

Communicating with JMX agents

- Using the RMI adapter from Sun
- Creating a Jini connector
- Creating a TCP adapter

You had your first exposure to working with an MBean server by using the HTML adapter you registered on the server. Previous chapters reminded you how JMX uses protocol adapters and connectors to enable a JMX agent for use by the outside world.

You studied the overall agent architecture in chapter 1 and explored the MBean server in greater detail in chapter 8. This chapter covers another component of JMX agents: protocol adapters and connectors. In this chapter, we will discuss two connectors that will enable you to distribute your agents across a network using Java Remote Method Invocation (RMI) and the Jini network technology. We will also spend some time discussing using TCP and Simple Network Management Protocol (SNMP) to enable access to JMX agents.

By allowing clients of your agent to contact the agent from remote locations, you greatly increase the agent's usefulness. By using connectors and adapters, you can collocate agents with managed resources and contact them from remote locations. Thus you can use web browsers, hand-held devices, and so forth to stay in contact with your managed resources.

As previously mentioned, such remote communication is particularly useful in a monitoring context. You can install a JMX agent in a hosted application environment and stay in communication with it over your network. This ability lets you maintain reliable, real-time status. Figure 9.1 depicts this scenario; you should recognize components of this figure from previous chapters.

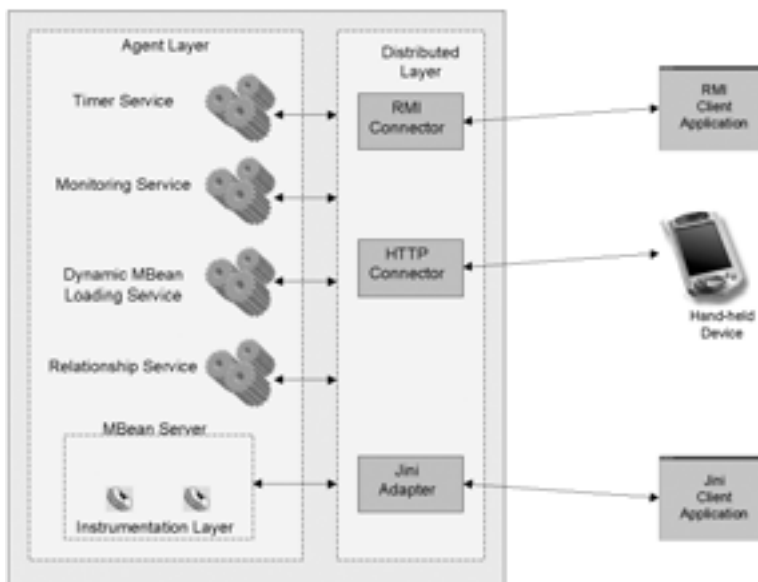


Figure 9.1
Contacting a JMX agent with a remote client by using a connector

When discussing protocol adapters and connectors as means of exposing your JMX agents to management tools, it is important to understand the differences between them.

9.1 Comparing connectors and protocol adapters

Protocol adapters and connectors are very similar in that they serve the same overall purpose: to open a JMX agent to managing entities. The difference between them is how they go about it. Protocol adapters generally must listen for incoming messages that are constructed in a particular protocol like HTTP or SNMP. In this sense, protocol adapters are made up of only one component that resides in the agent at all times.

Connectors, on the other hand, are made up of two components: one component resides in the JMX agent, and the other is used by client-side applications. Clients use the client-side connector component to contact the server-side component and communicate with a JMX agent. In this manner, connectors hide the actual protocol being used to contact the agent; the entire process happens between the connector's two components.

9.2 Connecting by using RMI

Recall from chapter 2's discussion of Sun Microsystems' JMX Reference Implementation (RI) that the RI includes a `jmx` folder and a `contrib` folder. The `contrib` folder contains the RMI connector that you included in the `JMXBookAgent` from chapter 3. This section is intended to make you more familiar with how the RMI connector works. It is unsupported in the RI, but it is also contained in Sun Microsystems' commercial JMX product, the Java Dynamic Management Kit (JDMK). (For more information about the JDMK, go to <http://www.javasoft.com>.)

9.2.1 Using the RMI connector

Figure 9.2 illustrates the components of the RMI connector. It is an MBean registered on an MBean server, just like the HTML adapter you have already used. However, whereas you used a web browser previously to contact the HTML adapter, the RMI connector comes with an RMI client.

You can use the RMI connector client to connect to the RMI server MBean and invoke methods that correspond directly to methods on the MBean server in which the MBean is registered. For example, after connecting to the server with an RMI connector client `rmiClient`, you could invoke the method

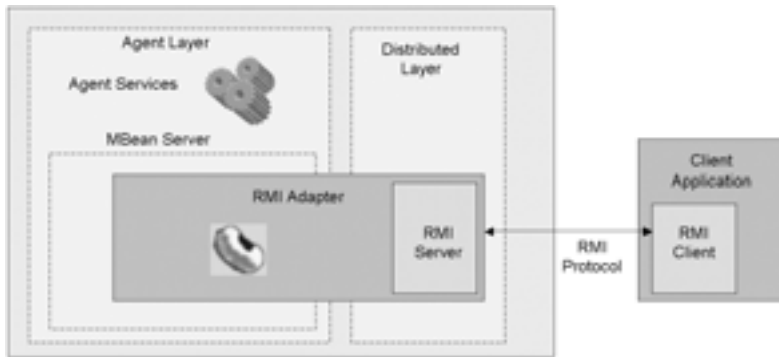


Figure 9.2 The components of the RMI connector included in the JMX RI from Sun Microsystems. The RMI connector uses both a server object and a client object.

`rmiClient.getMBeanCount()` to acquire the number of MBeans running on the remote MBean server. You will find every method on the RMI client that you would find in the `MBeanServer` interface.

