors 27

The Notification Model 28

Local Notification Listeners 29

Remote Notification Listeners 29

Agent Services 30

Query and Filtering 31

Dynamic Loading 31

Monitoring 32

Scheduling 33

Cascading 33

Discovering Agents 34

Defining Relations 35

Security 37

Password Protection 37

Context Checking 38

Data Encryption 40

Secure Dynamic Loading 41

The SNMP Toolkit 41

Developing An SNMP Agent 41

SNMP MIB Compiler – `mibgen` 42

SNMP Manager API 43

3. The Development Process 45

Instrumenting Resources 46

Designing an Agent Application 46

Generating Proxy MBeans 47

Designing a Management Application 48

Defining Input and Output 49

Specific Versus Generic 49

Preface

The Java Dynamic Management™ Kit provides a set of Java™ classes and tools for developing management solutions. This product conforms to the Java Management extensions (JMX™), v1.0 Final Release, which defines a three-level architecture: resource instrumentation, dynamic agents and 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.

This *Getting Started with the Java Dynamic Management Kit 4.2* Guide presents the architecture of the Java Dynamic Management Kit, 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 Dynamic Management Kit.

Familiarity with Java programming and the JavaBeans™ component model is assumed. Familiarity with the JMX specification is also recommended.

This book is not intended to be an exhaustive reference: management tutorials intended to demonstrate each of the management levels and how they interact are covered in *Java Dynamic Management Kit 4.2 Tutorial*, and the complete Javadoc™ API definitions are provided in the product's online documentation package.

How This Book Is Organized

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

- "What is the Java Dynamic Management Kit?"
- "Architectural Components "
- "The Development Process"

After You Read This Book

In order to build and run the sample programs or use the tool commands provided in the Java Dynamic Management Kit 4.2, you must have a complete installation of the product on your machine. Please refer to the *Java Dynamic Management Kit 4.2 Installation Guide and Release Notes* document for instructions on how to install the product components and configure your environment.

After familiarizing yourself with the concepts of the Java Dynamic Management Kit, you should familiarize yourself with the tools for developing management applications. Then, through the lessons of the tutorial, you will learn how to instrument new or existing resources, write intelligent agent applications and access them from remote managers written in the Java programming language. You are then ready to design and develop your own Java Dynamic Management solution.

The following books are part of the product documentation set:

- *Java Dynamic Management Kit 4.2 Tools Reference*
- *Java Dynamic Management Kit 4.2 Tutorial*

These books are available online after you have installed the documentation package of the Java Dynamic Management Kit 4.2. The online documentation also includes the Javadoc API for the Java packages and classes, including those of the Java Management extensions. Using any web browser, open the homepage corresponding to your platform:

Operating Environment	Homepage Location
Solaris	*installDir*/SUNWjdmk/jdmk4.2/*JDKversion*/index.html
Windows NT	*installDir*\SUNWjdmk\jdmk4.2*JDKversion*\index.html

In these file names, *installDir* refers to the base directory of your Java Dynamic Management Kit installation. In a default installation procedure, *installDir* is:

- `/opt` on the Solaris platform
- `C:\Program Files` on the Windows NT platform

The *JDKversion* is that of the Java Development Kit (JDK™) which you use and which you selected during installation. Its value can be either `1.1` or `1.2`, when used in a directory, filename, or path.

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

Related Books

The Java Dynamic Management Kit relies on the management architecture of the Java Management extensions. The specification document, *Java Management Extensions Instrumentation and Agent Specification, v1.0* (Final Release, July 2000), is provided in the product documentation package, under the filename `jmx_instr_agent.pdf`.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

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

Typographic Conventions

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

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.
`AaBbCc123`	Class or object names, methods, parameters or any other element of the Java programming language	Instantiate the `MyBean` class.
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.

Shell Prompts

The following table shows the default system prompts for the different platforms and shells.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`
Windows NT system prompt	`C:\`

Unless otherwise noted, the command examples in this book use the Korn shell prompt.

CHAPTER 1

The Java Dynamic Management™ Kit

The Java Dynamic Management™ Kit 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 Dynamic Management Kit 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 Dynamic Management Kit 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.

Contents:

- The “Introduction” 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.
- “Advantages of a Java Dynamic Management Solution” on page 18 highlights the benefits of the product for designers and developers.

Introduction

In this section, we answer these fundamental questions about the Java Dynamic Management Kit:

- 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 only perform 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 allows 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 may need to implement multiple management technologies, in order 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 they communicate through a Java Dynamic Management agent, management applications 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 Dynamic Management Kit provides a distributed model that is protocol independent: management applications rely on the API, not on any one protocol.

