Java™ Management Extensions (JMX™) Remote API 1.0 Specification

Final Release

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

October 2003

JMX™ Remote API Specification ("Specification")

Version: 1.0
Status: FCS
Release: October 17, 2003

NOTICE; LIMITED LICENSE GRANTS

Sun Microsystems, Inc. ("Sun") hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under the Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation, which shall be understood to include developing applications intended to run on an implementation of the Specification provided that such applications do not themselves implement any portion(s) of the Specification.

Sun also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or patent rights it may have in the Specification to create and/or distribute an Independent Implementation of the Specification that: (i) fully implements the Spec(s) including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (iii) passes the TCK (including satisfying the requirements of the applicable TCK Users Guide) for such Specification. The foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose.

You need not include limitations (i)-(iii) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to implementations of the Specification (and products derived from them) that satisfy limitations (i)-(iii) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Spec in question.

For the purposes of this Agreement: "*Independent Implementation*" shall mean an implementation of the Specification that neither derives from any of Sun's source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun's source code or binary code materials; and "*Licensor Name Space*" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof.

This Agreement will terminate immediately without notice from Sun if you fail to comply with any material provision of or act outside the scope of the licenses granted above.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, JMX, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Specification and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#135575/Form ID#011801)

Contents

CHAPTER 1

Introduction

This document defines the Java™ Management Extensions (JMX™) Remote API, in conjunction with the API documentation generated by the Javadoc™ tool. This specification was produced through the Java Community Process℠ (JCP), as Java Specification Request (JSR) number 160.

1.1 Purpose of This Standard

Java Specification Request (JSR) 3 [JSR3] defines the JMX specification. What is standardized by JSR 3 is the way in which resources are instrumented within a management agent based on Java technology, and a certain number of agent-local services based on that instrumentation. Although JSR 3 defines terminology for remote access to instrumentation, it does not standardize any particular remote access API or protocol. Many solutions exist for exporting JMX API instrumentation either through existing management protocols such as the simple network management protocol (SNMP) or through proprietary protocols. This JSR (JSR 160) standardizes one such solution.

The principal goals of this standard are *interoperability*, *transparency*, *security*, and *flexibility*.

The standard is *interoperable* because it completely defines the standard protocols that are used between client and server, so that two different implementations of the standard can communicate.

The standard is *transparent* because it exposes an API to the remote client that is as close as possible to the API defined by the JMX specification for access to instrumentation within the agent.

The standard is *secure* because it builds on the Java technology standards for security, namely the Java Secure Socket Extension (JSSE), the Simple Authentication and Security Layer (SASL), and the Java Authentication and Authorization Service

(JAAS). These standards enable connnections between clients and servers to be private and authenticated and allow servers to control what operations different clients can perform.

The standard is *flexible* because, in addition to the required protocol, it provides ways for new transport protocols to be added and new implementations of the existing protocols to be substituted.

1.2 Required Version of the JMX Specification

This standard is separate from the JMX specification. It builds on JMX technology to provide a remoting capability. The required version of the JMX specification is 1.2, or any later version. Version 1.2 defines (among other things) these features, used by this standard:

- A parent interface of the MBeanServer interface, called `MBeanServerConnection` (See Section 2.3 “MBean Server Operations Through a Connection” on page 17)
- The ability to construct a proxy for a managed bean (MBean) accessed through an `MBeanServerConnection`, to simplify access to MBeans and make it type-safe
- Permissions that can be used to control access to individual MBeans, and even to particular attributes and operations within those MBeans

1.3 History

It was originally planned that this JSR would define a new version of the JMX specification that added a remoting capability. Because it was desired to integrate the JMX technology into the Java 2 Platform Enterprise Edition (the J2EE™ platform) 1.4, and given the schedule constraints of the J2EE platform 1.4, this plan was modified by splitting the work into two parts:

- The necessary changes to the existing JMX specification were handled as a Maintenance Release of JSR 3. This produced JMX 1.2, which was released in December 2002.
- JSR 160 was redefined as a separate specification that builds on the JMX specification to provide a remoting capability.

In addition to this change, two items from the original JSR proposal have been omitted from this initial version of the standard:

- A discovery and lookup service is no longer proposed. Rather than reinventing the wheel, this standard instead details how to advertise and find JMX API agents using existing discovery and lookup infrastructures. See Chapter 6 “Bindings to Lookup Services”.
- A Hypertext Transfer Protocol (HTTP) connector is no longer proposed as standard. HTTP incurs a significant protocol overhead. The main argument for retaining it would be that the secure HTTP/S variant is more likely to be accepted by firewalls. But the JMXMP connector proposed by this standard is based on the same security technology, Transport Layer Security (TLS).

CHAPTER 2

Connectors

The JMX specification defines the notion of *connectors*. A connector is attached to a JMX API MBean server and makes it accessible to remote Java technology-based clients. The client end of a connector exports essentially the same interface as the MBean server.

A connector consists of a *connector client* and a *connector server*.

A connector server is attached to an MBean server and listens for connection requests from clients.

A connector client takes care of finding the server and establishing a connection with it. A connector client will usually be in a different Java Virtual Machine[1] (JVM™) from the connector server, and will often be running on a different machine.

A given connector server can establish many concurrent connections with different clients.

A given connector client is connected to exactly one connector server. A client application can contain many connector clients connected to different connector servers. There can be more than one connection between a given client application and a given server.

Many different implementations of connectors are possible. In particular, there are many possibilities for the protocol used to communicate over a connection between client and server. This standard defines a standard protocol based on Remote Method Invocation (RMI) that must be supported by every conformant implementation. It also defines an optional protocol based directly on TCP sockets, called the JMX Messaging Protocol (JMXMP). An implementation of this standard can omit the JMXMP connector.

1. The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java platform.

2.1 Sessions and Connections

A distinction is made between a *session* and a *connection*. A connector client sees a *session*. During the lifetime of that session, there can be many successive *connections* to the connector server. In the extreme case, there might be one connection per client request, for example if the connector uses a stateless transport such as the user datagram protocol (UDP) or the Java Message Service (JMS).

A session has state on the client, notably its listeners (see Section 2.4 “Adding Remote Listeners” on page 18). A session does not necessarily have state on the server, and for the two connectors defined by this specification, it does not.

A connection does not necessarily have state on the client or server, although for the two connectors defined here it does.

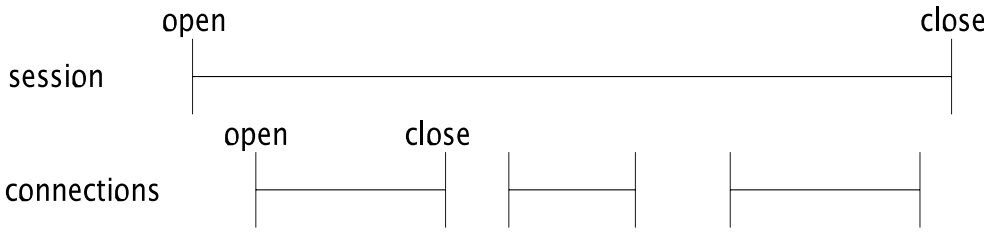

FIGURE 2-1 A Session Can Contain Many Successive Connections

In FIGURE 2-1 three connections are opened and closed over the lifetime of a single session.

2.2 Connection Establishment

In FIGURE 2-2, a connector client connects to a connector server with the address "`service:jmx:jmxmp://host1:9876`". A successful connection request returns the client end of the connection to the connector client.

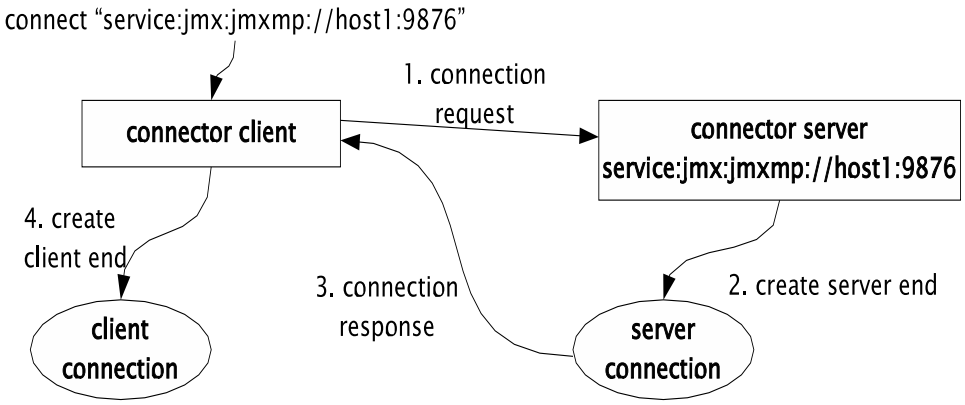

FIGURE 2-2 Connector Client and Server Communicate to Make a Connection

2.3 MBean Server Operations Through a Connection

From the client end of a connection, user code can obtain an object that implements the `MBeanServerConnection` interface. This interface is very similar to the `MBeanServer` interface that user code would use to interact with the MBean server if it were running in the same Java Virtual Machine.

`MBeanServerConnection` is the parent interface of `MBeanServer`. It contains all the same methods except for a small number of methods only appropriate for local access to the MBean server. All of the methods in `MBeanServerConnection` declare `IOException` in their "throws" clause in addition to the exceptions declared in `MBeanServer`.

Because `MBeanServer` extends `MBeanServerConnection`, client code can be written that works identically whether it is operating on a local MBean server or on a remote MBean server through a connector.

In FIGURE 2-3, the operation `getMBeanInfo("a:b=c")` on the `MBeanServerConnection` in a remote client is translated into a `getMBeanInfo` request that is sent to the server end of the connection via the connector protocol. The server reacts to this request by performing the corresponding operation on the local MBean server, and sends the results back to the client. If the operation succeeds, the client’s `getMBeanInfo` call returns normally. If the operation

produces an exception, the connector arranges for the client's `getMBeanInfo` call to receive the same exception. If there is a problem in the communication of the request, the client's `getMBeanInfo` call will get an `IOException`.

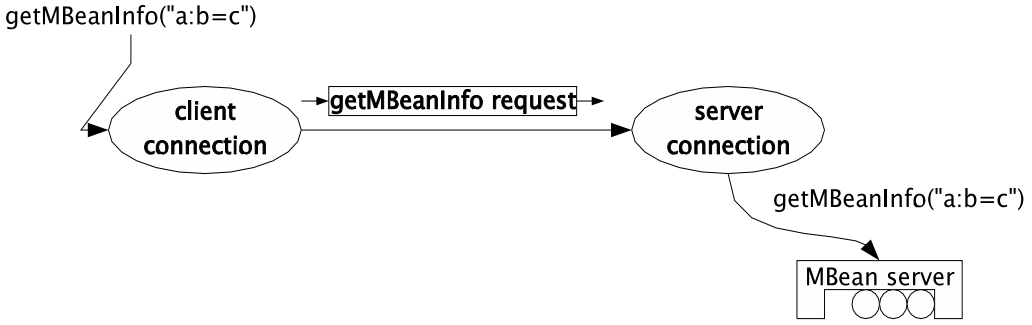

FIGURE 2-3 An Operation on the Client Results in the Same Operation on the MBean Server

2.4 Adding Remote Listeners

One of the operations in the `MBeanServerConnection` interface is the `addNotificationListener` operation. As in the local case, this operation registers a *listener* for the notifications emitted by a named MBean. A connector will arrange for the notifications to be sent from the server end of a connection to the client end, and from there to the listener.

The details of how notifications are sent depend on the connector protocol. The two connectors defined in this specification use a *stateless notification buffer*, as described in Section 2.4.3 "Notification Buffer" on page 20.

2.4.1 Filters and Handbacks

The `addNotificationListener` method in the `MBeanServerConnection` interface has four parameters: the *object name*, the *listener*, the *filter*, and the *handback*. The object name specifies which MBean to add the listener to. The listener is the object whose `handleNotification` method will be called when a notification is emitted by the MBean. As described in Section 2.4 "Adding Remote Listeners" on page 18, this listener object is local to the client.

The optional *filter* selects which notifications this listener is interested in. A given connector can execute the filter when the notification arrives at the client, or it can transmit the filter to the server to be executed there. Executing the filter on the server is much more efficient because it avoids sending a notification over the network only to have it discarded on arrival. Filters should be designed so that they work whether they are run on the client or on the server. In particular, a filter should be an instance of a serializable class known to the server. Section 2.11 “Class Loading” on page 28, describes class loading in more detail.

The connectors defined by this standard execute filters on the server.

To force filtering to be done on the client, the filtering logic can be moved to the listener.

The optional *handback* parameter to `addNotificationListener` is an arbitrary object that will be given to the listener when the notification arrives. This allows the same listener object to be registered with several MBeans. The handback can be used to determine the appropriate context when a notification arrives. The handback object remains on the client - it is not transmitted to the server and does not have to be serializable.

The `MBeanServerConnection` interface also has an `addNotificationListener` variant that specifies the listener as an `ObjectName`, the name of another MBean that is to receive notifications. With this variant, both the filter and the handback are sent to the remote server.

2.4.2 Removing Listeners

In general, a listener that has been added with the following method is uniquely identified for a given *name* by the triple (*listener,filter,handback*):

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

It can subsequently be removed either with the two-parameter `removeNotificationListener`, specifying just *listener*, or with the four-parameter `removeNotificationListener` that has the same parameters.

A problem arises with the four-parameter method in the remote case. The filter object that is deserialized in the `removeNotificationListener` method is not generally identical to the filter object that was deserialized for `addNotificationListener`. Since notification broadcaster MBeans usually check for equality in the (*listener,filter,handback*) triple using identity rather than the `equals` method, it would not in general be possible to remove just one (*listener,filter,handback*) triple remotely.

The standard connectors avoid this problem by using *listener identifiers*. When a connector client adds a (*listener,filter,handback*) triple to an MBean, the connector server returns a unique identifier for that triple on that MBean. When the connector client subsequently wants to remove the triple, it uses the identifier rather than passing the triple itself. To implement the two-parameter `removeNotificationListener` form, the connector client looks up all the triples that had the same listener and sends a `removeNotificationListener` request with the listener identifier of each one.

This technique has the side-effect that a remote client can remove a triple even from an MBean that implements `NotificationBroadcaster` but not `NotificationEmitter`. A local client of the `MBeanServer` interface cannot do this.

2.4.3 Notification Buffer

The two connectors defined by this specification handle notifications and listeners as follows. Every connector server has a *notification buffer*. Conceptually, this is a list of every notification ever emitted by any MBean in the MBean server that the connector server is attached to. In practice, the list is of finite size, so when necessary the oldest notifications are discarded.

Entries in the notification buffer consist of a `Notification` object and an `ObjectName`. The `ObjectName` is the name of the MBean that emitted the notification.

For every MBean that can send notifications (implements the `NotificationBroadcaster` interface), the connector server registers a listener that adds each notification to the notification buffer. The connector server tracks the creation of MBeans, and when a new `NotificationBroadcaster` MBean is created, the listener is added to it.

Entries in the notification buffer have sequence numbers. Sequence numbers are positive. A later notification always has a greater sequence number than an earlier one. Sequence numbers are not necessarily contiguous, but the notification buffer always knows what the next sequence number will be.

FIGURE 2-4 shows a connector server with its notification buffer. The notification buffer has saved four notifications, with sequence numbers 40 to 43. The next notification will have sequence number 44.

The client state of a session includes the sequence number of the next notification that the client has not yet seen. In FIGURE 2-4, the client of session 1 has not yet seen the notifications starting with number 41. The client of session 2 has seen all notifications, so the next notification it will see will have the next available sequence number, 44.

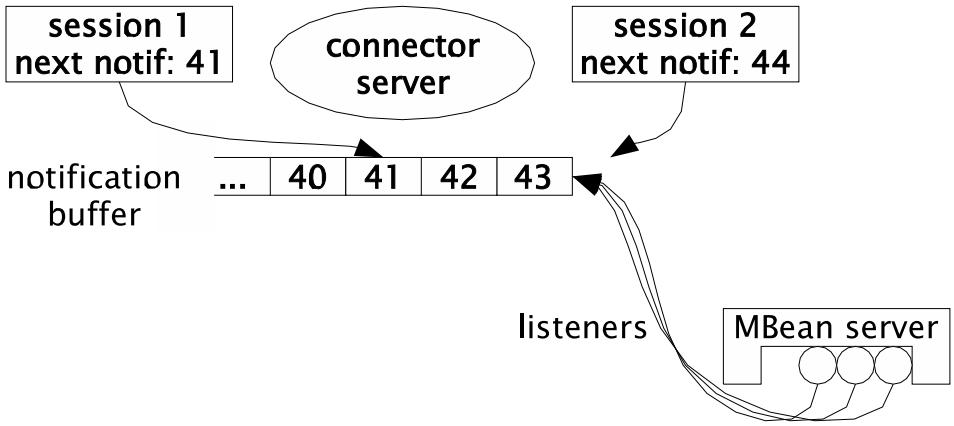

FIGURE 2-4 Notification Buffer Saves Notifications From All MBeans

When a new session is created, the client asks for the next sequence number that will be used. It is only interested in notifications with that number or greater, not the arbitrarily old notifications that are already present.