9.2.2 Creating the RMI server MBean

The server portion of the RMI connector is contained in the `RmiConnectorServer` class. To create the server, you need to perform the following three steps:

- 1 Create an instance of `RmiConnectorServer` using one of its four constructors. The different constructors let you specify different values for the server registration port and service name.
- 2 Register the connector and the MBean server.
- 3 Invoke the connector's `start()` method. The `start()` method tells the server to bind to an RMI registry and prepare itself to receive client calls.

Reexamining the `JMXBookAgent` class

When you created the `JMXBookAgent` class in chapter 3, you gave it a `startRMIConnector()` method that added the RMI connector MBean to the agent. However, in that chapter, we did not discuss what took place in code. Listing 9.1 shows the method again; let's examine it.

Listing 9.1 The `startRMIConnector()` method of the `JMXBookAgent` class

```
protected void startRMIConnector()
{
    RmiConnectorServer connector = new RmiConnectorServer();
    ObjectName connectorName = null;
```

```

try
{
    connectorName = new ObjectName(
        "JMXBookAgent:name=RMIConnector");
    server.registerMBean( connector, connectorName );
    connector.start();
}
catch(Exception e)
{
    e.printStackTrace();
}
}

```

The `JMXBookAgent` class imports the `com.sun.jdmk.comm` package in order to obtain the `RmiConnectorServer` class. It contains the classes contributed by Sun Microsystems in Sun's JMX RI.

The class uses the default constructor of the `RmiConnectorServer` class. The constructor tells the server to use the default port and service name. However, as mentioned previously, this is not the `RmiConnectorServer` class's only constructor. Table 9.1 lists the constructors and their arguments.

Table 9.1 The constructors of the `RmiConnectorServer` class

Constructor	Description
<code>RmiConnectorServer()</code>	The default constructor
<code>RmiConnectorServer(int port)</code>	Specifies a new port for binding to an RMI registry
<code>RmiConnectorServer(String name)</code>	Specifies a name for the object registered on the RMI registry
<code>RmiConnectorServer(int port, String name)</code>	Specifies both a new registry port and new remote object name

By using one of the other constructors, you can control the RMI registry the server will bind to, and you can change the name of the remote object that will be registered on the registry.

9.2.3 Connecting to the RMI server

Now that we have examined a basic agent that uses an RMI connector, let's look at the RMI connector client contained in the class `RmiConnectorClient`. As mentioned earlier, this class declares methods that correspond to every method

available on an MBean server. The following example shows you how to connect to the RMI connector server running on the `JMXBookAgent`.

Reexamining the `RMIClientFactory` class

In chapter 3, you created the `RMIClientFactory` class. Recall that you use this class to acquire an RMI client in which to contact your `JMXBookAgent` class. Listing 9.2 lists the class again.

Listing 9.2 `RMIClientFactory.java`

```
package jmxbook.ch3;

import javax.management.*;
import com.sun.jdmk.comm.*;

public class RMIClientFactory
{
    public static RmiConnectorClient getClient()
    {
        RmiConnectorClient client = new RmiConnectorClient();
        RmiConnectorAddress address = new RmiConnectorAddress();
        System.out.println("\t\tTYPE\t= " +
            address.getConnectorType ());
        System.out.println("\t\tPORT\t= " + address.getPort());
        System.out.println("\t\tHOST\t= " + address.getHost());
        System.out.println("\t\tSERVER\t= " + address.getName());

        try
        {
            client.connect( address );
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }

        return client;
    }
}
```

To tell the `RmiConnectorClient` object where to find the `RmiConnectorServer`, you need to use the `RmiConnectorAddress` class. This class encapsulates host, port, and lookup name values that tell the client object where to find the RMI registry and look up the remote object of the RMI connector. If you created the `RmiConnectorServer` using the default constructor, then you can create the address

object with its default constructor. Both classes contain the same default values for host, port, and lookup name. The default values are the following:

- *Host*—Defaults to the local host value
- *Port*—Default value is contained in the static variable `ServiceName.RMI_CONNECTOR_PORT`
- *Lookup name*—Defaults to the value of `ServiceName.RMI_CONNECTOR_SERVER`

After creating the `RmiConnectorClient` object, you invoke its `connect()` method. This method tells the client object to make a connection with the server-side component of the RMI connector. After successfully connecting to the agent, you can return the client reference—ready for use.

9.2.4 Additional uses for the RMI connector

In addition to providing RMI connectivity to a JMX agent for invoking methods on a remote MBean server, the RMI connector offers some other useful features. The remaining features of the RMI connector are as follows:

- *Remote notifications*—The RMI connector will transmit notifications emitted from the remote server to the remote client.
- *Connector heartbeat*—Connector clients can emit a heartbeat to monitor the connection to the connector server. Doing so allows agents and client to retry or clean up bad connections.
- *Client context checking*—This feature allows the server to verify that a client has the correct context before invoking requested operations.

Recall that the RMI connector is contributed unsupported to the Sun JMX RI. The RMI connector is part of Sun's commercial product, the JDMK. With that in mind, the following sections briefly describe the three previously listed features in more detail.

Retrieving notifications

The RMI connector provides two ways to receive notifications from a remote MBean server. When an RMI connector client connects to an RMI connector server, it can specify a notification receiving mode via the client's `setMode()` method. The `setMode()` method takes a single parameter: either `ClientNotificationHandler.PUSH_MODE` or `ClientNotificationHandler.PULL_MODE`. The first value indicates that notifications from the remote JMX agent will be pushed to the RMI connector client, and the second value indicates that the RMI connector client will pull notifications from the remote agent.

To receive notifications from a particular MBean, you simply invoke the `addNotificationListener()` method of the client. Because all `Notification` objects are serializable, they can be transmitted over RMI to interested clients.

Connector heartbeat

The RMI connector uses a notification system to detect the health of the client and/or server portions of the connector. When using an `RmiConnectorClient` object, you can add a notification listener for receiving `HeartBeatNotification` objects. A `HeartBeatNotification` object can indicate several conditions about a connection to the RMI connector server, as listed in table 9.2. The values in the `Notification Type` column are `public static` members of the `HeartBeatNotification` class.