The Java Dynamic Management Kit brings new solutions to the management domain through:

- Compliance to the Java Management extensions, the specification for managing Java objects through Java applications, as developed through the Java Community Process.
- A single suite of components that provides uniform instrumentation for managing systems, applications, and networks, and that allows universal access to these resources.
- A flexible architecture that distributes the management load and which 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 for you 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 can also be network-based, downloaded 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 Dynamic Management Kit a wide domain of application as it integrates the current and future management standards.

What Is the Java Dynamic Management Kit?

The Java Dynamic Management Kit is a Java API with 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 Dynamic Management Kit 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 Dynamic Management Kit 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 which 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.

The development tools are implemented as two standalone applications:

- `proxygen` – This tool is a proxy object generator which 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 MIB (management information base) represents the management interface of resources in an SNMP agent, and `mibgen` generates the corresponding Java objects.

Finally, the Java Dynamic Management Kit 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 which 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 Dynamic Management Kit 4.2 implements the JMX Specification v1.0, Final Release.

How Do I Develop a Java Dynamic Management Solution?

The intstrumentation 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 Dynamic Management Kit 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 wish 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*: this 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 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 may provide the MBeans that plug into their customer's existing agents. Management solution integrators may develop the MBeans for resources which have not been previously instrumented. Because MBeans follow the JMX specification, they can be instantiated in any JMX–compliant agent. This makes them 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 wish 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 may ask for the description of an MBean's management interface, in order to know 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 only accessible 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: it 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, it 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: scheduling events, monitoring attributes, establishing and enforcing relations, discovering other agents, creating subagent hierarchies, and downloading of new MBean objects. You may 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 Dynamic Management Kit allows you to access agents and their resources very 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 simply 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 may 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 Java-based. For example, the HTML adaptor represents MBeans as web pages that can be viewed in any Web browser. The SNMP adaptor exposes special MBeans that represent an SNMP MIB and responds to requests in both SNMP v1 and v2 protocols.

All connectors and adaptors are themselves implemented as MBeans. Management applications may therefore create, configure and remove communication resources dynamically, according to network conditions or available protocols. Of course, the

agent application may also implement security features that prevent unwanted operations on the communicator MBeans.

The flexibility of communicator MBeans and the availability of connectors for multiple protocols makes 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 may also create your own connectors and adaptors to accommodate proprietary protocols and future management needs.

Key Concepts

The diagram in Figure 1–1 gives a visual representation the key concepts of the Java Dynamic Management Kit 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 for each of the ports through which it wishes to communicate.

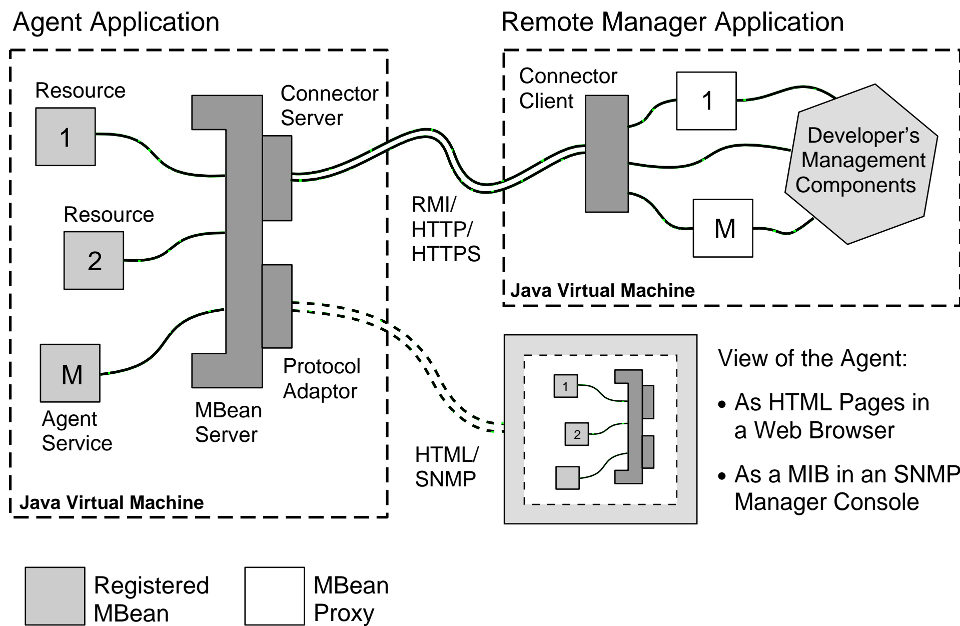


Figure 1–1 Key Concepts of the Java Dynamic Management Kit

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 may call the proxy objects to invoke an operation on the first resource and configure the monitoring service to poll the second resource.