A notification buffer has no state related to the connector server. So an implementation can use the same notification buffer for more than one connector server.

2.4.4 Getting Notifications From the Notification Buffer

Conceptually, a connector client receives notifications by sending a *fetch-notifications* request to the connector server. The request looks like this:

> "Give me the notifications starting with sequence number *s* that match my filters."

Here, *s* is the next sequence number the client expects to see. In FIGURE 2-4, *s* is 41 for session 1 and 44 for session 2.

"My filters" means the `ObjectName` and `NotificationFilter` values for every `addNotificationListener` operation that has been done on the connector client. This filter list is either sent with every *fetch-notifications* request, or it is maintained as part of the state of a connection. The latter approach is followed in the two protocols defined by this specification, because the filter list is potentially very big.

The *fetch-notifications* request will wait until one of the following conditions is met:

- There is at least one notification in the buffer that matches the client's criteria, namely that has a sequence number at least s and matches the client's filters.
- A timeout specified by the client is reached.
- The connector server decides to terminate the operation, typically because of a timeout of its own.

The result of the *fetch-notifications* call includes the following information:

- Zero or more notifications that matched the client's criteria. The result does not have to include all available notifications. It may be limited to a maximum number, for example. But if there are notifications, they will be the earliest available ones.
- A sequence number n that is the number the client should use in its next *fetch-notifications* call. This is the sequence number of the first notification that matched the caller's criteria but was not included in the result, or it is the next available sequence number if all matched notifications were included.
- A sequence number f that is the smallest sequence number of a notification still in the buffer. If $f > s$, it is possible that notifications the client was interested in have been lost. It is not certain, however, because the notifications between s and f might not have matched the caller's filters.

This information is encapsulated in the `NotificationResult` class from the API.

As an example, suppose that in FIGURE 2-4 the notifications 41 and 43 match the filters for session 1. Its *fetch-notifications* call will have $s = 41$ and can return immediately with notifications 41 and 43, $n = 44$, and $f = 40$. No notifications have been lost ($f \leq s$) and the next *fetch-notifications* will have $s = 44$.

2.5 Concurrency

A JMX Remote API connector must support concurrent requests. If a thread calls a potentially slow operation like `invoke` on the client end of a connector, another thread should not be forced to wait for that operation to complete before performing an operation.

2.6 Normal Termination

Either end of a session can terminate the session at any time.

If the client terminates a session, the server will clean up any state relative to that client. If there are client operations in progress when the client terminates the session, then the threads that invoked them will receive an `IOException`.

If the server terminates a session, the client will get an `IOException` for any remote operations that were in progress and any remote operations subsequently attempted.

It is not specified what happens to MBean server operations that are running when the remote session that caused them is closed. Typically, they will run to completion, since in general there is no reliable way to stop them.

2.7 Abnormal Termination

The client end of a session can detect that the server end has terminated abnormally. This might happen for example because the JVM software that the server was running in exited, or because the machine it was running on crashed. The connector protocol (or its underlying transport) might also determine that the server is unreachable, because communication to it has not succeeded for a certain period of time. This can happen if there is a physical or configuration problem with the network.

In all of these cases, the client can terminate the session. The behavior seen by code using the client should be the same as if the server had terminated the session normally, except that the details of the exception seen by the client might differ.

Similarly, the server end of a session, or a connection within a session, can detect that the client end has terminated abnormally or become unreachable. It should behave as if the client had terminated the connection normally, except that the notification of connection termination indicates a failure.

2.7.1 Detecting Abnormal Termination

Transport protocols such as TCP usually have built-in detection of abnormal termination. When a Java Virtual Machine exits, any TCP connections it had are explicitly closed by the TCP protocol, meaning that the other end of the connection is informed promptly that the connection has been closed. But when a machine crashes or the network connection fails, this is detected less promptly. For example, TCP will only notice that a connection is broken if an attempt is made to write on it, and even then it will typically only signal the problem after a timeout on the order of minutes. Connectors should close connections that receive errors, but an additional mechanism is needed if connections are mostly idle or if the time to detect a failed connection is too long.

For the two connectors defined by this specification, an implementation is not required to detect failure promptly. However, the following approach is recommended:

1. A *fetch-notifications* call from the client should be terminated with zero notifications if none arrive within a certain period.
2. A connector server should close a connection that has not received any client requests (including *fetch-notifications*) for a certain time.
3. A client should specify a timeout in each *fetch-notifications* call. If the call does not return after the timeout (plus some margin for delays) then the client should close the connection.

This approach is based on the assumption that a client will always do a new *fetch-notifications* call shortly after the previous one returns. So case 2 never happens for a working connection.

If a session has no listeners, there is no need for it to do a *fetch-notifications* call. In this case, a server that follows the approach detailed here will close idle connections. The client will re-open the connection the next time it needs to do an operation on it.

2.8 Connector Server Addresses

A connector server usually has an address, which clients can use to establish connections to the connector server. Some connectors can provide alternative ways to establish connections, such as through connection *stubs* (see Section 2.9.2 “Connection Stubs” on page 26).

When a connector server has an address, this address is usually described by the class `JMXServiceURL`. The API documentation for that class and for the standard connectors explains the semantics of these addresses.

A user-defined connector can choose to use another address format, but it is recommended to use `JMXServiceURL` where possible.

An example of a connector server address is shown below:

```
service:jmx:jmxmp://host1:9876
```

All `JMXServiceURL` addresses begin with "`service:jmx:`". The following `jmxmp` indicates the connector to use, in this case the JMXMP Connector (see Chapter 4 “Generic Connector”). `host1` and `9876` are respectively the host and the port on which the connector server is listening.

2.9 Creating a Connector Client

A connector client is represented by an object that implements the `JMXConnector` interface. There are two ways in which a connector client can be created:

- Using an address, as covered in Section 2.9.1 “JMXConnectorFactory” on page 25
- Using a connection stub, as covered in Section 2.9.2 “Connection Stubs” on page 26

Which way an application uses depends mainly on the infrastructure that is used to find the connector server to which the client wants to connect.

2.9.1 JMXConnectorFactory

If the client has the address (`JMXServiceURL`) of the connector server to which it wants to connect, it can use the `JMXConnectorFactory` to make the connection. This is the usual technique when the client has found the server through a text-based discovery or directory service such as SLP.

For example, an application *app1* that includes an MBean server might export that server to remote managers as follows:

1. Create a connector server `cServer`
2. Get `cServer`’s address `addr`, either by using the `JMXServiceURL` that was supplied to its constructor to tell it what address to use, or by calling `cServer.getAddress()`
3. Put the address somewhere the management applications can find it, for example in a directory or in an SLP service agent

A manager can start managing *app1* as follows:

1. Retrieve *addr* from where it was stored in step 3 above
2. Call `JMXConnectorFactory.connect(addr)`

2.9.2 Connection Stubs

An alternative way for a client to connect to a server is to obtain a *connector stub*. A connector stub is a `JMXConnector` object generated by a connector server. It is serializable so that it can be transmitted to a remote client. A client that retrieves a connector stub can then call the stub's `connect` method to connect to the connector server that generated it.

For example, an application *app1* that includes an MBean server might export that server to remote managers as follows:

1. Create a connector server `cServer`
2. Obtain a connector stub `cStub` by calling `cServer.toJMXConnector`
3. Put the stub somewhere the management applications can find it, for example in a directory, in the Jini™ lookup service, or in an HTTP server

A manager can start managing *app1* as follows:

1. Retrieve `cStub` from where it was stored in step 3 above
2. Call `cStub.connect` to connect to the remote MBean server through `cServer`

In some circumstances, a connector server might not have all the information needed to generate a connector stub that any client can use. The details of connection might depend on the client's environment. In such cases, the connector stub would need to be generated by a third party, for example by administrative tools that know the relevant details of the client and server environments.

2.9.3 Finding a Server

Chapter 6 "Bindings to Lookup Services", defines how an agent based on JMX technology can register its connector servers with existing lookup and discovery infrastructures, so that a JMX Remote API client can create or obtain a `JMXConnector` object to connect to the advertised servers. In particular, that chapter provides the following information:

- Section 6.3 "Using the Service Location Protocol" on page 62, describes how a client can retrieve a JMX service URL from SLP, and use it to connect to the corresponding server
- Section 6.4 "Using the Jini Network Technology" on page 66, describes how a client can retrieve a connector stub from the Jini lookup service (LUS) and connect to the corresponding server

- Section 6.5 “Using the Java Naming and Directory Interface (LDAP Backend)” on page 72, describes how a client can retrieve a JMX service URL from the Lightweight Directory Access Protocol (LDAP) directory, and use it to connect to the corresponding server

2.10 Creating a Connector Server

A connector server is represented by an object of a subclass of `JMXConnectorServer`. The usual way to create a connector server is through the `JMXConnectorServerFactory`. Using a `JMXServiceURL` provided as a parameter, the factory determines what class to instantiate, in a way similar to the `JMXConnectorFactory` described in Section 2.9.1 “JMXConnectorFactory” on page 25.

A connector server can also be created by instantiating a subclass of `JMXConnectorServer` explicitly.

To be useful, a connector server must be attached to an MBean server, and it must be *active*.

A connector server can be attached to an MBean server in one of two ways. Either the MBean server to which it is attached is specified when the connector server is constructed, or the connector server is registered as an MBean in the MBean server to which it is attached.

A connector server does not have to be registered in an MBean server. It is even possible, though unusual, for a connector server to be registered in an MBean server different from the one to which it is attached.

CODE EXAMPLE 2-1 shows how to create a connector server that listens on an unspecified port on the local host. It is attached to the MBean server `mbs` but not registered in it:

CODE EXAMPLE 2-1 Creating a Connector Server attached to an MBean Server

```
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
JMXServiceURL addr = new JMXServiceURL("jmxmp", null, 0);
JMXConnectorServer cs =
  JMXConnectorServerFactory.newJMXConnectorServer(addr, null, mbs);
cs.start();
```

The address that the connector server is actually listening on, including the port number that was allocated, can be obtained by calling `cs.getAddress()`.

CODE EXAMPLE 2-2 shows how to do the same thing but with a connector server that is registered as an MBean in the MBean server to which it is attached:

CODE EXAMPLE 2-2 Creating a Connector Server Registered in an MBean Server

```
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
JMXServiceURL addr = new JMXServiceURL("jmxmp", null, 0);
JMXConnectorServer cs =
  JMXConnectorServerFactory.newJMXConnectorServer(addr, null, null);
ObjectName csName = new ObjectName(":type=cserver,name=mycserver");
mbs.registerMBean(cs, csName);
cs.start();
```

2.10.1 Publishing a Server

Chapter 6 “Bindings to Lookup Services” defines how an agent can publish its connector servers with existing lookup and discovery infrastructures, so that a JMX Remote API client that does not know about such a server can find it and connect to it. In particular, that section provides the following information:

- Section 6.3 “Using the Service Location Protocol” on page 62, describes how an agent registers the JMX service URL of a connector server with SLP, so that a JMX Remote API client can retrieve it and use it to connect to the server
- Section 6.4 “Using the Jini Network Technology” on page 66, describes how an agent registers the connector stub of a connector server with the Jini lookup service, so that a JMX Remote API client can retrieve this stub and connect to the server
- Section 6.5 “Using the Java Naming and Directory Interface (LDAP Backend)” on page 72, describes how an agent registers the JMX Service URL of a connector server in an LDAP directory, so that a JMX Remote API client can retrieve this URL and use it to connect to the server.

2.11 Class Loading

Every non-primitive Java object has a class, and every class has a class loader. A subtle pitfall of class loading is that the class `a.b.C` created by the class loader *cl1* is not the same as the class `a.b.C` created by the class loader *cl2*. Here, "created" refers

to the class loader that actually creates the class with its `defineClass` method. If *cl1* and *cl2* both find `a.b.C` by delegating to another class loader *cl3*, it is the same class.

A value of type "`a.b.C` created by *cl1*" cannot be assigned to a variable or parameter of type "`a.b.C` created by *cl2*". An attempt to do so will result in an exception such as `ClassCastException`.

When one end of a connection receives a serialized object from the other end, it is important that the object be deserialized with the right class loader. This section explains the rules for determining the class loader to use in every case.

These rules for class loading are needed when the types of attributes, or of operation parameters and return values, are application-specific Java types. To avoid having to deal with these rules, it is a good idea to use only standard types defined by the Java platform or by the JMX and JMX Remote APIs. The types defined for *Open MBeans* in the JMX API allow arbitrarily complex data structures to be described without requiring application-specific types. An important side-effect is that interoperation with non-Java clients is greatly simplified.

These rules are also needed when application-specific notification filters are applied. (See Section 2.4.1 “Filters and Handbacks” on page 18.) To avoid having to manage class-loading rules, consider using only the three standard notification filter types from the JMX API, `NotificationFilterSupport` , `MBeanServerNotificationFilter`, and `AttributeChangeNotificationFilter`. An alternative is to filter in the client’s listener, though this can increase network traffic with notifications that are discarded as soon as they are received.

2.11.1 Class Loading on the Client End

A connector client can specify a *default class loader* when making a connection to a server. This class loader is used when deserializing objects received from the server, whether they are returned values from `MBeanServerConnection` methods, exceptions thrown by those methods, or notifications emitted by MBeans to which the client is listening.

The default class loader is the value of the attribute `jmx.remote.default.class.loader` from the `JMXConnector` environment. The `JMXConnector` first looks for this attribute in the environment `Map` that was supplied when the `JMXConnector` was connected. If there was none, or the attribute is not found, it then looks in the environment `Map` that was supplied at creation time. If there was none, or the attribute is not found, then the default class loader is the context class loader

(`Thread.currentThread().getContextClassLoader()`) that was in place when the `JMXConnector` was connected. It is not specified what happens if the default class loader determined by these rules is null.

If the value of the `jmx.remote.default.class.loader` attribute is not a class loader, then the attempt to connect the `JMXConnector` gets an `IllegalArgumentException`.

Note – ***serialization:*** When a `JMXConnector` is serialized, the environment `Map` that was supplied when the `JMXConnector` was created is lost: the `Map` is not serialized because it is expected to contain objects, like class loaders, which are not serializable. As a consequence, when a specific default class loader is required for a `JMXConnector`, it is recommended always to specify it in the `Map` supplied when connecting.

2.11.2 Class Loading on the Server End

The class loader to be used when deserializing parameters received from the client depends on the operation. Sometimes the appropriate class loader is the one that belongs to the target MBean, because that MBean might have parameter types that are not defined by the JMX API or the JMX Remote API. Sometimes the appropriate class loader is one configured during the creation of the connector server, because the connector server is intended to be used with a particular management application. Such an application might define its own subclasses of MBean parameter types, or it might define its own `NotificationFilter` classes for listeners. An MBean being managed cannot be expected to anticipate every notification filter that a management application might want to use, so it does not make sense to use only the MBean's class loader to deserialize notification filters with listeners being added to the MBean.

Like a connector client, a connector server has a *default class loader* that is determined when the connector server is started. The default class loader is determined as follows:

- If the connector server's environment map contains the attribute `jmx.remote.default.class.loader`, the value of that attribute is the default class loader
- If the environment map contains the attribute `jmx.remote.default.class.loader.name`, the value of that attribute is the `ObjectName` of an MBean that is the default class loader. This allows a connector server to be created with a class loader that is a management applet (m-let) in the same MBean server
- If neither of the above attributes is defined, the default class loader is the thread's context class loader at the time when the `JMXConnectorServer` was started

If both `jmx.remote.default.class.loader` and `jmx.remote.default.class.loader.name` are defined, or if the value of `jmx.remote.default.class.loader` is not a `ClassLoader`, or if the value of `jmx.remote.default.class.loader.name` is not an `ObjectName` that names a `ClassLoader`, the attempt to start the connector server gets an `IllegalArgumentException`.

For certain operations that interact with a single "target" MBean, *M*, objects are deserialized using *M*'s *extended class loader*. This is a class loader that loads each class *X*, as follows:

1. The class loader that loaded or is loading *M* is asked to load *X*
2. If that fails with a `ClassNotFoundException`, the default class loader is asked to load *X*
3. If step 1 fails with an exception other than `ClassNotFoundException`, or if step 2 fails with any exception, that exception is the result of loading *X*

The rules for deserialization of `MBeanServerConnection` operations are as follows:

- The parameters to `setAttribute`, and `setAttributes` are deserialized using the target MBean's extended class loader
- The `Object` array in `invoke` is deserialized using the target MBean's extended class loader
- The `Object` array in the `createMBean` forms that have one is deserialized using the target MBean's extended class loader. Here, "the class loader that loaded or is loading *M*" is the class loader described in the API documentation for the particular `createMBean` form. In the case of the form that uses the Class Loader Repository, it is a class loader that always delegates to that repository
- The `QueryExp` in the `queryNames` and `queryMBeans` operations is deserialized using the default class loader
- The `NotificationFilter` and the `Object` handback in the `addNotificationListener` and `removeNotificationListener` operations (all forms) are deserialized using the target (notification broadcaster) MBean's extended class loader

Remaining parameters are of type `String` (which is a final class known to the bootstrap class loader), `String[]`, or `ObjectName`.