Table 9.2 Notification types used by a `HeartBeatNotification` notification

Notification type	Description
<code>CONNECTION_ESTABLISHED</code>	A connection has been made between the client and server.
<code>CONNECTION_LOST</code>	A connection has died.
<code>CONNECTION_REESTABLISHED</code>	A connection was temporarily unavailable, but is now connected.
<code>CONNECTION_RETRYING</code>	The client is trying to reestablish a dead connection.
<code>CONNECTION_TERMINATED</code>	The connection has been closed.

You acquire the condition value by invoking the `getType()` method of the notification.

Client context checking

The last feature of the RMI connector covered in this chapter is the concept of a client context checker. A context checker ensures that a client passes a predefined test before it can invoke any methods on the remote MBean server.

The client must set an `OperationalContext` object into its `RmiConnectorClient` object. The connector client object will pass this context to the RMI connector server, which uses it to decide whether to complete a client's request on an MBean server. To do so, the server uses an instance of the class `MBeanServerChecker`.

An `MBeanServerChecker` object encapsulates an `MBeanServer` object and contains a `check` method for every method declared by the `MBeanServer` class. For instance, for a client that attempted to invoke `create Bean()` on a remote MBean server, the `MBeanServerChecker` would first invoke the method `checkCreate()`. This method would verify the client's `OperationalContext` in some way and, if it

were valid, would invoke the method on the MBean server. To provide your own implementation for the check methods, you would provide a subclass to the `MBeanServerChecker` class.

9.3 Connecting to agents using Jini

The RMI connector we just discussed should give you a good idea what can be accomplished by making your JMX agents remotely accessible. As you discovered, the RMI connector not only gives you the ability to invoke MBean server methods, but also lets you receive notifications.

However, you might have noticed one drawback to using the RMI connector: you must know the address of the RMI connector server. That is, you have to be able to tell your `RmiConnectorClient` object where to look up the remote server object.

To get around this issue, you can build a Jini connector. By using Jini, you can distribute a JMX agent just like the RMI connector does, without requiring clients to know the exact location of the agent. Jini enables developers to write services that can describe themselves and can be discovered by clients.

For instance, clients wishing to interact with a JMX agent can construct a Jini connector client, enter a few search parameters, and locate the nearest matching agent, as illustrated in figure 9.3.

Your Jini connector will advertise itself by using the value of the default domain name of the MBean server in which it is registered. The Jini connector client will do a search for agents by using the domain name as a search parameter. Later, you can expand the search capabilities as needed. As the chapter continues,

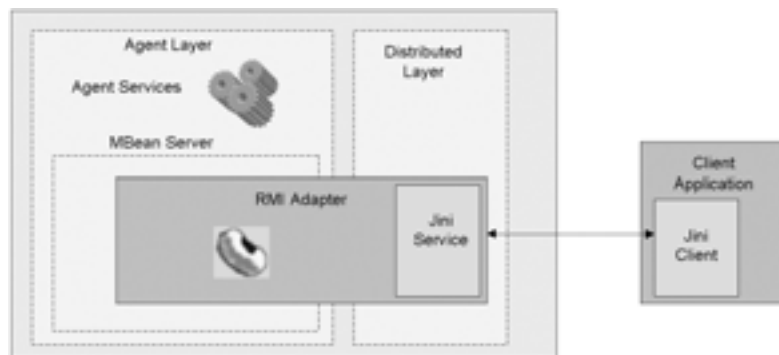


Figure 9.3 The Jini connector makes the JMX agent available to a greater client audience by allowing itself to be discovered.

we will discuss more connector scenarios that use the Jini connector. The connector will be made up of three components: an MBean, a Jini service, and a Jini client. The following section explores these three components in detail.

9.3.1 Components of the Jini connector

As the previous section mentioned, you need the following three components to create this connector:

- *MBean*—The MBean allows the connector to be managed through the MBean server (like the RMI connector and HTML adapter MBeans).
- *Jini service*—The service is created by the MBean. It allows clients to connect to the agent.
- *Jini client*—People use the client class to locate the Jini service from the agent.

The following sections describe the role each component plays in the connector. Then, we will begin to examine the code.

The MBean

The role of the MBean in the connector is to set up and manage the Jini service. The MBean gives the agent the capability to control the Jini service, including setting up its description and deciding when to make the service available for discovery. The MBean contains a reference to its MBean server, which allows the Jini service to perform callbacks to the MBean server methods. (You will learn more about this process when we examine the code.) The MBean component will be defined by the `JINIServerMBean` interface and the `JINIServer` class (which implements the `JINIServerMBean` interface).

The Jini service

The Jini service implements methods that correspond to each of the methods found in the MBean server interface. As you will see in the code to follow, this process allows the Jini client to forward its method invocations to the MBean server via the Jini service. The Jini service is defined in a class named `JINIConnectorImpl`, which implements the interface `JINIConnector`. The interface is a remote interface used in a Java RMI environment, enabling clients to make remote invocations on the Jini service.

The connector client

Client-side objects use the Jini client to discover and use the Jini connector service (and therefore use the JMX agent). Like the Jini service, it contains methods

that correspond to the methods of the `MBeanServer` interface. Invocations of these methods are forwarded to the Jini service, which forwards them to the MBean server of the JMX agent. In addition, the connector client shields the developer from the process of finding the Jini service. The `JINIConnectorClient` class defines the Jini client.

9.3.2 Writing the Jini connector

Now that we have discussed the components that form the Jini connector, let's begin writing the code. You will create the components in the order they were presented: the MBean, the Jini service, and then the Jini client. All the classes are in the package `jmxbook.ch9`. After you write all the classes, we will go over what you need to do to compile and run them. To test the connector, you will need to write some code for your `JMXBookAgent` class to include an instance of the Jini connector.