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

We will further define and describe each of these concepts in Chapter 2: "Architectural Components".

Advantages of a Java Dynamic Management Solution

To summarize, the benefits of the Java Dynamic Management Kit include:

- Simplified design and development of instrumentation, smart agents and remote managers.
- Deployment flexibility through protocol independence and SNMP compatibility.
- Dynamic extensibility and scalability.

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, each level being 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 only need 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 JMX–compliant agent.
- 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 JMX-compliant MBean may 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.
- No need to implement basic management tasks, these are done in the agent by the agent services.

At all three levels, the modularity also means the simple designs may be implemented rapidly, and then additional functionality may be added as it is needed. You can have a prototype running after your first day of development, thanks to 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 an agent uses for communicating with external applications. All interactions with MBeans are necessarily handled by the MBean server and thus defined by the APIs of the Java Management extensions.

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

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

Dynamic Extensibility And Scalability

By definition, all agents and manager applications developed with the Java Dynamic Management Kit are extensible and scalable. The library of agent services is always available: managers may instantiate new services when they are 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 may be registered and unregistered with the MBean server in an agent while it is running. This is useful to represent application resources which may come and go on a given host. The scalability allows 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, it is possible to 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.

Finally, JMX conformance insures that all JMX-compatible components can be incorporated into Java Dynamic Management agents, whether they are manageable resources, new services, or new communication components.

CHAPTER 2

Architectural Components

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

Contents:

- "MBeans" on page 21 describes the three ways to instrument a resource so that it is manageable.
- "The MBean Server" on page 24 describes how a JMX agent exposes the MBeans it contains.
- "Communication Components" on page 25 presents the components which establish connections between agents and managers.
- "The Notification Model" on page 28 explains how resources and agents can signal events and how events are forwarded to remote listeners.
- "Agent Services" on page 30 briefly explains each of the agent services.
- "Security" on page 37 describes the security features built into the communication components of the Java Dynamic Management Kit.
- "The SNMP Toolkit" on page 41 describes 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 Managed 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 will be managed. For example, attribute patterns enable you to make the distinction 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 *must* be represented as an MBean. Both the agent application and remote managers may access MBeans in an agent. MBeans can generate notification events which are sent to all local or remote listeners. For more information regarding managing MBeans remotely, please refer to "Communication Components" on page 25.

MBeans can also be downloaded from a Web server and plugged 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 PC's or Web phones.

There are three types of MBeans:

- Standard MBeans
- Dynamic MBeans
- Model MBeans, which are an extension of dynamic 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 will need to 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 which are defined in advance and 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 defined by the `DynamicMBean` interface.

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

Dynamic MBeans provide a simple way to wrap existing Java object which do not follow the design patterns for standard MBeans. They can also be implemented to access non-Java technology based resources 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 may be quite 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 which 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 it 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 which 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 which configures a model MBean may modify its management interface at any time. It may 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 different MBean types. However, if a manager understands model MBeans, it will be able to obtain additional management information about the managed resource. This informations includes behavioral and runtime metadata that is specific to model MBeans.

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.

Registering a resource's MBean makes it visible to management applications and exposes it to management requests. The MBean server makes no distinction between the types of MBeans: standard, dynamic and model MBeans are all 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 deregistering 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 either be assigned 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 (JVM), 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 allow agents to be accessed and managed by remote management applications.

An agent may contain any number of connectors or protocol adaptors, enabling it to be managed simultaneously by several managers, through different protocols. It is up to the agent application to coordinate all of the port numbers 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 VM. The Java Dynamic Management Kit 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 which instantiates a connector client may perform all management operations which are available through the agent's MBean server.

In the client-server model, it is the connector client which initiates all connections and all management request. An agent is identified by an address which 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 may 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 only establish one connection at a time. This implies that a manager instantiates one connector client for each agent it wishes to contact. 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 which 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 manger exchange management requests and responses based on the JMX architecture. The underlying encoding is hidden and not accessible to the applications.

Connector Heartbeat

All connectors provided in the Java Dynamic Management Kit implement a heartbeat mechanism. The heartbeat allows 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 will 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 may also register for heartbeat notifications which are sent whenever a connection is established, retrying, reestablished, lost, or terminated.

Proxy MBeans

A proxy MBean is an object which represents a specific MBean instance and 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 may 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 which are possible on an MBean:

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

Because dynamic MBeans only expose their management interface 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. The following diagram shows management components interacting with both standard and dynamic MBeans through both standard and generic proxies. Notice that a generic proxy may also represent a standard MBean.