If a user-defined subclass of `ObjectName` is sent from client to server, it is not specified how it is deserialized, so this is not guaranteed to work in general.

2.12 Connector Server Security

Connector servers typically have some way of authenticating remote clients. For the RMI connector, this is done by supplying an object that implements the `JMXAuthenticator` interface when the connector server is created. For the JMXMP connector, this is done using SASL.

In both cases, the result of authentication is a JAAS `Subject` representing the authenticated identity. Requests received from the client are executed using this identity. With JAAS, you can define what permissions the identity has. In particular, you can control access to MBean server operations using the `MBeanPermission` class. For this to work, though, you must have a `SecurityManager`.

If a connector server does not support authentication or is not set up with authentication, then client requests are executed using the same identity that created the connector server.

As an alternative to JAAS, you can control access to MBean server operations by using an `MBeanServerForwarder`. This is an object that implements the `MBeanServer` interface by forwarding its methods to another `MBeanServer` object, possibly performing additional work before or after forwarding. In particular, the object can do arbitrary access checks. You can insert an `MBeanServerForwarder` between a connector server and its MBean server using the method `setMBeanServerForwarder`.

2.12.1 Subject Delegation

Any given connection to a connector server has at most one authenticated `Subject`. This means that if a client performs operations as or on behalf of several different identities, it must establish a separate connection for each one.

However, the two standard connectors also support the notion of *subject delegation*. A single connection is established between client and server using an authenticated identity, as usual. With each request, the client specifies a per-request `Subject`. The request is executed using this per-request identity, provided that the authenticated per-connection identity has permission to do so. That permission is specified with the permission `SubjectDelegationPermission`.

For each delegated `Subject`, the client obtains an `MBeanServerConnection` from the `JMXConnector` for the authenticated `Subject`. Requests using this `MBeanServerConnection` are sent with the delegated `Subject`. `MBeanServerConnection` objects for any number of delegated identities can be obtained from the same `JMXConnector` and used simultaneously.

CHAPTER 3

RMI Connector

The RMI connector is the only connector that must be present in all implementations of this specification. It uses the RMI infrastructure to communicate between client and server.

3.1 RMI Transports

RMI defines two standard transports, the Java Remote Method Protocol (JRMP) and the Internet Inter-ORB Protocol (IIOP).

JRMP is the default transport. This is the transport you get if you use only the `java.rmi.*` classes from the Java 2 Platform Standard Edition (the J2SE™ platform).

IIOP is a protocol defined by CORBA. Using RMI over IIOP allows for interoperability with other programming languages. It is covered by the `javax.rmi.*` and `org.omg.*` classes from the J2SE plaform.

RMI over these two transports is referred to as RMI/JRMP and RMI/IIOP.

The RMI connector supports both transports. Refer to the API documentation (in particular the description of the `javax.management.remote.rmi` package) for details.

3.2 Mechanics of the RMI Connector

For every RMI connector server, there is a remotely-exported object that implements the remote interface `RMIServer`. A client that wants to communicate with the connector server needs to obtain a remote reference, or *stub*, that is connected to this remote object (how the stub can be obtained is described in Section 3.3 “How to Connect to an RMI Connector Server” on page 37). RMI arranges that any method called on the stub is forwarded to the remote object. So, a client that has a stub for the `RMIServer` object can call a method on it, resulting in the same method being called in the server’s object.

FIGURE 3-1 shows two clients that both have stubs for the same server object. The server object is labeled *impl* because it is the object that implements the functionality of the `RMIServer` interface.

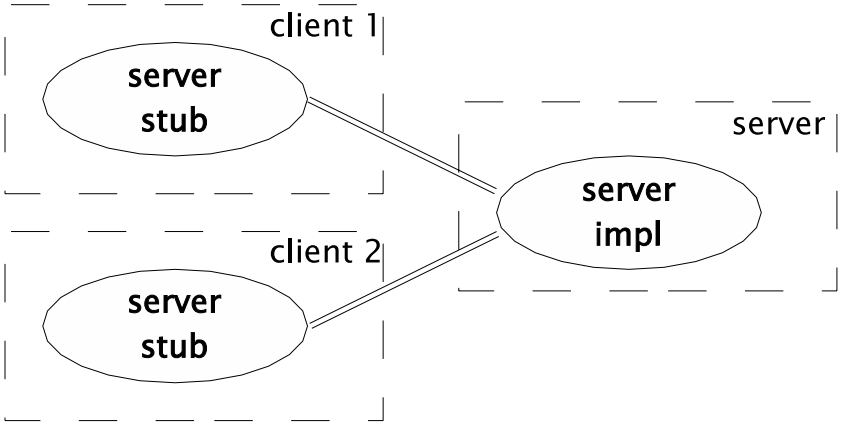

FIGURE 3-1 Several Clients can Have Stubs Connected to the Same Server Object

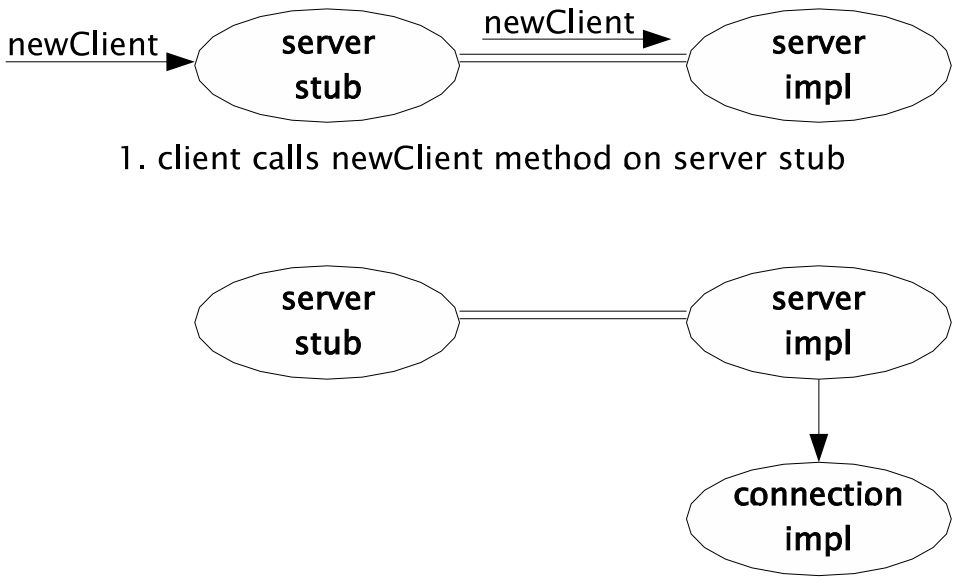

FIGURE 3-2 A New Client Connection Is a New Remote Object on the Server

In addition to the remote object representing the connector server, there is one remote object for every client connection made through the connector to the MBean server. When a client wants to invoke methods on the remote MBean server, it invokes the `newClient` method in its server stub. This causes the `newClient` method in the remote server object to be invoked. This method creates a new remote object that implements the remote interface `RMIConnection`, as shown in FIGURE 3-2. This interface contains all the remotely-accessible methods of the MBean

server. The value returned from the client's `newClient` method is a stub that is connected to this new object. When the client calls an MBean server method such as `getAttribute`, this produces a call to the corresponding method in the `RMIConnection` stub, and hence a remote call to the corresponding implementation object in the server.

3.2.1 Wrapping the RMI Objects

User code does not usually interact directly with the `RMIServer` and `RMIConnection` objects.

On the server side, the `RMIServer` object is created and exported by an `RMIConnectorServer`. `RMIConnectorServer` is a subclass of `JMXConnectorServer`, and as such is a connector server for the purposes of this standard. `RMIConnection` objects are created internally by the `RMIServer` implementation, but user code in the server never sees them.

On the client side, an `RMIServer` stub can be obtained explicitly, as described in Section 3.3 "How to Connect to an RMI Connector Server" on page 37. More usually, it is obtained as part of the process of looking up a URL for the RMI connector, but is wrapped in an `RMIConnector` object. User code usually only deals with this `RMIConnector` object. `RMIConnector` implements the `JMXConnector` interface and it is through this interface that it is usually accessed.

In normal use, user code never invokes any methods from `RMIServer`, and never sees any objects of type `RMIConnection`. These objects are hidden by the `RMIConnector` class.

3.2.2 RMIConnection

The `RMIConnection` interface is similar to the `MBeanServerConnection` interface defined by the JMX specification, but has some important differences:

- Parameters that are subject to the class loading rules detailed in Section 2.11 "Class Loading" on page 28 are wrapped inside a `MarshalledObject` so that they can be unwrapped by the server after it has determined the appropriate class loader to use
- The `addNotificationListeners` and `removeNotificationListener` methods use listener IDs instead of listeners, as detailed in Section 2.4 "Adding Remote Listeners" on page 18
- There are additional methods to get the connection ID and to close the connection
- There is an additional method to obtain outstanding notifications

The RMIConnection object represents a *connection*, not a *session*, in the terminology of Section 2.1 “Sessions and Connections” on page 16. Either end of the connection can close it at any time without affecting the session. The server closes the connection by unexporting the RMIConnection object. Ongoing RMI calls on the object run to completion and return normally, but new calls will fail. When the client sees such a failure, it will obtain a new RMIConnection object as described in Section 3.2 “Mechanics of the RMI Connector” on page 34.

3.2.3 Notifications

The RMI connector uses the techniques described in Section 2.4 “Adding Remote Listeners” on page 18. The connector server has a stateless notification buffer (Section 2.4.3 on page 20). If the connector client has listeners, it uses the fetchNotifications call on the RMIConnection object to receive notifications for them.

The list of (ObjectName,NotificationFilter) pairs corresponding to the client’s listeners is not passed in every call to fetchNotifications. Rather, it is established with a single addNotificationListeners call when the RMIConnection object is created. Changes to the notification list while the connection is open are made with further calls to addNotificationListeners and to removeNotificationListener.

3.3 How to Connect to an RMI Connector Server

Broadly, there are three ways to connect to an RMI connector server:

1. Supply a JMXServiceURL to the JMXConnectorFactory that specifies the rmi or iiop protocol. This is the most usual way to connect. The JMXServiceURL either contains the stub in an encoded form, or indicates a directory entry in which an RMIServer stub can be found. This is further described in the API specification of the javax.management.remote.rmi package. The details of looking up this directory entry and creating a JMXConnector from it are hidden from the caller

2. Obtain a JMXConnector stub from somewhere, for example a directory such as LDAP, the Jini Lookup Service, or as the returned value of an RMI method call. This stub is an object generated by RMIConnectorServer.toJMXConnector. It

is an object of type `JMXConnector`. It is not an RMI stub and should not be confused with the RMI stubs of type `RMIServer` or `RMIConnection`. However, it references an `RMIServer` stub which it uses when its connect method is called

3. Obtain an `RMIServer` stub from somewhere and use it as a parameter to the constructor of `RMIConnector`

3.4 Basic Security With the RMI Connector

The RMI connector provides a simple mechanism for securing and authenticating the connection between a client and a server. This mechanism is not intended to address every possible security configuration, but provides a basic level of security for environments using the RMI connector. More advanced security requirements are better addressed by the JMXMP connector (see Section 4.3.3 “Security Features in the JMXMP Connector” on page 50).

To make an RMI connector server secure, the environment supplied at its creation must contain the property `jmx.remote.authenticator`, whose associated value is an object that implements the interface `JMXAuthenticator`. This object is responsible for examining the authentication information supplied by the client and either deriving a JAAS Subject representing the client, or rejecting the connection request with a `SecurityException`.

A client connecting to a server that has an `JMXAuthenticator` must supply the authentication information that the `JMXAuthenticator` will examine. The environment supplied to the connect operation must include the property `jmx.remote.credentials`, whose associated value is the authentication information. This object must be serializable.

This specification does not include any predefined authentication system. The simplest example of such a system is a secret string shared between client and server. The client supplies this string as its `jmx.remote.credentials`, and the server’s `JMXAuthenticator` checks that it has the correct value.

As a slightly more complicated example, the authentication information could be a `String[2]` that includes a username and a password. The `JMXAuthenticator` verifies these, for example by consulting a password file or by logging in through some system-dependent mechanism, and if successful derives a `Subject` based on the given username.

3.4.1 How Security Affects the RMI Connector Protocol

The authentication information supplied by the client is passed as an argument to the `newClient` call (see FIGURE 3-2). The connector server gives it to the `JMXAuthenticator`. If the `JMXAuthenticator` throws an exception, that exception is propagated to the client. If the `JMXAuthenticator` succeeds, it returns a `Subject`, and that `Subject` is passed as a parameter to the constructor of the new `RMIConnection` object. All of the MBean server methods in `RMIConnection` perform privileged work as this particular `Subject`, so that they have the permissions appropriate to the authenticated client.

3.4.2 Achieving Real Security

The solution outlined above is enough to provide a basic level of security. A number of problems have to be addressed to achieve a real level of security, however:

1. If the authentication information includes a password, and if the network is not secure, then attackers might be able to see the password sent from client to server
2. Attackers might be able to substitute their own server for the server that the client thinks it is talking to, and retrieve the password that the client sends to authenticate itself
3. Attackers might be able to see the RMI object ID of a legitimately-created `RMIConnection` object as it is accessed remotely. They could then use RMI to call that object, executing MBean server methods using the `Subject` that was authenticated when the object was created
4. Attackers might be able to guess this RMI object ID, for instance if object IDs are allocated as consecutive small integers

The first three problems can be solved by using an RMI socket factory so that the connection between client and server uses the Secure Socket Layer (SSL). This is covered in more detail elsewhere (see for example "Using RMI with SSL" [RMI/SSL]).

A special case of problem 2 is that attackers might be able to modify the contents of a directory or lookup service that is used during connection establishment. This might be either the directory that is used to find the `RMIServer` stub, or the directory that is used to find the URL. If an RMI Registry is used for the `RMIServer` stub, it should be secured with SSL.

The fourth problem can be solved by setting the standard RMI system property `java.rmi.server.randomIDs` to "`true`". This causes the 64-bit object ID of every export RMI object to be generated using a cryptographically strong random number generator. (See the documentation for the class `java.rmi.server.ObjID`.)

3.5 Protocol Versioning

The remote `RMIServer` interface includes a method `getVersion` that returns a string including a protocol version number. This standard specifies version 1.0 of the RMI connector protocol, which is currently the only version. Any given future version of this standard might or might not include an updated version of the protocol.

Each protocol version will have a version number which is the same as the version of this standard that first defines it. For example, if version 1.1 of this standard does not change the protocol but version 1.2 does, then the next RMI connector protocol version number will be 1.2.

All future versions of the RMI connector will include a remote `RMIServer` object that has at least the same methods as the current version, 1.0, and in particular the `getVersion` method. A future version might add further methods too.

If a future version adds methods to the `RMIServer` interface, it must ensure that the methods that a 1.0 client calls work as expected.

If the client side of the RMI connector defined in a future version uses methods added to the server in that version, it must check, using `getVersion`, that the server it is communicating with supports that version. Otherwise, it must limit itself to the methods that the server does support, perhaps losing some functionality as a consequence.

CHAPTER 4

Generic Connector

The JMX Remote API includes a *generic connector* as an optional part of the API. This connector is designed to be configurable by plugging in modules to define the following:

- The *transport protocol* used to send requests from the client to the server and to send responses and notifications from the server to the clients
- The *object wrapping* for objects sent from the client to the server whose class loader can depend on the target MBean

The JMXMP Connector is a configuration of the generic connector where the transport protocol is based on TCP and the object wrapping is native Java serialization (as defined by `java.io.ObjectOutputStream` etc.). Security is based on JSSE [JSSE], JAAS [JAAS], and SASL [JSR28][RFC2222].

The generic connector and its JMXMP configuration are optional, which means that an implementation can choose not to include them. An implementation that does include them must conform to their specification here and in the API documentation.

4.1 Pluggable Transport Protocol

Each configuration of the generic connector includes a *transport protocol*, which is an implementation of the interface `MessageConnection`. Each end of a connection has an instance of this interface. The interface defines three main methods:

- The `writeMessage` method writes a Java object to the other end of the connection. The Java object is of the type `Message` defined by the connector. It can reference other Java objects of arbitrary Java types. For the JMXMP Connector, the possible types of messages are contained in the package `javax.management.remote.message`.