Writing the MBean

With the MBean, you must decide what attributes and operations you want to expose in order to make the Jini service configurable and more useful. Table 9.3 lists the attributes and operations exposed by the `JINI_SERVER_MBEAN` interface. Remember from chapter 4 that this type of interface indicates you are creating a Standard MBean.

Table 9.3 Attributes and operations exposed by the `JINI_SERVER_MBEAN` interface

Attribute/operation	Description
Domain	Read-only attribute that indicates the domain of the agent that contains this connector.
EntryName	Read/write attribute that supplies the Jini service started by the MBean with an identifier. This attribute is optional, but providing a value gives clients a way to refine their search for the service (it makes the Jini service more unique).
Groups	Read/write attribute that indicates which lookup service groups the Jini service will register with. (For more information about the lookup service, go to http://www.javasoft.com .)
enableConnections	Operation that tells the MBean to start the Jini service.

Table 9.3 gives a good view of what the MBean will be like. Now that you know the exposed attributes and operations of the MBean, look at the `JINI_SERVER_MBEAN` interface:

```
package jmxbook.ch9;

public interface JINI_SERVER_MBEAN
```

```

{
    public String getDomain();
    public String getEntryName();
    public void setEntryName( String name );
    public String[] getGroups();
    public void setGroups( String[] groups );
    public void enableConnections();
}

```

After creating the interface, you need to implement it with the `JINIserver` MBean class. Recall from the previous sections that the `JINIserver` class will create the Jini service when requested by the JMX agent. Listing 9.3 shows the `JINIserver` class. (Starting with this class, you will notice a lot of Jini-related packages and classes; this discussion goes into detail for many but not all of the Jini-related issues. If you need to take time to read more about Jini, check the documents at <http://www.javasoft.com>.)

Listing 9.3 `JINIserver.java`

```

package jmxbook.ch9;

import javax.management.*;
import java.rmi.*;
import java.util.*;
import net.jini.discovery.*;
import net.jini.core.lookup.*;
import net.jini.lookup.*;
import net.jini.lease.*;
import net.jini.core.discovery.*;
import net.jini.lookup.entry.*;
import net.jini.core.entry.*;

public class JINIserver implements JINIserverMBean,
                                   MBeanRegistration,
                                   ServiceIDListener
{
    private MBeanServer mbs = null;
    private JoinManager jm = null;
    private ServiceID id = null;
    private String domain = null;
    private ObjectName name = null;
    private String[] groups;
    private Name entry = null;
    private boolean enabled = false;

    public JINIserver()
    {
        groups = new String[ 1 ];
        groups[ 0 ] = "public";
    }
}

```

① **Import Jini packages**

② **Implement necessary interfaces**

```
public String getDomain()
{
    return domain;
}

public String getEntryName()
{
    return entry.name;
}

public void serviceIDNotify( ServiceID id )
{
    this.id = id;
}

public ObjectName preRegister( MBeanServer server,
                               ObjectName name) throws Exception
{
    this.mbs = server;
    if( name == null )
    {
        name =
            new ObjectName( mbs.getDefaultDomain() +
                           ":connectorType=JINI" );
    }
    this.domain = name.getDomain();
    return name;
}

public void postRegister (Boolean registrationDone) {}
public void preDeregister() throws Exception {}
public void postDeregister(){}

public void setGroups( String groups[] )
{
    if( groups != null )
        this.groups = groups;
}

public String[] getGroups()
{
    return groups;
}

public void enableConnections()
{
    createService();
}

public void setEntryName( String name )
{
    Entry old = entry;
    this.entry = new Name( name );
    if( enabled )
```

```

        {
            Entry[] newlabels = { entry };
            Entry[] labels = { old };

            jm.modifyAttributes( labels, newlabels );
        }
    }

    private void createService()
    {
        try
        {
            JINIConnector connector =
                new JINIConnectorImpl( this );

            Entry[] labels = { entry };

            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager( groups, null, null );

            jm = new JoinManager( connector, labels, this,
                mgr,
                new LeaseRenewalManager() );

            enabled = true;
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    /*
     * call back methods
     */
    public Integer getMBeanCount() throws Exception
    {
        return mbs.getMBeanCount();
    }

    public ObjectInstance createMBean( String className,
        ObjectName name ) throws Exception
    {
        return mbs.createMBean( className, name );
    }
}

```

3 Create JoinManager

4 Set up Jini service

4 Set up Jini service

5 Implement remaining methods

- These import statements import packages from the Jini toolkit. These packages contain the classes needed to find lookup services and manage and describe Jini services. All of these packages come with the downloadable Jini toolkit from Sun Microsystems.

- ② The `JINIServer` class implements the following three interfaces:
 - `JINIServerMBean`—This MBean interface declares the exposed attributes and operations for this MBean.
 - `MBeanRegistration`—Recall from chapter 4 that this interface allows the MBean to acquire a reference to the MBean server. For more information, look back at that chapter.
 - `ServiceIDLListener`—This is a Jini interface that allows the Jini lookup service to perform a callback and inform the listener (the interface implementer) of the unique service id generated for a particular Jini service. It declares only one method: `public void serviceIDNotify()`.

With a reference to the MBean server, the MBean can propagate corresponding invocations from the Jini service to the MBean server. (More about this in a moment.)

- ③ The `JINIServer` class uses a Jini utility class, `JoinManager`, which handles the lookup, registration, and management of lookup services for a particular Jini service. You can see how it is created in the `createService()` method. The MBean keeps a reference to this class in order to manage the attributes of the `JINIServerImpl` service.
- ④ The `createService()` method is where all the action takes place in this MBean. This method is invoked when the JMX agent calls the `enableConnections()` method of the MBean. It is responsible for creating the Jini service class (`JINIServerImpl`) and registering it with available Jini lookup services. We will examine the `JINIServerImpl` class in the next code listing, but as you can see, all you have to do is use the constructor that accepts a reference to this MBean. The `JINIServerImpl` class will use that reference to make callbacks to the MBean.