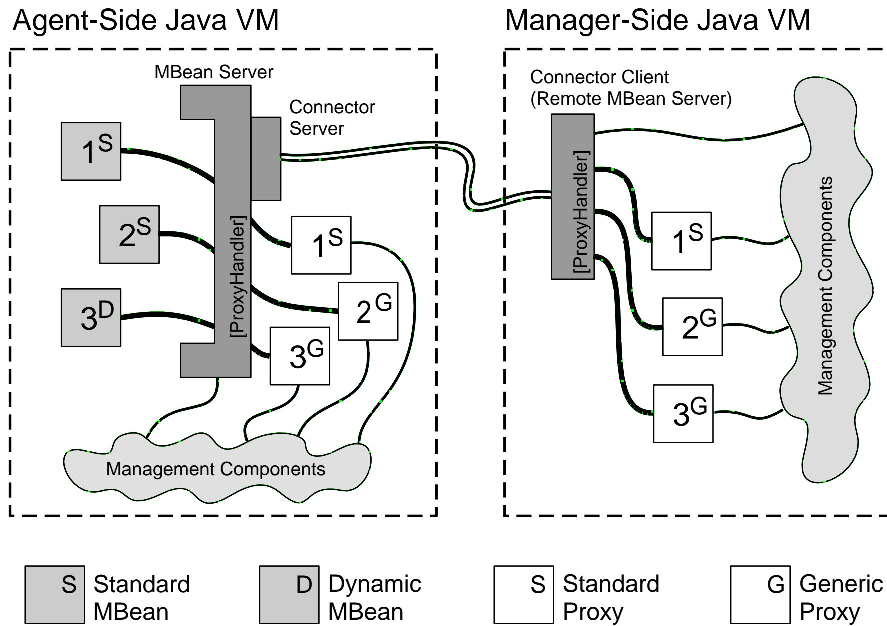


Figure 2–1 Binding Proxy MBeans to Local and Remote Servers

This diagram also shows that proxies may 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 request to either of them are identical. This creates a symmetry that allows the same management components to 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 by using the `proxygen` compiler, supplied with the Java Dynamic Management Kit. 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 Dynamic Management Kit libraries and are thus always available.

Protocol Adaptors

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

For example, the Java Dynamic Management Kit provides an HTML adaptor which presents the agent and its MBeans as HTML pages viewable in any web browser. Because the HTML protocol is text based, only data types which have a string

representation may 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, as some requests may rely on unsupported data types.
- Notifications from a broadcaster MBean may not be supported.
- A given protocol adaptor may require private data structures or helper MBeans in order to respond to requests.

The SNMP adaptor provided in the Java Dynamic Management Kit is limited by the constraints of SNMP. The richness of the JMX architecture cannot be translated into SNMP, but all of 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 underway to fully define and standardize the mappings between the JMX architecture and the most widespread management protocols such as SNMP, CORBA, and TMN. Please see the Java Community Process℠ web site (`http://java.sun.com/aboutJava/communityprocess/`) for more details.

The Notification Model

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

The JMX notification model enables a listener to register only once and still receive all different notifications that an MBean may broadcast. A listener object may 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 may also be added to an MBean identified by its symbolic name.

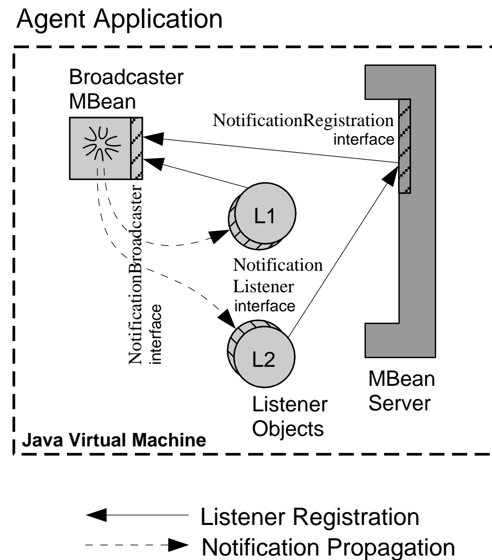


Figure 2–2 Adding Local Listeners on the Agent Side

In the figure above, 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.

Remote Notification Listeners

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

Listeners do not need to be aware of the fact 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.