- The `readMessage` method reads a Java object from the other end of the connection. The Java object is of type `Message` and again can refer to objects of arbitrary other types.
- The `close` method closes the connection

The connection is a full-duplex connection between the client and the server. A stream of requests is sent from client to server, and a stream of responses and notifications is sent from server to client. See FIGURE 4-1.

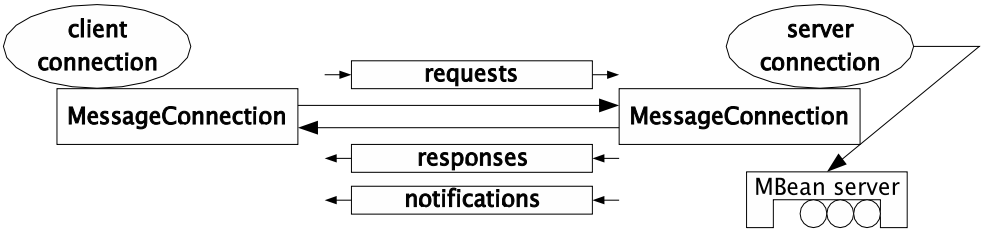

FIGURE 4-1 `MessageConnection` Defines a Full-Duplex Transport Between Client and Server

When client code issues an `MBeanServerConnection` request such as `getMBeanInfo`, the request is wrapped inside an `MBeanServerRequestMessage` object and written to the server using `MessageConnection.writeMessage`. The client code then waits for the corresponding response. Meanwhile, another thread in the client can write another request. When a response arrives, its message ID is used to match it to the request it belongs to, and the thread that issued that request is woken up with the response.

4.2 Pluggable Object Wrapping

The arguments to an MBean method called through `MBeanServer.invoke`, and the attribute values supplied to `setAttribute` or `setAttributes`, can be of Java classes that are known to the target MBean but not to the connector server. If these objects were treated like any other, the connector server would get a `ClassNotFoundException` when it tried to deserialize a request containing them.

To avoid this problem, deserialization at the server end of a connection proceeds in two stages. First, the objects that are necessarily of classes known to the connector server are deserialized. This is enough to determine what kind of request has been received, which MBean it is destined for (if any), and therefore what class loader is appropriate for use. Then the remaining objects (arguments to invoke or attribute values for `setAttribute(s)`) can be deserialized using this class loader.

The `ObjectWrapping` interface allows object wrapping to be customized. By default, it constructs a byte array containing the output of `ObjectOutputStream.writeObject` on the object or objects to be wrapped. But this would be inappropriate if, for example, the `MessageConnection` is using the Extensible Markup Language (XML). So, in such a case an `ObjectWrapping` object could be plugged into the connector that wraps the objects in XML. This XML can then be included in the larger XML text constructed by the `MessageConnection`.

4.3 Generic Connector Protocol

The generic connector protocol defines a set of protocol messages that are exchanged between the client and the server ends of the connection, and the sequence these message exchanges must follow. Implementations of this specification must exchange these messages in the defined sequence so that they can interoperate with other implementations. FIGURE 4-2 depicts the UML diagram of all the messages defined by the generic connector protocol.

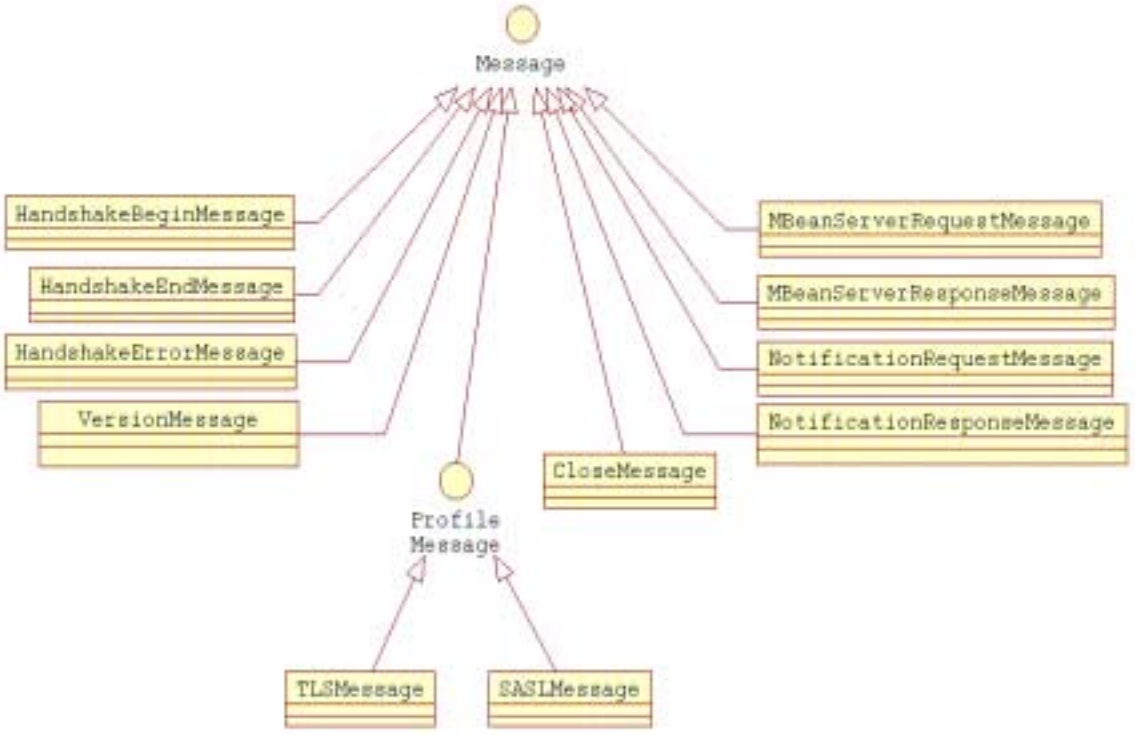

FIGURE 4-2 Generic Connector Protocol Messages

The generic connector protocol messages can be divided into four categories:

- Handshake messages:

```
HandshakeBeginMessage
HandshakeEndMessage
HandshakeErrorMessage
VersionMessage
```

- Profile messages:

```
TLSMessage (JMXMP Connector only)
SASLMessage (JMXMP Connector only)
```

- MBean server operation messages:

```
MBeanServerRequestMessage
MBeanServerResponseMessage
NotificationRequestMessage
NotificationResponseMessage
```

- Connection messages

```
CloseMessage
```

4.3.1 Handshake and Profile Message Exchanges

The handshake message exchanges are started by the server end of the connection as soon as the `connect` method on the `JMXConnector` class is called by the client and the connection between the client and the server is established.

The server end of the connection sends a `HandshakeBeginMessage` to the client with the profiles supported by the server. These profiles are retrieved from the environment map through the `jmx.remote.profiles property`. The client then starts the profile message exchanges for the profiles chosen from the server's supported profiles.

The JMXMP profile is used to negotiate the version of JMXMP to use. This profile is always implicitly enabled, but is only negotiated if the client and server differ in their default versions. See Section 4.3.5 "Protocol Versioning" on page 52.

For the other profiles, the client will first check that all the profiles requested in its environment map are supported by the server. If not, it will send a `HandshakeErrorMessage` to the server and close the connection. (This is the behavior of the standard JMX Remote API. Other APIs for JMXMP can provide ways to pick which of the proposed profiles to use.)

Then, for each profile asked for in the client's environment map, the client will negotiate that profile. The order in which profiles are negotiated is the order they appear in the client's environment map. This order can be important. For example, if the client negotiates the SASL/PLAIN profile before the TLS profile, it will send a password in clear text over the connection. If it negotiates TLS first, the connection will become encrypted before the password is sent.

It is not specified how the server accepts or denies the sequence of profiles run by the client. However, it is recommended that if the profiles in the server's environment map imply a certain level of security, the server should reject a connection whose negotiated profiles do not ensure that level of security. For example, if the server is configured with only the TLS profile, then it should reject connections that do not negotiate TLS. If the server is configured with the TLS profile and with the SASL/DIGEST-MD5 profile specifying the same level of security as regards authentication and encryption, then it should reject connections that negotiate neither profile.

The profile exchanges are performed one at a time and always started by the client. Once the profile exchanges are completed the client sends a `HandshakeEndMessage` to the server. No further profile exchanges are then possible. The server replies either with the same `HandshakeEndMessage` if it accepts the profiles that have been negotiated, or with a `HandshakeErrorMessage` if it does not. In the latter case, the connection is closed.

After the handshake phase has been completed the client can get a reference to the remote MBean server, send MBean server requests, and register listeners for receiving notifications. The server will send responses to the client MBean server requests and will forward notifications to the interested clients. FIGURE 4-3 depicts the initial handshake and profile message exchanges.

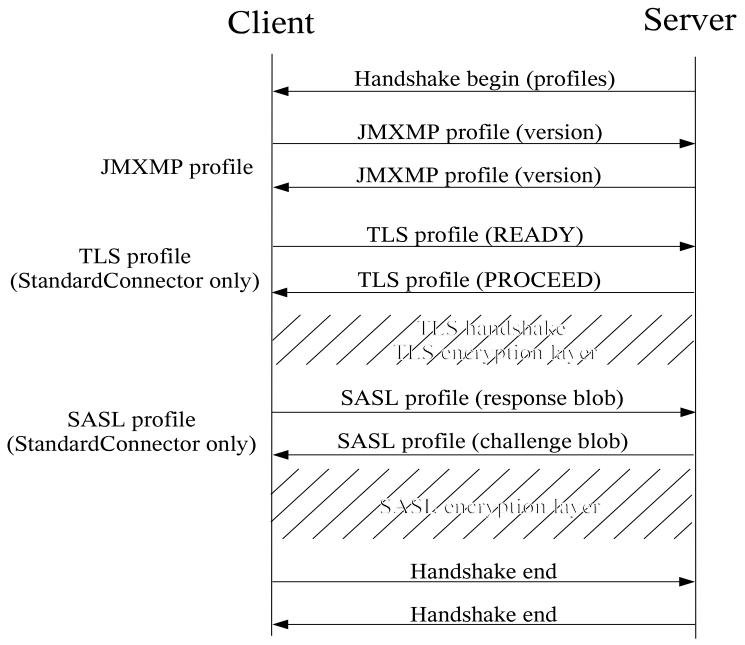

FIGURE 4-3 Handshake and Profile Message Exchanges

Notice that only the handshake begin and handshake end messages are mandatory. The profile message exchanges depend on the configuration of the server and the client by means of the `jmx.remote.profiles` property in the environment map passed in at the creation of the `JMXConnector` and `JMXConnectorServer`.

At any time during the handshake phase, if an error is encountered by either peer (client or server), it must send an indication (`HandshakeErrorMessage`) as to why the operation failed. The peer that encountered the problem will send the error message to the other peer and immediately close the connection. The peer that receives the message on the other end of the connection will also close the connection immediately on reception of a handshake error message. FIGURE 4-4 depicts how an error is indicated by either a client or a server to the other peer during the initial handshake message exchanges.

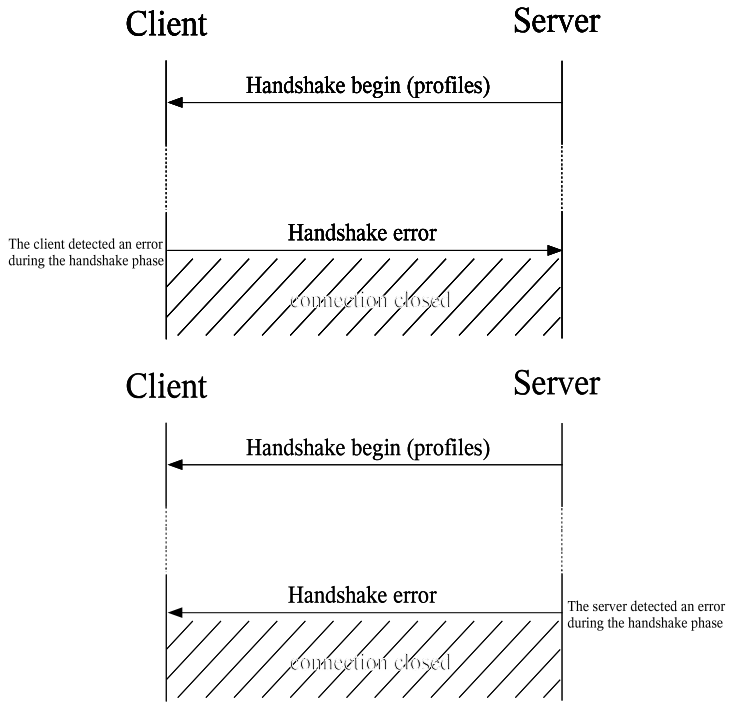

FIGURE 4-4 Handshake Error Message Exchanges

4.3.2 MBean Server Operation and Connection Message Exchanges

Once the initial handshake phase has been terminated, and all profiles negotiated, the client can retrieve a reference to the remote MBean server by calling the `getMBeanServerConnection` method on the `JMXConnector` instance. Through the `MBeanServerConnection` interface the client can perform operations on the registered MBeans, including registration for receiving notifications. These MBean server operations will be mapped by the protocol to `MBeanServerRequestMessage` messages. For each such message the server will receive it, decode it, perform the operation on the MBean server, and return the result of the operation in an `MBeanServerResponseMessage` message.

If several client threads are performing MBean server operations at the same time, there can be several `MBeanServerRequestMessages` that have been sent without yet having received the corresponding `MBeanServerResponseMessages`. There is no requirement that a client receive a response for each request before sending the next request.

Each `MBeanServerRequestMessage` contains an identifier that the matching `MBeanServerResponseMessage` must also contain. At any time, the client has a set of identifiers {*id1*, *id2*, ...} for requests it has sent that have not yet received a response. Each new request must have an identifier that is not in the set, and that is added to the set when the request is sent. Each response must have an identifier that is in the set, and that is removed from the set when the response is received. It is a protocol error for these conditions to be violated. The peer that detects the error must close the connection, optionally after sending a `CloseMessage` to the other peer.

Notifications are handled using the techniques described in Section 2.4 “Adding Remote Listeners” on page 18. The connector server has a stateless notification buffer (Section 2.4.3 on page 20). If the connector client has listeners, it uses the `NotificationRequestMessage` to receive notifications for them. Each such message solicits a `NotificationReplyMessage`.

The list of (`ObjectName`,`NotificationFilter`) pairs corresponding to the client’s listeners is not passed in every `NotificationRequestMessage`. Rather, it is established with a single `addNotificationListeners` in an `MBeanServerRequestMessage` when the connection is established. Changes to the notification list while the connection is open are made with further `MBeanServerRequestMessage`s containing `addNotificationListeners` or `removeNotificationListener`.

FIGURE 4-5 depicts the MBean server operation message exchanges.

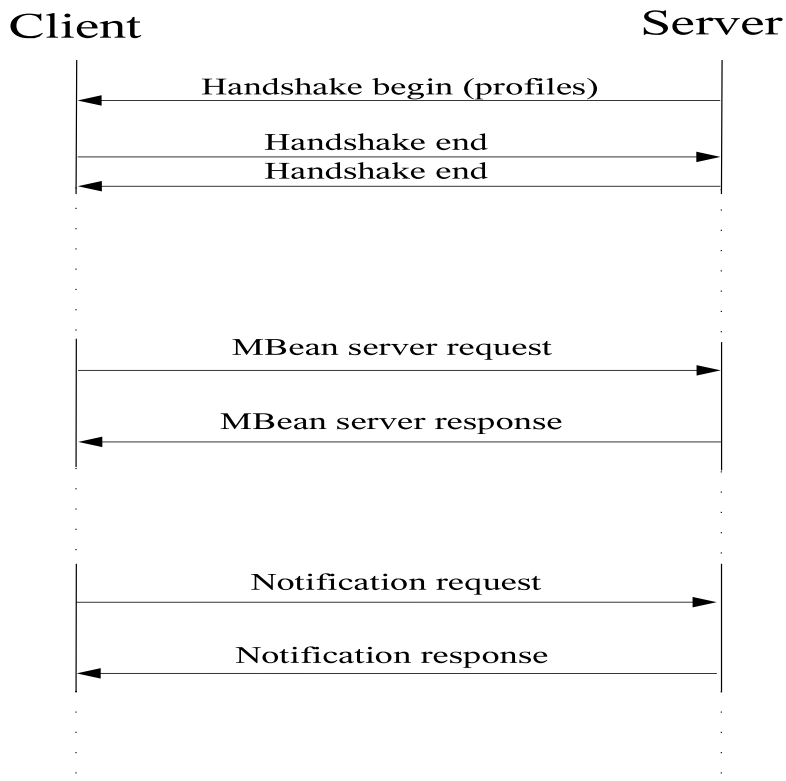

FIGURE 4-5 MBean Server Operations Message Exchanges

At any time after the handshake phase and during the MBean server operation message exchanges, either the client or the server can want to close the connection. On the one hand, the client can achieve that by calling the close method on the `JMXConnector` instance. On the other hand, the server can achieve that by calling the stop method on the `JMXConnectorServer` instance. Additionally, the client or server can close the connection at any time, for example as detailed in Section 2.7.1 “Detecting Abnormal Termination” on page 23. The peer initiating the connection close action will send a message of type `CloseMessage` to inform the other peer that the connection must be closed and that the necessary clean-up should be carried out.