Once the service is created, it must be made available to possible clients. As mentioned earlier, the `JINIServer` MBean uses a Jini utility class called `JoinManager` to handle all the logistics surrounding registration and management of a service on a lookup service. (For more information about the `JoinManager` class, refer to the javadocs bundled in the Jini download.)

- ⑤ All the methods after this point correspond to methods in the MBean server. These are the callback methods the Jini service invokes in order to perform an operation on the `MBeanServer` instance residing in the JMX agent. For the sake of space, only two methods are currently implemented: `getMBeanCount()` and `createMBean()`. You will use the latter method in your tests later.

Writing the Jini service

Now that you've written the `JINIServer` MBean, let's examine the `JINIConnectorImpl` class that the MBean uses to expose its JMX agent to Jini clients. With all Jini services, `JINIConnectorImpl` must implement a remote interface that declares the methods available to a client. The following code is the service interface, `JINIConnector`:

```
package jmxbook.ch9;

import java.rmi.*;
import javax.management.*;

public interface JINIConnector extends Remote
{
    public Integer getMBeanCount() throws JINIConnectorException;

    public ObjectInstance createMBean(String className,
        ObjectName name) throws JINIConnectorException;
}

```

As you can see, it contains only two methods. Recall from the previous code discussion that the Jini connector is left incomplete for the sake of space. For this connector to be complete, all the methods of the MBean server must be made available. This interface declares the methods available to your `JINIConnectorClient` class (discussed in a moment). Remember, however, that all methods declared in this interface will also be made available to the client, allowing the user to invoke the corresponding methods on the remote MBean server.

Notice also that the interface declares the methods as throwing an instance of the `JINIConnectorException` exception class. We'll define this exception class shortly; basically, it extends `java.rmi.RemoteException` and wraps server-side exceptions, enabling service clients to acquire the wrapped exception. You will see it thrown from the Jini service code to follow, and also used in the Jini connector client later.

Listing 9.4 shows the `JINIConnectorImpl` class. It contains the Jini service created by the `JINIServer` MBean.

Listing 9.4 `JINIConnectorImpl.java`

```
package jmxbook.ch9;

import java.rmi.*;
import java.rmi.server.*;
import javax.management.*;

public class JINIConnectorImpl extends
    UnicastRemoteObject implements JINIConnector

```

Extend ①
UnicastRemoteObject class ←

```
{
private JINIServer server = null;

public JINIConnectorImpl( JINIServer server )
    throws RemoteException
{
    this.server = server; ❷ Store reference
to Jini service
}

public Integer getMBeanCount() throws JINIConnectorException
{
    try
    {
        return server.getMBeanCount();
    }
    catch( Exception e )
    {
        throw new JINIConnectorException( "getMBeanCount", e );
    }
}

public ObjectInstance createMBean(String className,
    ObjectName name) throws
    JINIConnectorException
{
    try
    {
        return server.createMBean( className, name );
    }
    catch( Exception e )
    {
        throw new JINIConnectorException( "createMBean", e );
    }
}
}
```

- ❶ In order to be a Jini service, this class must be a remote class. That is, it must be available in a Java RMI environment. Toward this end, it extends the `UnicastRemoteObject` class, which provides the service with the capabilities of a remote object. In addition, it implements its remote interface, `JINIConnector`, which declares the methods that will be available to clients.
- ❷ The key feature of this class is that it contains a reference to the `JINIServer` MBean from the JMX agent. This reference allows the service to make callbacks to the MBean to perform the necessary invocations for a service client.

As you can see, the source is short. However, remember that you implemented only two corresponding methods to the MBean server—to finish this

Jini service, you would need to write the final methods that correspond to the remaining method of the MBean server. From the first two methods already implemented, you should be able to tell that this is not a difficult task.

Before moving on to the Jini connector client, let's look at the `JINIConnectorException` class to clarify its purpose. Listing 9.5 shows the exception class.

Listing 9.5 `JINIConnectorException.java`

```
package jmxbook.ch9;

import java.rmi.*;

public class JINIConnectorException extends RemoteException
{
    private Exception exception = null;

    public JINIConnectorException( String message, Exception ex )
    {
        super( message, ex );
        this.exception = ex;
    }

    public Exception getWrappedException()
    {
        return exception;
    }
}
```

This class extends `RemoteException` so that it can serve as the single exception thrown from the remote interface `JINIConnector`. However, `RemoteException` does not grant access to its wrapped exception, so this subclass does. In the constructor, the class stores the wrapped exception in a class member variable for later retrieval by the `getWrappedException()` method.

This exception class will let the client decipher the exact type of the exception being thrown from a service method (such as `createMBean()`) and throw the appropriate type on the client side.

The last component to cover is the connector client. The following section discusses the `JINIConnectorClient` class, which creates a Jini client.

Writing the Jini connector client

The first thing you will notice when examining the `JINIConnectorClient` class (listing 9.6) is the large section of stubbed-out MBean server methods at the end. The client does in fact implement the `MBeanServer` interface; it does so to provide client-side users with all the methods that would be available on the server

side. We've stubbed out all but two of these methods for the sake of space. The remaining two MBeanServer methods are used to test the connector later. Examine the listing; the discussion follows.