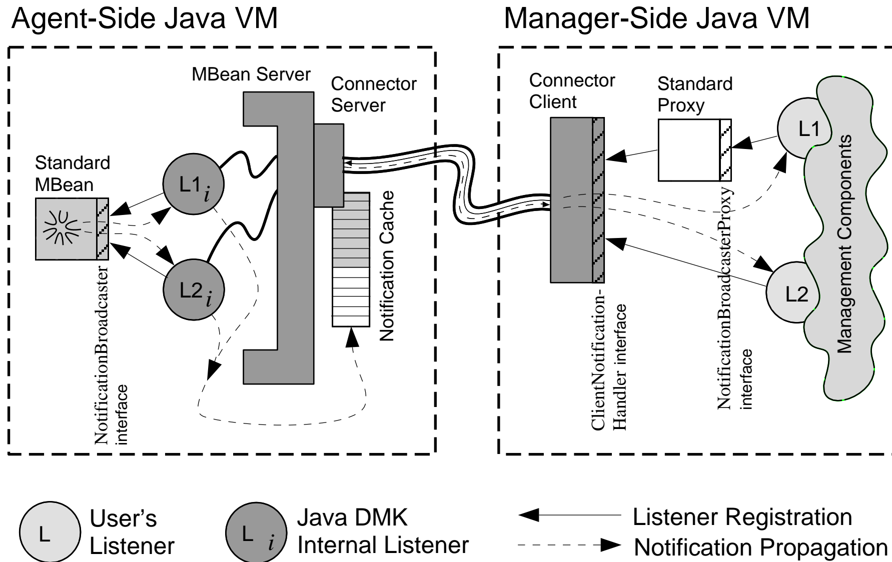


Figure 2–3 Adding Remote Listeners on the Manager Side

As shown in the figure above, the connector components implement a complex mechanism for registering remote listeners and forwarding notifications. Since notifications are based on the Java event model, broadcasters cannot send notifications outside their Java VM. So the connector server instantiates local listeners who receive all notifications an place them in an cache buffer, waiting to be sent to the manager application. The buffer allows the connector to avoid saturating the communication layer in case of a burst of notifications.

Notifications can either 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, in order 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 Dynamic Management Kit supplies a set of agent services.

These services are simply implemented as MBeans which perform some operations on the other MBeans in an agent. Here we list all of the provided agent services and explain each one briefly.

Query and Filtering

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

Usually, a management application will perform a query in order to find the MBeans which will be the target of its management requests. To select MBeans, applications may specify:

- An object name filter – This is an incomplete object name which the MBean server will try to match with the object names of all registered MBeans. All MBeans whose name matches the filter pattern will be selected. Filters may contain wildcards to select sets of MBeans, or a filter may be a complete object name which 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 of the MBeans 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 name contains "`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 will load 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>` which specifies information about the MBeans to be loaded. It resembles the `<APPLET>` tag except that it will only load MBean classes. The tag contains information for downloading the class, such as the classname and the location of its class file. You may also specify any arguments to the constructor used to instantiate the MBean.

The m-let service loads a URL which 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, a application could make new MBean classes available at some 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 equalling 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 of the monitors has specific settings for the type of the observed attribute:

- Counter monitor – Observes an attribute of integer type 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 or floating point 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, the value which triggered the event, as well as the previous value for comparison. Using this information, listeners know which MBean triggered an event, and they don't need to access the MBean before taking the appropriate action.

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

Scheduling

The timer service is a notification broadcaster that send notifications at specific dates and times. This provides a scheduling mechanism that may be used to trigger actions in the listeners. Timer notifications may 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 may 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 which identifies the notification to its listeners. The timer service uses a single Java thread to trigger all notifications at their designated time.

The timer service may be stopped to prevent it from sending notifications. When it is started again, notifications which 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 may be registered in an agent and configured by remote applications. However, the timer MBean may also be used as a stand-alone object in any application that needs a simple scheduling service.

For more information regarding the timer service, please refer to the JMX specification document.

Cascading

Cascading is the term used to describe a hierarchy of agents, where management requests may be 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 which 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 which together determine the set of MBeans in the subagent which are 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 may be accessed through the mirror MBean. The mirror MBean forwards all management requests through the cascading service to the corresponding MBean in the subagent.

You may 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 only works 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 connectors components internally and may therefore be used with any of the following protocols: RMI, HTTP, or HTTPS. The user specifies the protocol and the subagent's address when configuring the cascading service.

Discovering Agents

The discovery service enables you to discover Java Dynamic Management agents in a network. Only agents that have a discovery responder registered in their MBean server can be discovered using 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. The agents must have a `DiscoveryResponder` registered in their MBean server in order to be found by the discovery service. All discovery responder which receive the discovery request send a response containing information about the connectors and protocol adaptor which are available in their agent.