When a client sends or receives a `CloseMessage` it must not send any further requests to the server over that connection. The server will continue to process existing requests and send the corresponding replies before closing the connection.

FIGURE 4-6 depicts the close-connection message exchanges.

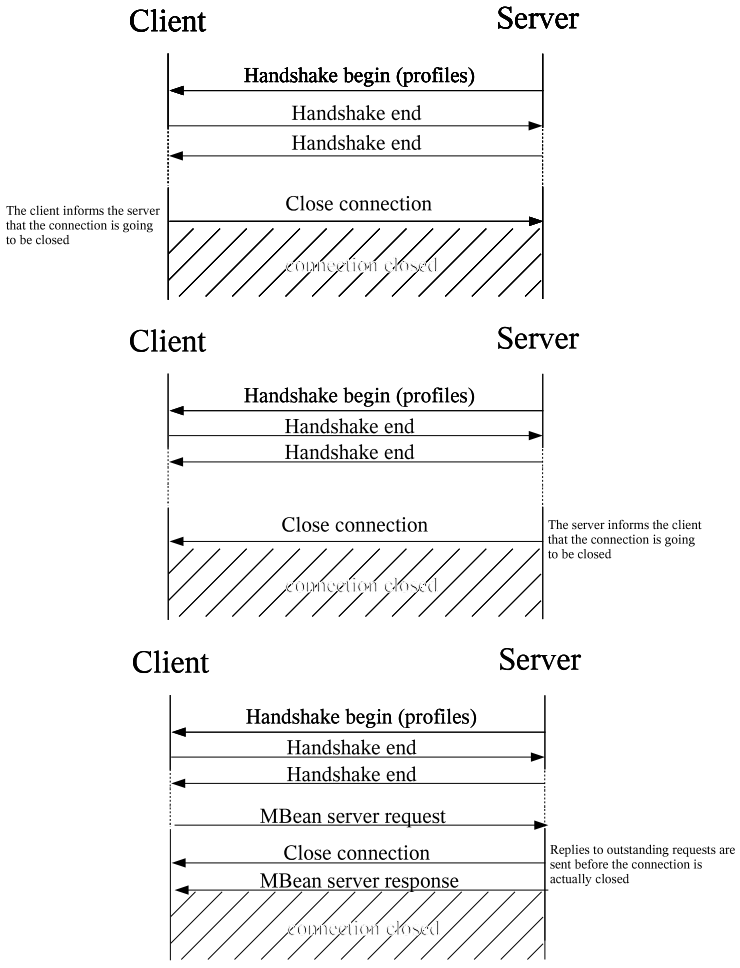

FIGURE 4-6 Close-Connection Message Exchanges

4.3.3 Security Features in the JMXMP Connector

The JMXMP Connector provides support for authentication and authorization through the TLS and SASL profiles. The JMX Remote API does not mandate the implementation and support of any specific SASL mechanism. It simply relies on third-party implementations that can be plugged in using the standard SASL interface [JSR28].

4.3.3.1 TLS Profile

The TLS profile allows the client and server ends of a connection to negotiate a TLS encryption layer. Certificate-based authentication and mutual client/server authentication are optional features configurable through properties in the environment map (see Section 4.3.6 “Properties Controlling Client and Server” on page 53).

4.3.3.2 SASL Profile

When using a SASL profile the way authentication is carried out is defined by the selected SASL mechanism and can vary from one mechanism to another.

However, at the end of the SASL handshake exchanges an authorization identity has been negotiated between the SASL client and the SASL server. Thus, the SASL profile has to make this identity available to allow the MBean server and the underlying MBeans to perform access control checks based on this identity.

The SASL profile implementation uses the JAAS framework to construct a `JMXPrincipal` based on this authorization identity, and stores this `JMXPrincipal` in a `Subject`. Then, when the `JMXMPConnectorServer` performs any of the subsequent MBean server operations, it must do so with the given subject for the required privileged action using an appropriate access control context.

An MBean interested in retrieving the authorization information can do so (if it has the appropriate permissions) by calling:

```
AccessControlContext acc = AccessController.getContext();
Subject subject = Subject.getSubject(acc);
Set principals = subject.getPrincipals();
```

4.3.4 Protocol Violations

If a peer receives a message from the other peer that does not respect the protocol described here, its behavior is unspecified. The recommended behavior is to send a `CloseMessage` indicating the detected violation and to close the connection immediately afterwards.

4.3.5 Protocol Versioning

This standard specifies version 1.0 of the JMXMP protocol, which is currently the only version. Any given future version of this standard might or might not include an updated version of the protocol.

Each protocol version will have a version number which is the same as the version of this standard that first defines it. For example, if version 1.1 of this standard does not change the protocol but version 1.2 does, then the next JMXMP protocol version number will be 1.2.

The first message sent over a newly-opened connection is a handshake begin message from the server to the client. This message includes the latest JMXMP version that the server understands. If the client also understands that version, then the subsequent communication will take place using that version. If the client only understands an earlier version, then it will send a `VersionMessage` requesting that the earlier version be used. If the server understands this earlier version, then it will reply with the same `VersionMessage`, and the subsequent communication will take place using that version. Otherwise, the server will send a `HandshakeErrorMessage` and the communication will be aborted.

In other words, suppose the server version is *S* and the client version is *C*. Then the version *V* to be used for communication is determined as follows:

- Server to client: "Version *S*"
- If client understands *S*, *V* = *S*
- Otherwise:
  - Client to server: "Version *C*"
  - If server understands *C*:

    - Server to client: "Version *C*"
    - *V* = *C*
  - Otherwise (server does not understand *C*):

    - Server to client: "Handshake error."
    - Connection aborted

A consequence of this negotiation is that every version of the protocol must understand every other version's `HandshakeBeginMessage` and `VersionMessage`. This will be true provided that Java serial compatibility is respected. See the section *Type Changes Affecting Serialization* in [Serial].

It is expected but not required that every implementation of any version of this standard understand all protocol versions from previous versions of the standard.

4.3.6 Properties Controlling Client and Server

When creating a `JMXConnector` or a `JMXConnectorServer`, an environment map can be supplied. One of the functions of this environment is to provide configuration parameters for the underlying profiles. The following subsections describe these parameters.

4.3.6.1 Global Properties of the Generic Connector

These properties control global aspects of the connection, that is they are valid regardless of the profiles that are selected.

- `jmx.remote.profiles`

  A string that is a space-separated list of profile names to be supported by the client and/or the server. Examples of profile names are: JMXMP, TLS, SASL/EXTERNAL, SASL/OTP. If this property is unspecified, no profiles will be used.

- `jmx.remote.context`

  An arbitrary `Object` to be conveyed by the handshake messages from one peer to the other. The `Object` should be serializable and of a class that is known to the other peer. If this property is unspecified, a null context will be conveyed.

  The JMXMP Connector currently makes no use of this object and does not expose it to user code on the client or server.

- `jmx.remote.authenticator`

  A `JMXAuthenticator` that is used at the end of the handshake phase to validate the new connection. The `authenticate` method of this object is called with a two-element `Object[]` as a parameter. The first element is the connection ID of the new connection. The second element is the authenticated `Subject`, if any. The method returns the authenticated `Subject` to use for the connection, or null if there is no authenticated ID. The returned `Subject` is usually the same as the `Subject` passed as a parameter, but it can have different `Principal`s. If the authenticator does not accept the connection id or `Subject`, it can throw a `SecurityException`.

4.3.6.2 TLS Properties

The following properties control the TLS profile:

- `jmx.remote.tls.socket.factory`

An object of type `javax.net.ssl.SSLSocketFactory` that is an already initialized TLS socket factory. The `SSLSocketFactory` can be created and initialized through the `SSLContext` factory. If the value of this property is not specified, the TLS socket factory defaults to `SSLSocketFactory.getDefault()`.

- `jmx.remote.tls.enabled.protocols`

  A string that is a space-separated list of TLS protocols to enable. If the value of this property is not specified, the TLS enabled protocols default to `SSLSocket.getEnabledProtocols()`.

- `jmx.remote.tls.enabled.cipher.suites`

  A string that is a space-separated list of TLS cipher suites to enable. If the value of this property is not specified the TLS enabled cipher suites default to `SSLSocket.getEnabledCipherSuites()`.

- `jmx.remote.tls.need.client.authentication`

  A string that is "`true`" or "`false`" according to whether the connector server requires client authentication. If `true`, a client that does not authenticate during the handshake sequence will be refused.

- `jmx.remote.tls.want.client.authentication`

  A string that is "`true`" or "`false`" according to whether the connector server requires client authentication if appropriate to the cipher suite negotiated. If true, then if a client negotiates a cipher suite that supports authentication but that client does not authenticate itself, the connection will be refused.

4.3.6.3 SASL Properties

The following properties control the SASL profile:

- `jmx.remote.sasl.authorization.id`

  A string that is the connector client's identity for authorization when it is different from the authentication identity. If this property is unspecified, the provider derives an authorization identity from the authentication identity.

- `jmx.remote.sasl.callback.handler`

  An object of type `javax.security.auth.callback.CallbackHandler` that is the callback handler to be invoked by the SASL mechanism to retrieve user information. If this property is unspecified, no callback handler will be used.

CHAPTER 5

Defining a New Transport

The standard protocols defined by this specification might not correspond to all possible environments. Examples of other protocols that might be of interest are:

- A protocol that runs over a serial line to manage a JMX API agent in a device that is not networked
- A protocol that uses HTTP/S because it is a familiar protocol that system administrators might be more willing to let through firewalls than RMI or JMXMP
- A protocol that formats messages in XML (perhaps in an XML-based RPC protocol such as SOAP) to build on an existing XML-based infrastructure. Such a transport could potentially be used by non-Java clients

There are two ways to implement a user-defined protocol. One is to define a transport for the generic connector using the `MessageConnection` and `MessageConnectionServer` classes as described in Chapter 4 “Generic Connector”. The other is to define a new provider for the `JMXConnectorFactory`.

Defining a transport for the generic connector has the advantage that many of the trickier implementation details, in particular concerning listeners, are already handled. The transport has to take care of establishing the connection and serializing and deserializing the various `Message` classes. Potentially, the transport can include other exchanges, for example to set up a secure connection, that are not the result of a `MessageConnection.writeMessage` and are never seen by a `MessageConnection.readMessage`. For example, this is the case for the TLS and SASL exchanges in the JMXMP Connector.

Defining a provider for the `JMXConnectorFactory` is explained in the API documentation for that class. A provider can be based on the generic connector, or it can implement a protocol completely from scratch.

CHAPTER 6

Bindings to Lookup Services

This standard specifies connectors that make it possible for a JMX Remote API client to access and manage MBeans exposed through a JMX API agent (an MBean server) running in a remote JVM. It also defines a `JMXServiceURL` class, which represents the address of a JMX Remote API connector server, and makes it possible for a client to obtain a JMX Remote API connector connected to that server. However, this standard does not provide any specific API that would make it possible for a client to find the address of a connector server attached to an agent it knows about, or to discover which agents are running, and what the addresses of the connector servers are that make it possible to connect to them. Rather than reinventing the wheel, this standard instead details how to advertise and find agents using existing discovery and lookup infrastructures.

This specification discusses three such infrastructures:

- The Service Location Protocol [SLP], as defined by [RFC 2608] and [RFC 2609]
- The Jini Network Technology [Jini]
- The Java Naming and Directory Interface™ ("J.N.D.I") API [JNDI] with an LDAP backend

The goal of this chapter is to specify how a JMX API agent can register its connector servers with these infrastructures, and how a JMX Remote API client can query these infrastructures in order to find and connect to the advertised servers.

This chapter imposes no requirements on implementations of the JMX Remote API. It details the conventions to be followed so that a server can be registered and found by clients, without having to share special knowledge between client and server.

6.1 Terminology

The term *JMX Remote API Agent* (or agent) is used throughout this section to identify a logical server application composed of:

- One MBean server
- One or more JMX Remote API connector servers allowing remote clients to access the MBeans contained in that MBean server

The term *JMX Remote API client* (or client) is used to identify a logical client application which opens a client connection with a JMX Remote API agent.

Note that a single JVM machine can contain many agents and/or clients.

6.2 General Principles

Although the APIs with which to register and query a server access point using a lookup service vary from one infrastructure to another, the general principles remain the same:

- The agent creates one or more JMX Remote API connector servers
- Then for each connector to expose, the `JMXServiceURL` (SLP, JNDI/LDAP) or the `JMXConnector` stub (Jini networking technology, JNDI/LDAP) is registered with the lookup service, possibly giving additional attributes which qualify the agent and/or connector
- The client queries the lookup service, and retrieves one or more `JMXServiceURL` addresses (or `JMXConnector` stubs) that match the query
- Then, it either uses the `JMXConnectorFactory` to obtain a `JMXConnector` connected with the server identified by a retrieved `JMXServiceURL` (SLP, JNDI/LDAP), or it directly connects to the server using the provided `JMXConnector` stub (Jini, JNDI/LDAP)

6.2.1 JMXServiceURL Versus JMXConnector Stubs

When using SLP, it is natural to register and retrieve a service URL from the lookup service. However, it is not as natural when using networking technologies like Jini. In the Jini networking technology, the Service object you register and get from the lookup service is usually a stub that directly implements the interface of the

underlying service, and not an object that gives you back some information on how to connect to the service. Therefore this standard specifies different ways of advertising a connector server, depending on the underlying lookup service used:

- **SLP**: register the URL string representation of the JMX Service URL (`JMXServiceURL.toString()`). This is natural as SLP is a URL-based protocol. See Section 6.3 “Using the Service Location Protocol” on page 62.
- **Jini networking technology**: register a `JMXConnector` stub. The `JMXConnector` interface is directly the interface of the JMX Connector Service. See Section 6.4 “Using the Jini Network Technology” on page 66
- **JNDI API/LDAP**: register the URL string representation of the JMX Service URL (`JMXServiceURL.toString()`). The JNDI API can be configured on the client side (via `StateFactories` and `ObjectFactories` - see [JNDI - Java Objects]) to create and return a new `JMXConnector` automatically from the `DirContext` containing the JMX Service URL, or simply return the `DirContext` from which that JMX Service URL can be extracted. See Section 6.5 “Using the Java Naming and Directory Interface (LDAP Backend)” on page 72.
  An alternative way to use JNDI/LDAP is to store a JMXConnector stub directly, as described for Jini. This specification does not define a standard way to do that.

6.2.2 Lookup Attributes

All three infrastructures considered in this specification have the notion of lookup attributes. These attributes are properties that qualify the registered services. They are passed to the infrastructure when the service is registered, and can be used as filters when performing a lookup.

A client can then query the lookup service to find all the connectors that match one or more attributes. A client that obtains several services as a result of a lookup query can also further inquire about the lookup attributes registered for those services to determine which of the returned matching services it wants to use.

For a client to be able to format a query to the lookup service independently of the JMX Remote API implementation used on the agent side, and to understand the meaning of the retrieved attributes, this standard specifies a common set of JMX Remote API lookup attributes whose semantics will be known by all agents and clients. In the remainder of this specification we will use the term *Lookup Attributes* for these.

When registering a connector server with a lookup service, an agent will:

1. Build the `JMXServiceURL` describing its connector server (SLP, JNDI/LDAP), or obtain a `JMXConnector` stub from that server (using Jini networking technology)
2. Register that URL (SLP, JNDI/LDAP), or `JMXConnector` stub (using Jini networking technology) with the lookup service

3. Provide any additional lookup attributes that might help a client to locate the server

TABLE 6-1 defines the set of common lookup attributes that can be provided at connector registration and that can be used to filter the lookup. Most of these attributes are optional: an agent can choose whether it wants to specify them when it registers a `JMXServiceURL` with the lookup service.

Note – The name format of the lookup attributes is different depending on the back-end lookup service (see Section 6.4 “Using the Jini Network Technology” on page 66)

TABLE 6-1 Lookup Attributes for Connectors

Name / ID	Type	Multi-valued	Optional	Description
`AgentName`	String	No	Mandatory	A simple name used to identify the agent in a common way. Can also be viewed as a logical name for the service implemented by the agent. Makes it possible to search for all connectors registered by a given agent. This specification does not define the format of an agent name. However, the characters colon (`:`) and slash (`/`) are reserved for future use.