Listing 9.6 JINIConnectorClient.java

```
package jmxbook.ch9;

import javax.management.*;
import java.rmi.*;
import java.util.*;
import net.jini.discovery.*;
import net.jini.core.lookup.*;
import net.jini.lookup.*;
import java.io.*;

public class JINIConnectorClient
    implements DiscoveryListener, MBeanServer
{
    private ServiceTemplate template = null;
    private LookupDiscovery reg = null;
    private JINIConnector server =null;

    public JINIConnectorClient()
    {
        System.setSecurityManager( new RMISecurityManager() );

        Class[] cls = { JINIConnector.class };
        template = new ServiceTemplate( null, cls, null );

        try
        {
            reg = new LookupDiscovery( new String[] { "" } );
            reg.addDiscoveryListener( this );

            while( server == null )
                Thread.sleep( 1000 );
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    public void discovered( DiscoveryEvent event )
    {
        if( server != null )
            return;
        ServiceRegistrar[] lookups = event.getRegistrars();
        try
        {
            ServiceMatches items = lookups[0].lookup( template,
```

1 Begin search
for Jini service

2 Implement
listener callback

```

        Integer.MAX_VALUE );
        server = ( JINIConnector ) items.items[ 0 ].service;
        System.out.println( "service found" );
    }
    catch( Exception e )
    {
        e.printStackTrace();
    }
}

public void discarded( DiscoveryEvent event ){}

public Integer getMBeanCount()
{
    try
    {
        return server.getMBeanCount();
    }
    catch( JINIConnectorException e )
    {
        return null;
    }
}

public ObjectInstance createMBean(String className,
    ObjectName name)
    throws ReflectionException,
    InstanceAlreadyExistsException,
    MBeanRegistrationException, MBeanException,
    NotCompliantMBeanException
{
    try
    {
        return server.createMBean( className, name );
    }
    catch( JINIConnectorException e )
    {
        Exception ex = e.getWrappedException();
        if( ex instanceof ReflectionException )
            throw ( ReflectionException ) ex;
        else if( ex instanceof InstanceAlreadyExistsException )
            throw ( InstanceAlreadyExistsException ) ex;
        else if( ex instanceof MBeanRegistrationException )
            throw ( MBeanRegistrationException ) ex;
        else if( ex instanceof MBeanException )
            throw ( MBeanException ) ex;
        else
            throw ( NotCompliantMBeanException ) ex;
    }
}
}
/*

```

**3 Implement
getMBeanCount()
method**

**4 Implement
createMBean()**

```

UNIMPLEMENTED METHODS BELOW
*/

public Object instantiate(String className)
    throws ReflectionException,
        MBeanException { return null; }
public Object instantiate(String className,
    ObjectName loaderName)
    throws ReflectionException, MBeanException,
        InstanceNotFoundException { return null; }
public Object instantiate(String className, Object params[],
    String signature[])
    throws ReflectionException, MBeanException
        { return null; }
public Object instantiate(String className,
    ObjectName loaderName,
    Object params[], String signature[])
    throws ReflectionException, MBeanException,
        InstanceNotFoundException { return null; }
public ObjectInstance createMBean(String className,
    ObjectName name,
    ObjectName loaderName)
    throws ReflectionException,
        InstanceAlreadyExistsException,
        MBeanRegistrationException, MBeanException,
        NotCompliantMBeanException,
        InstanceNotFoundException { return null; }
public ObjectInstance createMBean(String className,
    ObjectName name,
    Object params[], String signature[])
    throws ReflectionException,
        InstanceAlreadyExistsException,
        MBeanRegistrationException, MBeanException,
        NotCompliantMBeanException { return null; }
public ObjectInstance createMBean(String className,
    ObjectName name,
    ObjectName loaderName, Object params[],
    String signature[])
    throws ReflectionException,
        InstanceAlreadyExistsException,
        MBeanRegistrationException, MBeanException,
        NotCompliantMBeanException,
        InstanceNotFoundException { return null; }
public ObjectInstance registerMBean(Object object,
    ObjectName name)
    throws InstanceAlreadyExistsException,
        MBeanRegistrationException,
        NotCompliantMBeanException { return null; }
public void unregisterMBean(ObjectName name)
    throws InstanceNotFoundException,
        MBeanRegistrationException { return; }

```

5 Implement remaining methods

```
public ObjectInstance getObjectInstance(ObjectName name)
    throws InstanceNotFoundException { return null; }
public Set queryMBeans(ObjectName name, QueryExp query)
    { return null; }
public Set queryNames(ObjectName name, QueryExp query)
    { return null; }
public boolean isRegistered(ObjectName name) { return false; }
public Object getAttribute(ObjectName name, String attribute)
    throws MBeanException,
    AttributeNotFoundException,
    InstanceNotFoundException,
    ReflectionException
    { return null; }
public AttributeList getAttributes(ObjectName name,
    String[] attributes)
    throws InstanceNotFoundException,
    ReflectionException { return null; }
public void setAttribute(ObjectName name, Attribute attribute)
    throws InstanceNotFoundException,
    AttributeNotFoundException,
    InvalidAttributeValueException,
    MBeanException,
    ReflectionException { return; }
public AttributeList setAttributes(ObjectName name,
    AttributeList attributes)
    throws InstanceNotFoundException,
    ReflectionException { return null; }
public Object invoke(ObjectName name, String operationName,
    Object params[], String signature[])
    throws InstanceNotFoundException,
    MBeanException,
    ReflectionException { return null; }
public String getDefaultDomain() { return null; }
public void addNotificationListener(ObjectName name,
    NotificationListener listener,
    NotificationFilter filter,
    Object handback)
    throws InstanceNotFoundException
    { return; }
public void addNotificationListener(ObjectName name,
    ObjectName listener,
    NotificationFilter filter,
    Object handback)
    throws InstanceNotFoundException
    { return; }
public void removeNotificationListener(ObjectName name,
    NotificationListener listener)
    throws InstanceNotFoundException,
    ListenerNotFoundException { return; }
public void removeNotificationListener(ObjectName name,
    ObjectName listener)
```

```

        throws InstanceNotFoundException,
        ListenerNotFoundException { return; }
public MBeanInfo getMBeanInfo(ObjectName name)
        throws InstanceNotFoundException,
        IntrospectionException,
        ReflectionException
        { return null; }
public boolean isInstanceOf(ObjectName name, String className)
        throws InstanceNotFoundException
        { return false; }
public ObjectInputStream deserialize(ObjectName name,
        byte[] data)
        throws InstanceNotFoundException,
        OperationsException { return null; }
public ObjectInputStream deserialize(String className,
        byte[] data)
        throws OperationsException,
        ReflectionException
        { return null; }
public ObjectInputStream deserialize(String className,
        ObjectName loaderName, byte[] data)
        throws InstanceNotFoundException,
        OperationsException, ReflectionException
        { return null; }

public static void main( String args[] ) ⑥ Test in main()
{ method
    try
    {
        JINIConnectorClient client = new JINIConnectorClient();
        System.out.println(client.getMBeanCount() );
        client.createMBean( "jmxbook.ch2.HelloWorld",
            new ObjectName(
                "JMXBookAgent:name=hwtest" ) );
    }
    catch( Exception e )
    {
        e.printStackTrace();
    }
}
}

```

- ① A good place to begin the examination of the code is the class constructor. The constructor is responsible for creating the search parameters that will find the Jini service portion of the Jini connector. It does so by creating an instance of the class `ServiceTemplate`. For now, the only parameter going into the template is the interface type name `jmxbook.ch9.JINIConnector`.