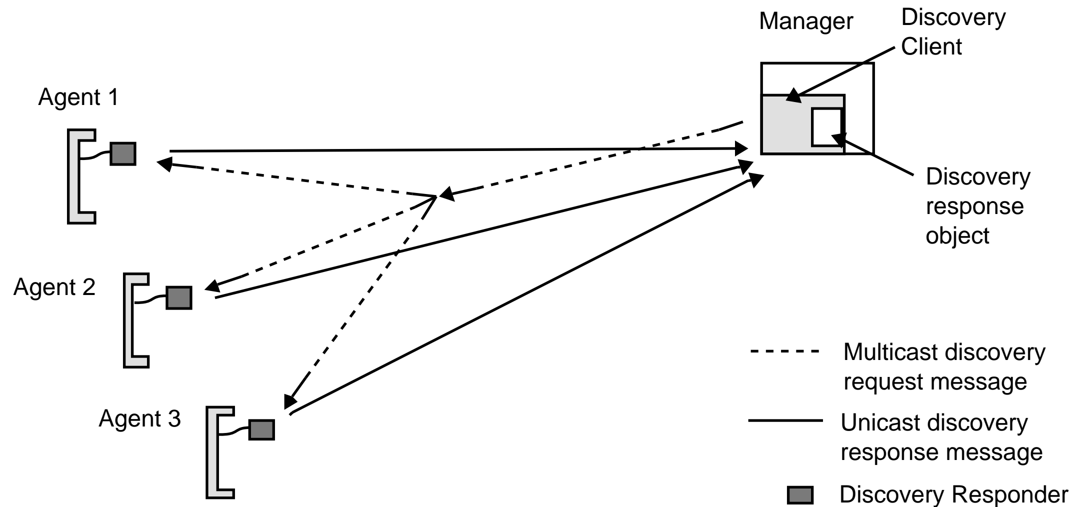


Figure 2–4 The Discovery Search Service

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 monitor 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 may know when agents become available or unavailable. The discovery support message for an agent that is being started also contains the list of 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 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, we can say that `Books` and

Owner are roles. Books represents any number of owned books of a given MBean class, and Owner is a single book owner of another MBean class. We might define a relation type containing these two roles and call it Personal Library: it represents the concept of book ownership.

The following diagram represents this sample relation type, as compared to the UML modeling of its corresponding association.

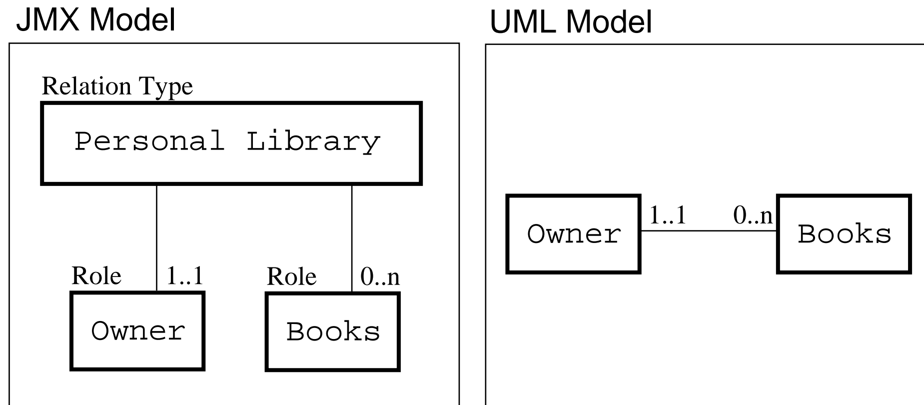


Figure 2–5 The Relation Model Defined by the JMX Specification

Through the relation service, users may create relation types and then create, access, and delete instances of a relation. In our example, a management application may add Book MBeans to a Personal Library relation, or it may 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 may be accessed from a remote application.

The relation service is notified when MBeans in a relation are deregistered, and it verifies that any relation involving that MBean still has the required cardinality. For example, if an Owner MBean were deregistered, 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 may also be a separate MBean object that fulfills certain requirements. The user instantiates and registers these MBeans, insures 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 may 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 Dynamic Management Kit 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; 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 Dynamic Management Kit.

Password Protection

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

Password protection can be used to associate managers with a set of privileges which determine access right to agents. The user is free to implement whatever access policy is needed on top of the password authentication mechanism.

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 password. 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, no authentication is performed and access is granted to all clients; this is the default behavior.

HTML Protocol Adaptor

Since 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 password when creating the HTML adaptor. When password protection in enabled in HTML, the web browser usually displays a dialog box for users to enter their login and password.

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

SNMP Access Control

SNMP defines an access control mechanism similar to password authentication. Lists of authorized manager hostnames are defined in an *access control list* (ACL) stored in an ACL file on the agent side. There are no passwords, but logical community names may be associated with authorized managers to define sets of allowed operations.