TABLE 6-1 Lookup Attributes for Connectors

Name / ID	Type	Multi-valued	Optional	Description
`ProtocolType`	String	No	Optional	The protocol type of the registered connector, as returned by `JMXServiceURL.getProtocol()`. Makes it possible to retrieve only the connectors using a given protocol that the client supports.
`AgentHost`	String	Yes	Optional	The name(s) or IP address(es) of the host on which the agent is running. This attribute is multivalued in order to allow aliasing - namely, if one single host is known under several names. This attribute is multivalued only if the underlying lookup protocol supports multivalued attributes.
`Property`	String	Yes	Optional	A string containing a Java-like property, in the form "`<property-name>=<value>`" - for example, "`com.sun.jmx.remote.tcp.connect.timeout=200`". This attribute is multivalued so that it can be used to map several properties. It might be used by agents as a means to provide additional information to client applications. For instance, this attribute could be used to hold some of the attributes that were passed to a connector server within the environment map at construction time. However, an agent must not rely on the fact that a Client will read these attributes, and a client must not rely on the fact that an agent will provide them. All the information that any client will need to connect to a specific server must be contained in the server's JMX Service URL, or in its JMX API connector stub.

6.3 Using the Service Location Protocol

The Service Location Protocol [SLP] is is an IETF standards track protocol [RFC 2608], [RFC 2609] that provides a framework to allow networking applications to discover the existence, location, and configuration of networked services in enterprise networks. You may wish to read the [SLP White Paper] for a concise description of SLP, and its positioning with respect to other technologies, like DNSSRV and LDAP.

6.3.1 SLP Implementation

The Java SLP API is the object of [JSR 140]. At the time of writing, this JSR is not yet finalized. The code extracts in this section are based on Sun's proprietary Java implementation of SLP, which closely follows [RFC 2614]. Code based on other implementations of that RFC will work similarly.

6.3.2 SLP Service URL

The `JMXServiceURL` defined by this standard is directly compliant with [RFC 2609]. Therefore there is a direct mapping between JMX Service URLs and SLP Service URLs, since their String representation is identical.

6.3.3 SLP Lookup Attributes

SLP supports multivalued attribute registrations; these attributes are provided at registration time, when registering the Service URL of the connector server. The filtering method used for lookup is an LDAPv3 filter string. The attributes that must or may be provided by an agent when registering a connector server URL are those defined in Section 6.2.2 "Lookup Attributes" on page 59.

6.3.4 Code Templates

The following sections provide some code templates for SLP.

6.3.4.1 Discovering the SLP Service

With SLP, discovering the lookup service is transparent to the user; the running SLP daemon is responsible for finding the Service Agent or Directory Agent (depending on the configuration of the daemon).

In fact, one line is enough to locate the lookup service, as shown in CODE EXAMPLE 6-1:

CODE EXAMPLE 6-1 Discovering the SLP Service

```
import com.sun.slp.ServiceLocationManager;
import com.sun.slp.ServiceLocationException;
import com.sun.slp.Advertiser;
import com.sun.slp.Locator;
...
try {

    // Getting the Advertiser (for registration purposes)
   Advertiser slpAdvertiser = ServiceLocationManager.getAdvertiser(Locale.US);

    // Getting the Locator (for lookup purposes)
    Locator slpLocator = ServiceLocationManager.getLocator(Locale.US);

} catch(ServiceLocationException e) {...}
```

6.3.4.2 Registering a JMX Service URL With SLP

The class `Advertiser` is used to perform the SLP registrations, as shown in CODE EXAMPLE 6-2:

CODE EXAMPLE 6-2 Registering a Service URL With SLP

```
import com.sun.slp.ServiceURL;
import com.sun.slp.ServiceLocationAttribute;
...
try {

    // Create a new JMXMPConnectorServer, let the system allocate a
    // a port number.
    //
    JMXServiceURL jmxUrl = new JMXServiceURL("service:jmx:jmxmp://myhost:0");
    final JMXConnectorServer cserver = new JMXMPConnectorServer(jmxUrl,null);

    // Get the Connector Server address
    final JMXServiceURL srvAddr = cserver.getAddres();

    // Note: It is recommended that the JMX Agents make use of the leasing
    //       feature of SLP, and periodically renew their lease.
    final ServiceURL serviceURL =
           new ServiceURL(srvAddr.toString(), ServiceURL.LIFETIME_DEFAULT);

    final Vector attributes = new Vector();
    final Vector attrValues = new Vector();

    // Using the default SLP scope
    attrValues.add("DEFAULT");
    final ServiceLocationAttribute attr1 =
       new ServiceLocationAttribute("SCOPE", attrValues);
    attributes.add(attr1);

    // AgentName attribute
    attrValues.removeAllElements();
    attrValues.add(new String("my-jmx-agent"));
    final ServiceLocationAttribute attr2 =
        new ServiceLocationAttribute("AgentName", attrValues);
    attributes.add(attr2);
    ...
    // Registration
    slpAdvertiser.register(serviceURL, attributes);

} catch(ServiceLocationException e) {...}
```

6.3.4.3 Looking up a JMX Service URL With SLP

The class `Locator` is used to perform the SLP lookup, as shown in CODE EXAMPLE 6-3:

CODE EXAMPLE 6-3 Looking up a JMX Service URL With SLP

```
import com.sun.slp.ServiceType;
import com.sun.slp.ServiceLocationEnumeration;
...
try {
    // lookup in default SCOPE.
     final Vector scopes = new Vector();
    scopes.add("DEFAULT");

    // Set the LDAPv3 query string
   //     Here we look for a specific agent called "my-jmx-agent",
    //     but we could have asked for any agent by using a wildcard:
    //     final String query = "(&(AgentName=*))";
    //
    final String query = "(&(AgentName=my-jmx-agent))";

    // lookup
    final ServiceLocationEnumeration result =
       slpLocator.findServices(new ServiceType("service:jmx"), scopes, query);

   // Extract the list of returned ServiceURL
    while(result.hasMoreElements()) {
        final ServiceURL surl = (ServiceURL) result.next();

        // Get the attributes
        final ServiceLocationEnumeration slpAttributes =
             slpLocator.findAttributes(surl, scopes, new Vector());

        while(slpAttributes.hasMoreElements()) {
             final ServiceLocationAttribute slpAttribute =
                  (ServiceLocationAttribute) slpAttributes.nextElement();
             ...
        }

        // Open a connection
        final JMXServiceURL jmxUrl = new JMXServiceURL(surl.toString());
        final JMXConnector client = JMXConnectorFactory.connect(jmxUrl);
        ...
    }
} catch(ServiceLocationException e) {...}
```

6.4 Using the Jini Network Technology

The *Jini Network Technology* [Jini] is an open software architecture that enables developers to create network-centric services that are highly adaptive to change.

The Jini specification offers a standard lookup service. A running Jini lookup service can be discovered with a simple API call. A remote service (device, software, application, etc.) that wants to be registered in the Jini lookup service provides a serializable Java object. When looked up by a remote client, a copy of this Java object is returned. Usually, this object acts as a proxy to the remote service.

In addition, Jini networking technology offers various APIs and mechanisms to download code from a remote HTTP server (necessary to get the classes required for instantiating the proxy objects), and the Jini specification supports security for code download based on the RMI security manager.

6.4.1 Jini Networking Technology Implementation

The Jini networking technology is Java-based software the implementation of which is available for download under the Sun Community Source Licence v3.0 (with Jini Technology Specific Attachment v1.0). See http://wwws.sun.com/software/communitysource/jini/download.html

6.4.2 Service Registration

The Jini specification is based on service registration. A service is registered through a serializable Java object, which can be a stub, a proxy or a simple class providing information about the service. Usually, the registered service is a stub which provides a direct link to the underlying service. Thus, although it would be possible to use the `JMXServiceURL` as the service, this standard specifies the use of a JMX Remote API connector stub, implementing the `JMXConnector` interface, as the service. This is consistent with the Jini specification's philosophy, where objects retrieved from the Jini lookup service are usually proxies implementing the interface of the service looked up.

The Jini lookup service, which uses Java RMI marshalling and dynamic class loading semantics, will make use of RMI annotations to download automatically from the server side all the classes needed to deserialize the service object on the client side. This makes it possible for a server to register any private implementation class, and for a client to use that class (through its generic `JMXConnector` interface) without any a-priori knowledge of the server implementation. However, this requires a

certain amount of configuration from the server-side. This standard completely specifies the JMX Remote API connector stubs for the protocols it describes, so that an instance of such a class serialized from the JMX Remote API implementation on the server side can be deserialized in an instance of the same class using the implementation on the client side, without having to download any new classes. Thus, no special configuration is needed on the server side when using standard connectors. Providers and users of non-standard connectors should however perform the required configuration steps if they want to make their non-standard connectors available to generic JMX API clients.

6.4.3 Using JMX Remote API Connector Stubs

When registering a JMX Remote API connector stub, the server application will either call the `JMXConnectorFactory.newConnector` method to obtain an unconnected stub, or call the `toJMXConnector` method on the `JMXConnectorServer` it wants to register. The `toJMXConnector` method returns a serializable connector stub that can be directly registered as the service provided by that connector.

When the client looks up the registered connector from the Jini lookup service, the returned connector stub is not yet connected to its corresponding server. The client application needs to call the `JMXConnector.connect()` method on that object before using it.

Calling `JMXConnector.connect()` on the server side is shown in CODE EXAMPLE 6-4:

CODE EXAMPLE 6-4 Calling `JMXConnector.connect()` on the Server Side

```
// get the connector stub:
JMXConnector c = server.toJMXConnector(null);

// register c as the Jini Service.
...
```

Calling `JMXConnector.connect()` on the client side, as shown in CODE EXAMPLE 6-5:

CODE EXAMPLE 6-5 Calling `JMXConnector.connect()` on the Client Side

```
// Obtain the service from Jini
Object service = ...
JMXConnector c = (JMXConnector) service;

// Build the env Map, add security parameters,
Map env = new HashMap();
env.put(...)

// Connect with the server
c.connect(env);
```

6.4.4 Jini Lookup Service Attributes

Like SLP, the Jini lookup service supports the specification of additional lookup attributes, called *entries*. The Java class of these attributes must implement the `net.jini.core.entry.Entry` interface. The `Name` entry defined by the Jini specification is interpreted as meaning the `AgentName` as defined in Section 6.2.2 “Lookup Attributes” on page 59. As this specification was being completed, the other entries were being standardized through the Jini Community Decision Process (JDP). Refer to the JMX technology home page for current information:

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

6.4.5 Code Templates

The following sections provide some code templates for the Jini lookup service:

6.4.5.1 Discovering the Jini Lookup Service

The Jini lookup service is represented by the `net.jini.core.lookup.ServiceRegistrar` class. There are two ways to discover the Jini lookup service. The first and most simple way assumes that you know the address of the lookup service, as shown in CODE EXAMPLE 6-6:

CODE EXAMPLE 6-6 Discovering the Jini Lookup Service Using an Address

```
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
...
LookupLocator lookup = new LookupLocator("jini://my_host");
ServiceRegistrar registrar = lookup.getRegistrar();
```

The second solution uses a broadcast mechanism to retrieve the lookup services running on the accessible network, as shown in CODE EXAMPLE 6-7:

CODE EXAMPLE 6-7 Discovering the Jini Lookup Service Using a Broadcast Mechanism

```
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
...
LookupDiscovery lookupDiscovery = null;
try {
    lookupDiscovery = new LookupDiscovery(null);
} catch (IOException e) {...}

lookupDiscovery.addDiscoveryListener(new LookupDiscoveryListener());

private class LookupDiscoveryListener implements DiscoveryListener {

    public LookupDiscoveryListener() {
   }

    public void discovered(DiscoveryEvent evnt) {
        ServiceRegistrar[] regs = evnt.getRegistrars();
        for(int i = 0; i < regs.length; i++) {
```

CODE EXAMPLE 6-7 Discovering the Jini Lookup Service Using a Broadcast Mechanism

```
                String[] regGroups = regs[i].getGroups();
                // Must verify here that the ServiceRegistrar
                // contains the groups I want to register in...
            }

            // It is generally better here to launch another Thread to use
            // the discovered ServiceRegistrar; this avoids blocking the
            // discovery process.
        }

        public void discarded(DiscoveryEvent evnt) {}
}
```

6.4.5.2 Registering a JMX Remote API Connector Stub With the Jini Lookup Service

Registering a JMX Remote API Connector Stub with the Jini Lookup Service is shown in CODE EXAMPLE 6-8:

CODE EXAMPLE 6-8 Registering a JMX Remote API Connector Stub With the Jini Lookup Service

```
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.entry.Entry;
import net.jini.core.lease.Lease;
import java.rmi.RemoteException;
...
// Get the Jini ServiceRegistrar with one of the above methods
ServiceRegistrar registrar = ...;

// Get a connector stub for the server you want to export
//
JMXConnector proxy = jmxConnectorServer.toJMXConnector(null);

// Prepare Service's attributes entry
Entry[] serviceAttrs = new Entry[] {
    new net.jini.lookup.entry.Name("MyAgentName");
    // Add here the lookup attributes you want to specify.
};
```

CODE EXAMPLE 6-8 Registering a JMX Remote API Connector Stub With the Jini Lookup Service

```
// Create a ServiceItem from the service instance
ServiceItem srvcItem = new ServiceItem(null, proxy, serviceAttrs);

// Register the Service with the Lookup Service
try {
    ServiceRegistration srvcRegistration =
         registrar.register(srvcItem, Lease.ANY);
    System.out.println("Registered ServiceID: " +
        srvcRegistration.getServiceID().toString());
} catch(RemoteException e) {...}
```

6.4.5.3 Looking up a JMX Connector Stub From the Jini Lookup Service

Looking up a JMX Connector stub from the Jini lookup service is shown in CODE EXAMPLE 6-9:

CODE EXAMPLE 6-9 Looking up a JMX Connector Stub From the Jini Lookup Service

```
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.entry.Entry;
...
// Get the Jini ServiceRegistrar with one of the above methods
ServiceRegistrar registrar = ...;

// Prepare Service's attributes entry to be matched
Entry[] serviceAttrs = new Entry[] {
 // Retrieve all services for which a Name entry was registered,
    // whatever the name is (null = wildcard).
    new net.jini.lookup.entry.Name(null)

    // Add here any other matching attribute.

};

// Look for a specific JMXMP Connector (you may also pass
// JMXConnector.class if you wish to get all types of JMXConnector)
//
ServiceTemplate template = new ServiceTemplate(null,
```

CODE EXAMPLE 6-9 Looking up a JMX Connector Stub From the Jini Lookup Service

```
                new Class[] {JMXMPConnector.class}, serviceAttrs);

ServiceMatches matches = null;
try {
    matches = registrar.lookup(template, Integer.MAX_VALUE);
} catch (RemoteException e) {...}

// Retrieve the JMX Connector and initiate a connection
for(int i = 0; i < matches.totalMatches; i++) {
    if(matches.items[i].service != null) {

        // Get the JMXConnector
        JMXConnector c = (JMXConnector)(matches.items[i].service);

        // Prepare env (security parameters etc...)
        Map env = new HashMap();
        env.put(...);

        // Initiate the connection
        c.connect(env);

        // Get the remote MBeanServer handle
        MBeanServerConnection server = c.getMBeanServerConnection();
        ...
    }
}
```

6.5 Using the Java Naming and Directory Interface (LDAP Backend)

The *Java Naming and Directory Interface* [JNDI] is a standard extension to the Java platform, providing Java technology-enabled applications with a unified interface to multiple naming and directory services in the enterprise. In particular, it provides a means to access X.500 directory services through the Lightweight Directory Access Protocol (LDAP). This standard defines how an LDAP server can be used to store information about JMX API agents, and how JMX Remote API clients can look up this information to connect to the agents.

A good understanding of using JNDI API with an LDAP backend can be obtained by following the [LDAP Thread in the JNDI Tutorial].