After creating the template, the constructor starts a lookup service-finding process by creating a `LookupDiscoveryManager` instance. This object actively searches for Jini lookup services across the network. The constructor adds the client class as a `DiscoveryListener` and will be notified via the `discovered()` callback when a lookup service is found. When a lookup service is found, the client is notified and can search that lookup service for an instance of the `JINIConnector` service.

- ❷ As mentioned in the previous paragraph, the `discovered()` method is invoked by the `LookupDiscoveryManager` when a lookup service is found. Now that the client has a reference to a lookup service, it uses the `ServiceTemplate` object created in the constructor to search for the `JINIConnector` service. Service matches are returned from the lookup service in a `ServiceMatches` object that contains an array of `ServiceItem` objects. A `ServiceItem` object contains the actual Jini service that matched the search (in this case, an instance of `JINIConnector`). At this point, your client acquires a reference to the Jini service for use. It stores the reference in a class member called `server`.
- ❸ The `getMBeanCount()` method is the first of two methods implemented on the client side to correspond to remote MBean server methods. It simply invokes the identically named method on the `JINIConnector` service and returns the result.
- ❹ The final method implemented in the `JINIConnectorClient` class is `createMBean()` (which corresponds to the remote MBean server `createMBean()` method that is identically declared). This method is singled out here as an example of using the `JINIConnectorException` class.

When this method is invoked, like `getMBeanCount()`, it simply invokes the same method on the `JINIConnector` service. However, unlike the `getMBeanCount()` method, it must be prepared to throw a variety of exceptions back to the user. To accomplish this, you use the `JINIConnectorClient` exception class. When the method catches a `JINIConnectorException` exception, it acquires the wrapped server-side exception, casts it to the appropriate type (the `getWrappedException()` method returns the type `Exception`), and throws it.

- ❺ Recall that we stubbed out the remaining methods declared in the `MBeanServer` interface. They are included below the comment block in order to successfully compile the connector client class.
- ❻ We include a `main()` method to use in a quick test later. The `main()` method creates a `JINIConnectorClient` instance and uses it to connect to a remote MBean server, get the MBean count, and create an instance of the `HelloWorld` MBean.

9.3.3 Outstanding issues

This connector example leaves out some important features. For instance, there are important details to consider when implementing the connector's notification delivery mechanism. You should also make changes so that multiple agents running this type of connector are distinguishable from each other.

Handling notifications

In a normal notification scenario, an agent directly invokes an object that has registered as a notification listener in order to deliver notifications to it. However, in the distributed environment, the notification listener may be in a remote location, and direct invocation may not be possible.

Therefore, the implementation of the `addNotificationListener()` methods of the `JINIConnectorClient` must be different than the usual `server.method()` invocation seen in the two methods you implemented. The add-listener methods must take into account the distributed nature of the connector. The best way to solve this problem is to have the `JINIConnectorClient` instance store the listener reference locally and add itself as the notification listener instead. The client can alternatively be a remote object; in that case the remote MBean can deliver notifications to the client, which can distribute them as needed to local listeners.

Jini connector attributes

Consider the environment that has two active JMX agents, both of which are using a Jini connector to allow remote clients to discover them. These two agents need the ability to ensure that their respective Jini services (created by the Jini connector) are uniquely identifiable. The `JINIserver` MBean allows the agent to set a single attribute for the service. The `JINIserver` MBean should always construct the Jini service using these attributes to be as unique as possible. In addition, the `JINIConnectorClient` needs code to allow a user to enter possible attribute values when the client begins to search for the Jini service. This code will let client users refine their search for a remote JMX agent.

9.3.4 Testing the Jini connector

We have covered all the components of the Jini connector; it is time to use it in an example. To test the connector, you need to modify the `JMXBookAgent` class. For your agent, you will add the `startJINIConnector()` method. For the client, you will use the `JINIConnectorClient` class's `main` method. Typically, a client will construct an instance of the `JINIConnectorClient` class to use. The following section examines the `startJINIConnector()` method.

The startJINIConnector() method

In chapter 3, which introduced the `JMXBookAgent` class, you gave it a `startRMIConector()` method that added the RMI connector MBean to the agent. However, in that chapter, we did not discuss what took place in the code. Listing 9.7 shows the `startJINIConnector()` method.