The SNMP adaptor will perform access control if an ACL file is defined. Because SNMP is a connectionless protocol, the manager host and community are verified with every incoming request. By default, the file is not loaded and any SNMP manager may send requests.

The ACL file is the default access control mechanism in the SNMP protocol adaptor. However, you may 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.

Context Checking

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

All management requests that arrive through a connector or protocol adaptor may be inspected by the agent application to determine if they are allowed. The management application may filter 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 which refuses attribute set operations, all operation invocation, and doesn't allow MBean registration or deregistration. A more selective filter could just insure that the agent cannot be disconnected: it would disallow MBean deregistrations, `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 may 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 which is stored in the thread object which handles the request. The context is an arbitrary object that your code can use to determine whether or not to allow the request.

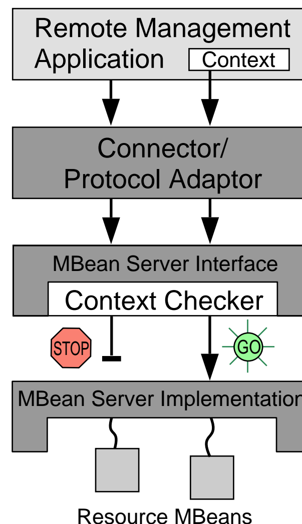


Figure 2–6 Context Checking Using Stackable MBean Server Objects

In the figure above, 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 will retain a reference to the real MBean server, to which it will forward all requests that are allowed. The context checker may 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 may insert or remove stackable MBean server objects. This operation is not exposed to management applications, who cannot even know if requests are being filtered. However, the context checker may choose to respond with an exception whose message explains why a request was denied.

Data Encryption

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

- Identification – 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 untrusted parties.

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

SNMP Encoding

SNMP requests follow standardized encoding rules 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 Dynamic Management Kit provide access to the byte encoding of these packets.

Your applications may customize the encoding and decoding of SNMP requests:

- On the manager side, after the request is translated into bytes, your encoding may add signature strings and then perform encryption.
- On the agent side, the bytes may 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 may insert code to filter requests based on any of these criteria.

In order to implement a secure SNMP management solution, you need to coordinate the security policy between the manager encoding and the agent decoding. However, SNMP request filtering may be performed unilaterally by the agent, to allow requests from unknown managers yet still be able to reject unauthorized operations.

HTTP/SSL

The HTTPS connector enables Java managers to access a Java Dynamic Management agent using HTTP over SSL (Secure Socket Layer). SSL security is implemented in the Java 2 platform. The HTTP/SSL connector provides identity authentication based on 'CRAM-MD5' (Challenge-Response Authentication Mechanism using 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.

The Java Dynamic Management Kit is compliant with the Java Secure Socket Extension 1.0 (JSSE) API. JSSE provides an API framework and reference implementation for security protocols.

Secure Dynamic Loading

The m-let service downloads Java classes from arbitrary locations over the network. If you wish to do so, you may enable code signing to insure that only trusted classes may be downloaded. Secure loading relies on code signing which differs between the JDK 1.1 and Java 2 platforms.

On a JDK 1.1 platform, the m-let service may be instantiated in secure mode to enforce code signing. The m-let service will then only load `.jar` files that have been signed by a trusted party using the `javakey` utility. The machine where the agent is running must have the signer's certificate in its keystore.

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–based architecture. This includes:

- Developing an SNMP agent with the SNMP protocol adaptor
- Representing your SNMP MIB (Management Information Base) as MBeans generated by the `mibgen` compiler
- If needed, developing an SNMP manager using the SNMP Manager API

For more information regarding the SNMP toolkit, refer to the *Java Dynamic Management Kit 4.2 Tools Reference* guide and the *Java Dynamic Management Kit 4.2 Tutorial*.

Developing An SNMP Agent

An SNMP agent is an application which responds to SNMP requests formulated as get and set 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 those MBeans specifically implement the MIB.

The SNMP protocol adaptor implements the both the SNMP v1 and v2 protocols. It responds to request in SNMP and translates that request into management operations on the specific MIB MBeans. The SNMP adaptor may also send traps, the equivalent of a JMX notification, in response to SNMP events or errors.

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

One advantage of the dual JMX–SNMP agent is that MIBs may be loaded dynamically in response to network conditions, or even in response to SNMP requests. Other Java Dynamic Management applications may 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 adaptor also sends inform requests from an SNMP agent to an SNMP manager. The SNMP manager will then send 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 SNMP v1 or SNMP v2 syntax.