6.5.1 LDAP Schema for Registration of JMX Connectors

Nodes in the LDAP directory tree are typed. A node can have several object classes. JMX Connectors should be registered in nodes of class *`jmxConnector`*. The *`jmxConnector`* class contains two attributes, which are the JMX Service URL of the corresponding connector (*`jmxServiceURL`*), and the name of the JMX API agent exporting this connector (*`jmxAgentName`*). The JMX Service URL can be absent if the agent is not accepting connections. The `jmxConnector` class also includes optional attributes, like *`jmxAgentHost`* and *`jmxProtocolType`*. The agent name makes it possible for a client application to get a connection to an agent it knows by name. Together with the *`jmxAgentHost`* and *`jmxProtocolType`* it also makes it possible to perform filtered queries, for instance, "*find all the JMXMP connectors of <this> JMX API agent*" or "*find all connectors of all agents running on <that> node*". CODE EXAMPLE 6-10 is the schema definition (as specified in [RFC 2252]) that should be used to register JMX Remote API connectors:

CODE EXAMPLE 6-10 LDAP Schema for Registration of JMX Remote API Connectors

```
-- jmxServiceURL attribute is an IA5 String
( 1.3.6.1.4.1.42.2.27.11.1.1 NAME 'jmxServiceURL'
  DESC 'String representation of a JMX Service URL'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )

-- jmxAgentName attribute is an IA5 String
( 1.3.6.1.4.1.42.2.27.11.1.2 NAME 'jmxAgentName'
  DESC 'Name of the JMX Agent'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )

-- jmxProtocolType attribute is an IA5 String
( 1.3.6.1.4.1.42.2.27.11.1.3 NAME 'jmxProtocolType'
  DESC 'Protocol used by the registered connector'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )

-- jmxAgentHost attribute is an IA5 String
( 1.3.6.1.4.1.42.2.27.11.1.4 NAME 'jmxAgentHost'
  DESC 'Names or IP Addresses of the host on which the
        agent is running. When multiple values are
        given, they should be aliases to the same host.'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )
```

CODE EXAMPLE 6-10 LDAP Schema for Registration of JMX Remote API Connectors

```
-- jmxProperty attribute is an IA5 String
( 1.3.6.1.4.1.42.2.27.11.1.5 NAME 'jmxProperty'
  DESC 'Java-like property characterizing the registered object.
        The form of each value should be: "<property-name>=<value>".
        For instance: "com.sun.jmx.remote.tcp.timeout=200"'
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )

-- jmxExpirationDate attribute is a Generalized Time
-- see [RFC 2252] - or X.208 for a description of
--                  Generalized Time
( 1.3.6.1.4.1.42.2.27.11.1.6 NAME 'jmxExpirationDate'
   DESC 'Date at which the JMX Service URL will
        be considered obsolete and can be removed
        from the directory tree'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.24 SINGLE-VALUE )

-- from RFC-2256 --
( 2.5.4.13 NAME 'description'
  EQUALITY caseIgnoreMatch
  SUBSTR caseIgnoreSubstringsMatch
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{1024} )

-- jmxConnector class - represents a JMX Connector.
--                      must contain the JMX Service URL
--                      and the JMX Agent Name
( 1.3.6.1.4.1.42.2.27.11.2.1 NAME 'jmxConnector'
  DESC  'A class representing a JMX Connector, and
         containing a JMX Service URL.
         The jmxServiceURL is not present if the server
         is not accepting connections'
  AUXILIARY
  MUST  ( jmxAgentName  )
  MAY   ( jmxServiceURL $ jmxAgentHost $ jmxProtocolType $
          jmxProperty $ jmxExpirationDate $ description ) )
```

The *jmxConnector* class is an AUXILIARY class, which means that its properties can be added to any node in the directory tree - namely, it does not impose any restriction on the structure of the directory tree.

To create a node in the directory tree, you also need a STRUCTURAL class. This specification does not impose any restriction on the structural classes that can contain JMX Remote API connectors. You can, for instance, reuse the *javaContainer* class from the *Java Schema* [JNDI - Java Schema] as defined in [RFC

2713], namely, create a node whose object classes would be *javaContainer* (STRUCTURAL) and *jmxConnector* (AUXILIARY). The node containing the *jmxConnector* can also have any additional auxiliary classes.

6.5.2 Mapping to Java Objects

This specification only requires that the JMX Service URL is stored in LDAP. JMX API agents can additionally store a serialized JMX Remote API connector stub, but this is not required by this specification. Clients should only rely on the JMX Service URL. The JNDI API makes it possible for a client to use `StateFactories` and `ObjectFactories` [JNDI - Java Objects] to recreate a `JMXConnector` from the URL when performing a `lookup()`, even if there is no Java Object bound to the containing `DirContext`. Alternatively, a client can directly retrieve the *jmxServiceURL* attribute to obtain a `JMXConnector` from the `JMXConnectorFactory`. Whether the JNDI API `lookup()` returns a `JMXConnector` or a `DirContext` depends on the configuration settings on the client side (`InitialContext`), and remains local to that client.

6.5.3 Structure of the JMX Remote API Registration Tree

The actual structure of a directory varies from one organization to another. Each organization, or enterprise, has its own directory tree structure, with guidelines, policies, etc. In order for JMX API agents to be able to integrate with any pre-existing directory structure, this specification does not impose a fixed directory tree structure for registering agents and JMX Remote API connector servers. Connectors must simply be located in nodes of the class *jmxConnector*. This makes it possible for an organization to set up its own structure for registering agents in an LDAP server. For instance, if an organization has an existing directory containing a node for each host in its network, it could decide to register each agent below the node of the host it is running on.

6.5.4 Leasing

JNDI/LDAP does not provide any built-in lease service. If an agent goes down, its service URLs might remain in the directory server forever . The *jmxExpirationDate* attribute in the *jmxConnector* auxiliary class can be used to avoid that happening, as shown in CODE EXAMPLE 6-11:

CODE EXAMPLE 6-11 Leasing using the jmxExpirationDate Attribute

```
-- jmxExpirationDate attribute is a Generalized Time
-- see [RFC 2252] - or X.208 for a description of
--                  Generalized Time
( 1.3.6.1.4.1.42.2.27.11.1.6 NAME 'jmxExpirationDate'
   DESC 'Date at which the JMX Service URL will
         be considered obsolete and may be removed
         from the directory tree'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.24 SINGLE-VALUE )
```

A JMX API agent would have to update the *jmxExpirationDate* attribute periodically. A Directory administrator might then write a daemon that would remove the *jmxConnector* nodes (or more generically the *jmxServiceURL* attributes) for which the *jmxExpirationDate* is obsolete.

6.5.5 Code Templates

The following sections provide some code templates for the JNDI API lookup service

6.5.5.1 Discovering the LDAP Server

JNDI/LDAP does not provide any standard means for discovering the LDAP server. Assuming the standard port (389) on the local host is the entry point is usually not an option, since the LDAP server is usually centralized, rather than having one server per host. The JNDI API specifies a means to discover the LDAP server(s) through DNS [JNDI - LDAP Servers Discovery], but this is operating system dependent, and not always feasible either since the LDAP servers cannot always be registered in DNS. This specification thus does not address the issue of discovering the LDAP server.

The JNDI API tutorial gives an example of how to configure an `InitialContext` with a list of LDAP URLs [JNDI - Multi URL].

6.5.5.2 Registering a JMXServiceURL in the LDAP server

This specification does not impose any structure on the directory tree for registering JMX Service URLs. It is assumed that the JMX API agent knows where to register its connectors, either from configuration, or from some built-in logic adapted to the environment in which it is running. This specification defines the form of the data that is registered in the directory (the *how* rather than the *where*), so that any JMX Remote API client can look it up in a generic way. See CODE EXAMPLE 6-12.

CODE EXAMPLE 6-12 Registering a JMXServiceURL in the LDAP server

```
import javax.naming.InitialContext;
import javax.naming.directory.DirContext;
import javax.naming.directory.Attribute;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttributes;
...

// Create initial context
Hashtable env = new Hashtable(11);
env.put(InitialContext.PROVIDER_URL, ldapServerUrls);
env.put(...);
InitialContext root = new InitialContext(env);

// Assuming that the Directory Administrator has created a
// context for this agent, get the DN of that context
// from configuration (e.g. Java property)
// String myOwnLdapDN =
//        System.getProperty("com.sun.jmx.myapplication.dn");
String myOwnLdapDN = ....
DirContext myContext = (DirContext)root.lookup(myOwnLdapDN);

// Create connector server
JMXServiceURL jmxUrl = new
   JMXServiceURL("service:jmx:jmxmp://localhost:9999");
JMXConnectorServer connectorServer =
   JMXConnectorServerFactory.newJMXConnectorServer(jmxUrl, null, null);

// Prepare attributes for register connector server
Attributes attrs = new BasicAttributes();

// Prepare objectClass attribute: we're going to create
// a javaContainer (STRUCTURAL) containing a
// jmxConnector (AUXILIARY).
```

CODE EXAMPLE 6-12 Registering a `JMXServiceURL` in the LDAP server

```
Attribute objclass = new BasicAttribute("objectClass");
objclass.add("top");
objclass.add("javaContainer");
objclass.add("jmxConnector");
attrs.put(objclass);

// Add jmxServiceURL of the connector.
attrs.put("jmxServiceURL",jmxUrl.toString());

// Add jmxAgentName
attrs.put("jmxAgentName","MyAgentName");

// Add optional attributes, if needed
attrs.put("jmxProtocolType","jmxmp");
attrs.put("jmxAgentHost",InetAddress.getLocalHost().getHostName());

// Now create the sub context in which to register the URL
// of the JMXMP connector.
// (we assume that the subcontext does not exist yet -
//  ideally the agent should contain some more complex logic:
//  if the context already exists, simply modify its attributes,
//  otherwise, create it with its attributes).
myContext.createSubcontext("cn=service:jmx:rmi", attrs);
```

6.5.5.3 Looking up a JMX Service URL From the LDAP Server

CODE EXAMPLE 6-13 shows how to look up a JMX service URL from the LDAP server.

CODE EXAMPLE 6-13 Looking up a JMX Service URL From the LDAP Server

```
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.SearchResult;
import javax.naming.directory.SearchControls;
...

// Create initial context
Hashtable env = new Hashtable();
env.put(InitialContext.PROVIDER_URL, ldapServerUrls);
env.put(...);
InitialContext root = new InitialContext(env);

// Prepare search filter
String filter = "(&(objectClass=jmxConnector) (jmxServiceURL=*))";

// Prepare the search controls
SearchControls ctrls = new SearchControls();

// Want to get all jmxConnector objects, wherever they've been
// registered.
ctrls.setSearchScope(SearchControls.SUBTREE_SCOPE);

// Want to get only the jmxServiceURL (comment this line and
// all attributes will be returned).
ctrls.setReturningAttributes(new String[] { "jmxServiceURL" });

// Search...
final NamingEnumeration results = root.search("", filter, ctrls);

// Get the URL...
for (;results.hasMore();) {
    final SearchResult res    = (SearchResult) results.nextElement();
```

CODE EXAMPLE 6-13 Looking up a JMX Service URL From the LDAP Server

```
 final Attributes   attrs  = res.getAttributes();
    final Attribute    attr   = attrs.get("jmxServiceURL");
    final String       urlStr = (String)attr.get();

    // Make a connector...
 final JMXServiceURL url = new JMXServiceURL(urlStr);
    final JMXConnector conn =
        JMXConnectorFactory.newConnector(url,null);

    // Start using the connector...
 conn.connect(null);
    ...
}
```

6.6 Registration With Standards Bodies

In parallel with the completion of this specification, the following registrations are being made with standards bodies:

- For SLP, the `jmx` service type and associated service template are being registered with IANA
- For LDAP, the OIDs for the lookup attributes defined in Section 6.5.1 “LDAP Schema for Registration of JMX Connectors” on page 73 are defined in Sun’s OID namespace
- For the Jini networking technology, the entries for the lookup attributes are being defined through the Jini Community Decision Process (JDP)

CHAPTER 7

Summary of Environment Parameters

The environment parameters defined by this standard all begin with the string "`jmx.remote.`". Implementations that define further parameters can use one of the following conventions:

- The reverse domain name convention used by Java platform packages, for example "`com.sun.jmx.remote.something`"
- A name beginning with the string "`jmx.remote.x.`" (including the final period)

An implementation must not define non-standard parameters that begin with "`jmx.remote.`" unless they begin with "`jmx.remote.x.`".

Names beginning with "`jmx.remote.x.`" can be shared between different implementations. They are useful for agreed-on experimental extensions, but they run the risk of collision, where two implementations use the same name to mean two different things.

In TABLE 7-1, each parameter is defined by the following characteristics:

- The *name* after the initial "`jmx.remote.`" string
- The *type* that the associated value must have
- Whether the parameter applies to connector clients, to connector servers, or both
- For server parameters, whether the parameter is visible, that is whether it appears in the `Map` returned by `JMXConnectorServerMBean.getAttributes()`

TABLE 7-1 Environment Parameters

Name `jmx.remote.+`	Type	Client/ Server	Visible	Meaning
`authenticator`	`JMXAuthen -ticator`	Server	No	Object to authenticate incoming connections to the connector. See Section 3.4 “Basic Security With the RMI Connector” on page 38, and Section 4.3.6.1 “Global Properties of the Generic Connector” on page 53.
`context`	`Object`	Both	No	Context transmitted during handshake. See Section 4.3.6 “Properties Controlling Client and Server” on page 53
`credentials`	`Object`	Client	N/A	Client credentials to authenticate to the RMI connector server. See Section 3.4 “Basic Security With the RMI Connector” on page 38
`default.class. loader`	`Class Loader`	Both	No	Default class loader to deserialize objects received from the other end of a connection. See Section 2.11 “Class Loading” on page 28
`default.class. loader.name`	`Object Name`	Server	Yes	Name of class loader MBean that will be used to deserialize objects received from the client. See Section 2.11 “Class Loading” on page 28
`jndi.rebind`	`String`	Server	Yes	“true” or “false” according as an RMI stub object can overwrite an existing object at the JNDI address specified in a JMXServiceURL
`message.connection`	`MessageCo nnection`	Client	N/A	Object describing the transport used by the Generic Connector. See Section 4.1 “Pluggable Transport Protocol” on page 41

TABLE 7-1 Environment Parameters

Name jmx.remote.+	Type	Client/ Server	Visible	Meaning
message.connection.server	MessageConnectionServer	Server	No	Object describing the transport used by the Generic ConnectorServer. See Section 4.1 “Pluggable Transport Protocol” on page 41
object.wrapping	ObjectWrapping	Both	No	Object describing how parameters with non-default serialization are handled. See Section 4.2 “Pluggable Object Wrapping” on page 42
profiles	String	Both	Yes	List of profiles proposed (server) or required (client) by the connector. See Section 4.3.6 “Properties Controlling Client and Server” on page 53
protocol.provider.class.loader	Class Loader	Client	N/A	See JMXConnectorFactory documentation.
protocol.provider.pkgs	String	Client	N/A	See JMXConnectorFactory documentation.
rmi.client.socket.factory	RMIClient Socket Factory	Server	No	Client socket factory for connections to the RMI connector. See Section 3.4 “Basic Security With the RMI Connector” on page 38
rmi.server.socket.factory	RMIServer Socket Factory	Server	No	Server socket factory for connections to the RMI connector. See Section 3.4 “Basic Security With the RMI Connector” on page 38
sasl.authorization.id	String	Client	N/A	Authorization ID when this is different from the authentication ID . See Section 4.3.6 “Properties Controlling Client and Server” on page 53
sasl.callback.handler	Callback Handler	Both	No	Callback handler for SASL mechanism. See Section 4.3.6 “Properties Controlling Client and Server” on page 53

TABLE 7-1 Environment Parameters

Name `jmx.remote.+`	Type	Client/ Server	Visible	Meaning
`server.address. wildcard`	`String`	Server	Yes	"`true`" or "`false`" according to whether connector server should listen on all local network interfaces or just one. See `JMXMPConnectorServer` documentation.
`tls.enabled.cipher .suites`	`String`	Both	Yes	TLS cipher suites to enable. See Section 4.3.6 “Properties Controlling Client and Server” on page 53
`tls.enabled. protocols`	`String`	Both	Yes	TLS protocols to enable. See Section 4.3.6 “Properties Controlling Client and Server” on page 53
`tls.need.client. authentication`	`String`	Server	Yes	"`true`" or "`false`" according to whether connector server requires client authentication. See Section 4.3.6 “Properties Controlling Client and Server” on page 53
`tls.socket.factory`	`SSLSocket Factory`	Both	No	TLS socket factory for this connector. See Section 4.3.6 “Properties Controlling Client and Server” on page 53
`tls.want.client. authentication`	`String`	Server	Yes	"`true`" or "`false`" according to whether connector server requires client authentication if supported by the negotiated cipher suite. See Section 4.3.6 “Properties Controlling Client and Server” on page 53

APPENDIX **A**

Service Templates

This appendix defines the service templates that describe the `service:jmx` services in conformance to [RFC 2609]. These service template are a formal description of the bindings between the Service Location Protocol and JSR 160 connectors.

Note – The following templates are a copy of the submissions that have been made to svrloc-list@iana.org.

A.1 Service Template for the `service:jmx` Abstract Service Type

- **Template Filename:** jmx.1.0.en
- **Name of submitter:** JSR-160 Expert Group <jsr-160-comments@jcp.org>
- **Language of service template:** en
- **Security considerations:**

  Security is defined by each of the concrete service types.