A MIB is like a management interface: it defines what is exposed, but it doesn't 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 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 the v1 and v2 protocols, including:

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

The Java Dynamic Management Kit also provides an example program, showing how an agent may act as an SNMP proxy to access MIBs in subagents. This allows SNMP managers to access hierarchies of agents through a single proxy agent. In this way, some MIBs may be implemented by native devices and others may 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 which can issue requests to SNMP agents.

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

The SNMP manager API supports requests in either the SNMP v1 or v2 protocol, including inform requests for communicating between SNMP managers. The manager API is used to access any compliant SNMP agent including those developed using the Java Dynamic Management Kit.

CHAPTER 3

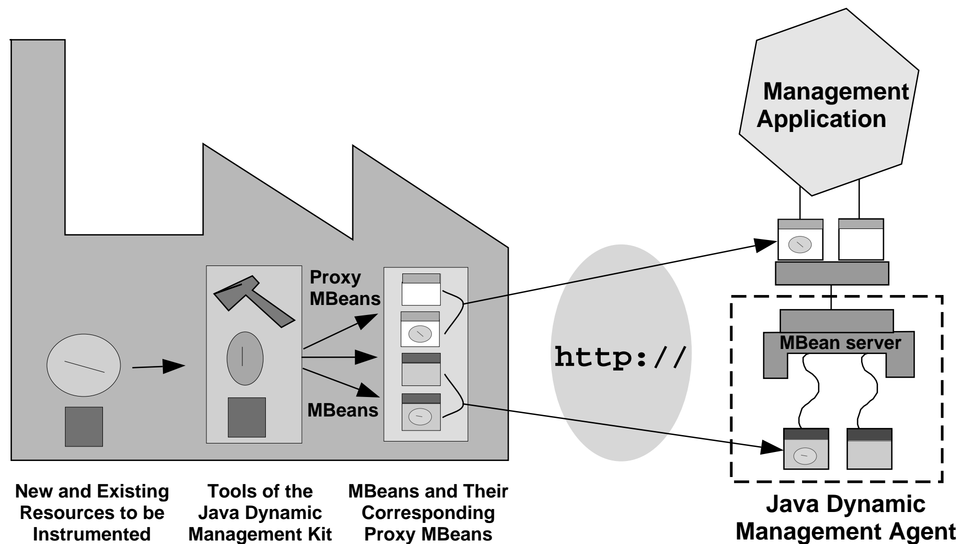
The Development Process

This chapter outlines the main steps involved in developing management solutions using the Java Dynamic Management Kit.

The steps in the development process are:

- "Instrumenting Resources" on page 46.
- "Designing an Agent Application" on page 46.
- "Generating Proxy MBeans" on page 47, an optional step.
- "Designing a Management Application" on page 48.

The following diagram summarizes these steps, from crafting MBeans in your factory to deploying them through the web.



This chapter is mostly concerned with design issues in the development process. For a description of how to write the code of management applications, see the programming examples in the *Java Dynamic Management Kit 4.2 Tutorial*.

Instrumenting Resources

MBeans conform to the JMX specification, which standardizes the representation of the MBean's management interface. Therefore, the first step of 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?" These are all questions whose answers determine the granularity of your MBean's management interface.

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

Sometimes, resources are already manageable through some other system. In this case, you only need to translate their existing management interfaces into an MBean. Because the JMX architecture is so rich, you can usually improve upon the existing management interface in the translation. Some operations may 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 insuring that they interact as expected.

Designing an Agent Application

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

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

The most simple 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! And 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 will hold.

At the other extreme, your entire management solution could be located in the agent. All the policies and all of resources you need could be managed locally. This application will be overburdened with its management tasks and does not take advantage of distributed management logic. You need to strike a balance 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 only be expected 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 may establish relations between MBeans in all of the subagents. Desktop machines and workstations can easily handle agents with thousands of MBeans.

The hierarchical model is very appropriate, since management logic and power are concentrated towards the top of the hierarchy. The information from many small devices gets concentrated on a few large servers where the management consoles are located. In between are medium sized agents which 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 which depends upon the design of your management application. As discussed earlier in this guide, a proxy 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 the 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 which returns a value; 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 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 which 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 Dynamic Management Kit. Generic proxies are part of the Java Dynamic Management runtime libraries and just need to be instantiated.

Options of the `proxygen` tool allow you to modify the characteristics of the proxies you generate from an MBean. For example, the read-only option will generate 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 could 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

In this section, we will focus 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, we give a list of features that managers may need to implement. 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

Don’t 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 which 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 may 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 which are available through some API. The manager also monitors all data for abnormal values and triggers a notification when they occur. These could arguably be the tasks of a smart agent, but let us suppose it is an intermediate manger for very simple agents in the routers.

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

In both cases, the applications may 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 which 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 may 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 we 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.