  See those templates for further details.
- **TemplateText:**

CODE EXAMPLE A-1 Service template for the `service:jmx` Abstract Service Type

```
Name of submitter: JSR-160 Expert Group <jsr-160-comments@jcp.org>
Language of service template: en
Security considerations:
    Security is defined by each of the concrete service types.
    See those templates for further details.
```

CODE EXAMPLE A-1 Service template for the `service:jmx` Abstract Service Type

```
TemplateText:
------------------------template begins here----------------------
template-type=jmx

template-version=1.0
template-description=
     This is an abstract service type.  The purpose of the jmx service
     type is to organize in a single category all JMX Connectors that
     make it possible to access JMX Agents remotely.
     JMX Connectors are defined by the Java Specification Request 160
     (JSR 160). More information on JSR 160 can be obtained from the
     Java Community Process Home Page at:
          http://www.jcp.org/en/jsr/detail?id=160

template-url-syntax=
     url-path=  ;  Depends on the concrete service type.


AgentName= string L
# The name of the JMX Agent - see JSR 160 specification.

ProtocolType= string O L
# The type of the protocol supported by the JMX Connector.
# Currently only two protocols are mandatory in the specification: "rmi" and
# "iiop". A third optional protocol is also standardized: "jmxmp".
# However this could be extended in the future to support other types
# of protocols, e.g. "http", "https", "soap", "beep", etc...
# Thus, the allowed values of this attribute are at least "rmi" and "iiop"
# for every implementation; additionally "jmxmp" for implementations that
# support it; and other protocol names that are understood by client and
# server.
# The value of this attribute is the same as the protocol name that appears
# after "service:jmx:" in the Service URL.  Registering the ProtocolType
# attribute means clients can search for connectors of a particular type.

AgentHost= string O M L
# The host name or IP address of the host on which the JMX Agent is running.
# If multiple values are given they must be aliases to the same host.

Property= string O M L
# Additional properties qualifying the agent, in the form of Java-like
# properties, e.g. "com.sun.jmx.remote.connect.timeout=200"
```

CODE EXAMPLE A-1 Service template for the `service:jmx` Abstract Service Type

```
# Note that in order to include '=' in an attribute value, it must be
# escaped.  Thus the example would be encoded as
# "com.sun.jmx.remote.connect.timeout\3D200"

------------------------template ends here----------------------
```

A.2 Service Template for the `service:jmx:jmxmp` Concrete Service Type

- **Template Filename:** jmx:jmxmp.1.0.en
- **Name of submitter:** JSR-160 Expert Group <jsr-160-comments@jcp.org>
- **Language of service template:** en
- **Security considerations:**

  Security for the JMXMP connector is defined by JSR 160 specification and is based on SASL mechanisms.

  For further details please refer to JSR 160 specification available at

  http://www.jcp.org/en/jsr/detail?id=160
- **TemplateText:**

CODE EXAMPLE A-2 Service Template for the `service:jmx:jmxmp` Concrete Service Type

```
Name of submitter: JSR-160 Expert Group <jsr-160-comments@jcp.org>
Language of service template: en
Security considerations:
    Security for the JMXMP connector is defined by JSR 160
    specification and is based on SASL mechanisms.
    For further details please refer to JSR 160 specification
    available at http://www.jcp.org/en/jsr/detail?id=160

TemplateText:
------------------------template begins here----------------------

template-type=jmx:jmxmp
```

CODE EXAMPLE A-2 Service Template for the `service:jmx:jmxmp` Concrete Service Type

```
template-version=1.0

template-description=
    This template describes the JMXMP Connector defined by JSR 160.
    More information on this connector can be obtained from the
    JSR 160 specification available from the JCP Home Page at:
    http://www.jcp.org/en/jsr/detail?id=160

template-url-syntax=
    url-path= ; There is no URL path defined for a jmx:jmxmp URL.

# Example of a valid Service URL:
# service:jmx:jmxmp://myhost:9876
# There are no default values for the host or port number, so in
# general these must be supplied when registering the URL.
-------------------------template ends here-----------------------
```

A.3 Service Template for the `service:jmx:rmi` Concrete Service Type

- **Template Filename:** jmx:rmi.1.0.en
- **Name of submitter:** JSR-160 Expert Group <jsr-160-comments@jcp.org>
- **Language of service template:** en
- **Security considerations:**

  Java Specification Request (JSR) 160 defines a secure configuration of the jmx:rmi connector, based on SSL socket factories.

  For further details please refer to JSR 160 specification available at

  http://www.jcp.org/en/jsr/detail?id=160

- **TemplateText:**

CODE EXAMPLE A-3 Service Template for the `service:jmx:rmi` Concrete Service Type

```
Name of submitter: JSR-160 Expert Group <jsr-160-comments@jcp.org>
Language of service template: en
Security considerations:
    Java Specification Request (JSR) 160 defines a secure
    configuration of the jmx:rmi connector, based on SSL socket
    factories.
    For further details please refer to JSR 160 specification
    available at http://www.jcp.org/en/jsr/detail?id=160

TemplateText:
------------------------template begins here----------------------
template-type=jmx:rmi

template-version=1.0
template-description=
    This template describes the RMI Connector defined by JSR 160.
    More information on this connector can be obtained from the
    JSR 160 specification available from the JCP Home Page at:
        http://www.jcp.org/en/jsr/detail?id=160

template-url-syntax=
    url-path        =   jndi-path / stub-path
    stub-path       =   "/stub/" *xchar
      ;  serialized RMI stub encoded as BASE64 without newlines
    jndi-path       =   "/jndi/" *xchar
      ;  name understood by JNDI API, shows where RMI stub is stored
    ; The following rules are extracted from RFC 2609
    safe            =   "$" / "-" / "_" / "." / "~"
    extra           =   "!" / "*" / "'" / "(" / ")" / "," / "+"
    uchar           =   unreserved / escaped
    xchar           =   unreserved / reserved / escaped
    escaped         =   1*(`\' HEXDIG HEXDIG)
    reserved        =   ";" / "/" / "?" / ":" / "@" / "&" / "=" / "+"
    unreserved      =   ALPHA / DIGIT / safe / extra

# Examples of the stub form:
# service:jmx:rmi://myhost:9999/stub/rO0ABX<270 chars deleted>gAAAeA==
# service:jmx:rmi:///stub/rO0ABX<270 chars deleted>gAAAeA==
# This form contains the serialized form of the Java object representing
# the RMI stub, encoded in BASE64 without newlines.  It is generated by
# the connector server, and is not intended to be human-readable.
```

CODE EXAMPLE A-3 Service Template for the `service:jmx:rmi` Concrete Service Type

```
#
# Examples of the JNDI form:
# service:jmx:rmi://myhost:9999/jndi/ldap://namehost:389/a=b,c=d
# service:jmx:rmi:///jndi/ldap://namehost:389/a=b,c=d
# If the client has an appropriate JNDI configuration, it can use
# a URL such as this:
# service:jmx:rmi:///jndi/a=b,c=d
#
# In both the /stub/ and /jndi/ forms, the hostname and port number
# (myhost:9999 in the examples) are not used by the client and, if
# present, are essentially comments.  The connector server address
# is actually stored in the serialized stub (/stub/ form) or in the
# directory entry (/jndi/ form).
#
# For more information, see the JSR 160 specification, notably the
# package javax.management.remote.rmi.
-------------------------template ends here-----------------------
```

A.4 Service Template for the `service:jmx:iiop` Concrete Service Type

- **Template Filename:** jmx:iiop.1.0.en
- **Name of submitter:** JSR-160 Expert Group <jsr-160-comments@jcp.org>
- **Language of service template:** en
- **Security considerations:**

  There is no special security defined for the `jmx:iiop` connector, besides the mechanisms provided by RMI over IIOP itself. In its default configuration, the `jmx:iiop` connector is not secure. Applications that are concerned with security should therefore not advertise their `jmx:iiop` connectors through this template, unless they have taken the appropriate steps to make it secure.

  For further details please refer to JSR 160 specification available at

  http://www.jcp.org/en/jsr/detail?id=160

- **TemplateText:**

CODE EXAMPLE A-4

```
Name of submitter: JSR-160 Expert Group <jsr-160-comments@jcp.org>
Language of service template: en
Security considerations:
There is no special security defined for the jmx:iiop connector,
besides the mechanisms provided by RMI over IIOP itself. In its
default configuration, the jmx:iiop connector is not
secure. Applications that are concerned with security should therefore
not advertise their jmx:iiop connectors through this template, unless
they have taken the appropriate steps to make it secure.

For further details please refer to JSR 160 specification available at
http://www.jcp.org/en/jsr/detail?id=160

TemplateText:
------------------------template begins here----------------------

template-type=jmx:rmi-iiop
template-version=1.0

template-description=
    This template describes the RMI/IIOP Connector defined by JSR 160.
    More information on this connector can be obtained from the
    JSR 160 specification available from the JCP Home Page at:
    http://www.jcp.org/en/jsr/detail?id=160
template-url-syntax=
    url-path        =   jndi-path / ior-path
    ior-path        =   "/ior/IOR:" *HEXDIG
      ;  CORBA IOR
    jndi-path       =   "/jndi/" *xchar
      ;  name understood by JNDI API, shows were RMI/IIOP stub is stored
    ; The following rules are extracted from RFC 2609
    safe            =   "$" / "-" / "_" / "." / "~"
    extra           =   "!" / "*" / "'" / "(" / ")" / "," / "+"
    uchar           =   unreserved / escaped
    xchar           =   unreserved / reserved / escaped
    escaped         =   1*(`\' HEXDIG HEXDIG)
    reserved        =   ";" / "/" / "?" / ":" / "@" / "&" / "=" / "+"
    unreserved      =   ALPHA / DIGIT / safe / extra

# Examples of the IOR form:
# service:jmx:iiop://myhost:9999/ior/IOR:000000000000003b<350 chars deleted>00
```

CODE EXAMPLE A-4

```
# service:jmx:iiop:///ior/IOR:000000000000003b<350 chars deleted>00
# This form contains the CORBA IOR for the remote object representing
# the connector server.  It is generated by the connector server, and
# is not intended to be human-readable.
#
# Examples of the JNDI form:
# service:jmx:iiop://myhost:9999/jndi/ldap://namehost:389/a=b,c=d
# service:jmx:iiop:///jndi/ldap://namehost:389/a=b,c=d
# If the client has an appropriate JNDI configuration, it can use
# a URL such as this:
# service:jmx:iiop:///jndi/a=b,c=d
#
# In both the /ior/ and /jndi/ forms, the hostname and port number
# (myhost:9999 in the examples) are not used by the client and, if
# present, are essentially comments.  The connector server address is
# actually stored in the IOR (/ior/ form) or in the directory entry
# (/jndi/ form).
#
# For more information, see the JSR 160 specification, notably the
# package javax.management.remote.rmi.
-------------------------template ends here----------------------
```

CODE EXAMPLE A-5 Service Template for the `service:jmx:iiop` Concrete Service Type

```
-------------------------template begins here----------------------

template-type=jmx:rmi-iiop
template-version=1.0

template-description=
    This template describes the RMI/IIOP Connector defined by JSR 160.
    More information on this connector can be obtained from the
    JSR 160 specification available from the JCP Home Page at:
    http://www.jcp.org/en/jsr/detail?id=160
template-url-syntax=
    url-path        =   jndi-path / ior-path
    jndi-path       =   "/jndi/" *xchar
      ;  name understood by JNDI API, shows were RMI/IIOP stub is stored
    ior-path        =   "/ior/IOR:" *HEXDIG
      ;  CORBA IOR
    ; The following rules are extracted from RFC 2609
    safe            =   "$" / "-" / "_" / "." / "~"
    extra           =   "!" / "*" / "'" / "(" / ")" / "," / "+"
```

CODE EXAMPLE A-5 Service Template for the `service:jmx:iiop` Concrete Service Type

```
    uchar           =   unreserved / escaped
    xchar           =   unreserved / reserved / escaped
    escaped         =   1*(‘” HEXDIG HEXDIG)
    reserved        =   ";" / "/" / "?" / ":" / "@" / "&" / "=" / "+"
    unreserved      =   ALPHA / DIGIT / safe / extra

------------------------template ends here-----------------------
```

APPENDIX **B**

Non-standard environment parameters

This appendix lists non-standard environment parameters that are understood by the Reference Implementation of this specification. These attributes are defined in the `jmx.remote.x` namespace. As described in Chapter 7 “Summary of Environment Parameters”, this namespace is reserved for non-standard extensions to the parameters defined in this specification.

Implementations are not required to support the parameters defined here. However, implementors are encouraged to use the same name and semantics where applicable.

The format of this table is the same as for the table in TABLE 7-1 on page 82.

Where the type of an attribute is “integer”, the value can be of any subclass of `java.lang.Number`, typically `Integer` or `Long`. It can also be a string, which is parsed as a decimal integer.

TABLE B-1 Environment Parameters

Name jmx.remote.x.+	Type	Client/ Server	Visible	Meaning
access.file	String	Server	No	Name of a file containing access levels for simple RMI and JMXMP connector access control. Uses Properties file format: property name is user name, property value is “readonly” or “readwrite”.
password.file	String	Server	No	Name of a file containing username and password entries for RMI authentication. Uses Properties file format: property name is user name, property value is password.
notification. buffer.size	integer	Server	Yes	Minimum size of the buffer that stores notifications for one or more connector servers. A connector server will remember a notification if there have not been this many others since it was sent.
notification. fetch.max	integer	Client	N/A	Maximum number of notifications that a client (RMI or JMXMP) will request in a single fetchNotifications request.
notification. fetch.timeout	integer	Client	N/A	Timeout in milliseconds that a client (RMI or JMXMP) will specify in each fetchNotifications request.
client. connection. check.period	integer	Client	N/A	Time in milliseconds between client probes of an open connection. The client will do a harmless operation on the connection with this period in order to detect communication problems on otherwise-idle connections. The value can be negative or zero to disable this probing.
server.max. threads	integer	Server	Yes	Maximum number of server threads for each JMXMP connection. If more than this many requests arrive simultaneously, the surplus ones will be blocked until others complete.
server.min. threads	integer	Server	Yes	Minimum number of server threads for each JMXMP connection. The server will keep at least this many threads alive, even if the current number of requests is less than this.
request. waiting. timeout	integer	Client	N/A	Timeout in milliseconds for the response to each JMXMP client request. If a response does not arrive within this time, the connection is assumed to be broken and is terminated. Specifying too short a value will cause this to happen for requests whose treatment happens to be slow. Default value is infinite.

References

J

JAAS

Sun Microsystems, Java Authentication and Authorization Service (JAAS), http //java.sun.com/products/jaas/

Jini

Sun Microsystems, Jini Network Technology, ,http //wwws.sun.com/software/jini/

JNDI ,

Sun Microsystems, Java Naming and Directory Interface,http //java.sun.com/products/jndi/

JNDI - Java Objects

JNDI Tutorial, Java Objects and the Directory, http //java.sun.com/products/jndi/tutorial/objects/index.html

JNDI - Java Schema

JNDI Tutorial, Java Schema for the Directory,http //java.sun.com/products/jndi/tutorial/config/LDAP/java.schema

JNDI - LDAP Servers Discovery

JNDI Tutorial, Automatic Discovery of LDAP Servers, ,http //java.sun.com/products/jndi/tutorial/ldap/connect/create.html#AUTO

JNDI - Multi URL

JNDI Tutorial, How to specify more than one URL when creating initial context., ,http //java.sun.com/products/jndi/tutorial/ldap/misc/src/MultiUrls.java

JSR 140

Nick Briers, et al, Service Location Protocol (SLP) API for Java, 2001,http //www.jcp.org/en/jsr/detail?id=140

JSR28

Lee, Rosanna, et al, Java SASL Specification, http //jcp.org/en/jsr/detail?id=28

JSR3

JSR3 Sun Microsystems et al, Java Management Extensions Specification, version 1.2, 2002 http

//jcp.org/en/jsr/detail?id=3
JSSE
Sun Microsystems, Java Secure Socket Extension (JSSE), ,http
//java.sun.com/products/jsse/
L
LDAP Thread in the JNDI Tutorial
Tips for LDAP Users, http
//java.sun.com/products/jndi/tutorial/ldap/index.html
R
RFC , , ,
RFC 2608
E. Guttman, et al, Service Location Protocol, Version 2, 1999,http
//www.ietf.org/rfc/rfc2608.txt
RFC 2609
E. Guttman, C. Perkins, J. Kempf, 1999,http
//www.ietf.org/rfc/rfc2609.txt
RFC 2614
J. Kempf, E. Guttman., An API for Service Location, 1999,http
//www.ietf.org/rfc/rfc2614.txt
RFC 2713
V. Ryan, et al., Schema for Representing Java Objects in an LDAP Directory, 1999,http
//www.ietf.org/rfc/rfc2713.txt
RFC2222
Myers, J, Simple Authentication and Security Layer (SASL), 1997,ftp
//ftp.rfc-editor.org/in-notes/rfc2222.txt
RMI/SSL
Sun Microsystems, Using RMI with SSL, 2001
S
Serial
Sun Microsystems, Inc, Java Object Serialization Specification
SLP
IETF SVRLOC working group, Service Location Protocol, http
//www.srvloc.org/
SLP White Paper
C. Perkins, http
//playground.sun.com/srvloc/slp_white_paper.html