Listing 9.7 The `startJINIConnector()` method of the `JMXBookAgent` class

```
protected void startJINIConnector()
{
    ObjectName connectorName = null;

    try
    {
        System.setSecurityManager( new RMISecurityManager() );

        JINIserver jini = new JINIserver();
        ObjectName jiniName = null;

        jiniName =
            new ObjectName( "JMXBookAgent:name=JINIConnector" );
        server.registerMBean( jini, jiniName );
        jini.enableConnections();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

The method creates an instance of the connector MBean (the `JINIserver` class) and a new `ObjectName` instance for the MBean, and registers the MBean on the MBean server. Finally, it calls the `enableConnections()` method of the MBean to create and start the Jini service within.

Running the example

Now you have written all the code, you need to test this connector. To compile and run the connector, however, you must download the latest Jini developer kit from <http://www.javasoft.com>. Once you have it, complete the following steps to test the connector:

- 1 Compile the `jmxbook.ch9` package. You need the JMX JARs, the `JMX_remoting.jar` file from the `contrib/jars` folder, and the Jini JAR files in your `CLASSPATH` in order to compile the agent. To compile the connector source files, you need the JMX JARs and the Jini JARs in your `CLASSPATH`.

In addition, you must use the `rmic` compiler to generate stubs for the `JINICConnectorImpl` class.

- 2 Set up the Jini environment. Doing so involves starting an HTTP server, starting the Java activation daemon (`rmid`), and starting a Jini lookup service. Read the Jini documentation to see how to perform these steps.
- 3 Start the JMX agent. You will need to start the agent with a policy file indicated by the `-Djava.security.policy` property in the `java` command.
- 4 Run the client class (`JINICConnectorClient`). Doing so will invoke the class's `main()` method. The method will use the `JINICConnector` to find the JMX agent, get its MBean count, and create a new `HelloWorld` MBean on the agent.

The best way to see the results of this simple test is to open your browser (while the agent is still running) to the location `http://localhost:9092` (assuming you are on the same machine as the agent). You should see all three adapter/connector MBeans (HTML, RMI, and Jini) as well as a new `HelloWorld` MBean.

9.4 JMX and SNMP

A large number of vendors have distributed many devices with SNMP management capabilities. It would be ideal if you could use this existing management base with new applications and systems you are building today. For example, a networking application could acquire knowledge of the health of the hardware it requires before making routing decisions. For such situations, it makes sense to use the SNMP technology already in place. Fortunately, due to the JMX architecture, your JMX agents can expose MBeans using an SNMP adapter. This section will review SNMP and provide information about using JMX with SNMP.

9.4.1 What is SNMP?

SNMP is a monitoring standard that has been in wide use for several years. (SNMP stands for Simple Network Management Protocol, but most developers might argue that it is not that simple.) Two versions of SNMP (v1 and v2) already exist, and a third version is being defined by the Internet Engineering Task Force (IETF).

In an SNMP system, there are *managed devices* such as routers, hubs, computers, operating systems, and even applications. Basically, any device or system that can expose information about itself can become a managed device. SNMP agents exist to convert requests or messages from the SNMP protocol to a device.

A network management system (NMS) sends information to and listens for information from agents.

SNMP provides capabilities for the NMS to communicate with the managed device. The SNMP API has two commands: `read` and `write`. The `read` command is sent to the agent in order to get information about the current state of a managed device. The `write` command is used to set the state on the managed device. Likewise, the managed device can signal the NMS that something interesting has occurred by sending an SNMP *trap*. A trap is the SNMP equivalent of a JMX notification.

Recall from chapter 1 that information about managed devices is stored in a Management Information Base (MIB). The MIB is a hierarchical representation of information about devices. A managed device can be located on a MIB tree using an object name or object identifier—for example, `organization.dod.enterprise.myenterprise.variables.theProduct`. An object identifier is a set of numbers that translates to the textual name. (For more information about MIBs, read the SNMP specifications at <http://www.ietf.org>.)

Due to incompatibilities between machines on the Internet, data must be exchanged using a neutral representation. A standard called Abstract Syntax Notation One (ASN.1) was developed to enable this exchange. Using this notation, people created rules for defining the management information called the *Structure of Management Information (SMI)*. SMI defines simple types such as integers, octet strings, and object ids. It also defines application data types such as network addresses, counters, gauges, time ticks, opaques, integers, and unsigned integers.

9.4.2 Using an SNMP protocol adapter

As with any protocol or transport technology, the flexible JMX architecture enables agents to communicate with SNMP management applications. An SNMP adapter translates data from an MBean to an SNMP MIB and uses the SNMP protocol to transport the information to interested listeners.

Sun Microsystems provides an implementation of an SNMP adapter with a tool suite included with the JDMK. The toolkit provides the capability to develop a JMX agent using an SNMP protocol adapter. Using a tool called `mibgen`, you can generate MBeans that represent SNMP MIBs. The `mibgen` tool creates Java objects for you using your existing MIB definitions. There is even a toolkit to build an NMS using a management API.

The SNMP protocol adapter can work with SNMP v1 and SNMP v2 protocols. As the protocol adapter receives requests from the SNMP system, it maps the

requests to specific MBean operations and executes them. In addition, the protocol adapter can send an SNMP trap to an NMS in place of JMX notifications.

Using the SNMP protocol adapter, an NMS can access the MBeans in the MBean server that represent various MIBs. SNMP does not support the richness of JMX capabilities, but MBeans can be built that support all the capabilities of SNMP. This means your MBeans may have more capabilities than can be accessed using SNMP, but existing NMS systems can take advantage of the exposed capabilities that adhere to the SNMP standard. Java Community Process (JCP) efforts are underway to standardize the mappings between JMX and existing SNMP standards.

For more information about the SNMP protocol adapter and the JDMK, visit the JDMK product page on the Sun web site (<http://java.sun.com/products/jdmk>).

9.5 Connecting by using a TCP adapter

Up to this point, you have seen how you can distribute access to your JMX agents with a number of different technologies. For example, you can use the RMI connector provided by Sun, or you can use Jini to allow your agents to be discovered. In this section, you will write a TCP protocol adapter.

Note that you won't use any object serialization—we don't want you to recreate an RMI-like connector. Instead, the TCP adapter is a socket-based adapter that allows any capable client to connect and send simple commands in order to interact with your agent. This command approach allows non-Java clients to connect to your agent and work with MBeans. For instance, a C++ program could open a socket to your remote agent and acquire attribute values from existing MBeans.

The TCP adapter presented in this section can create MBeans, get and set attributes, and invoke operations. It places a few restrictions on the order and manner that arguments are sent by the client, as you will see when we walk through the code. The adapter is modeled as an MBean that creates a `ServerSocket` object to listen to a particular port. Figure 9.4 illustrates the use of the TCP adapter MBean.

When a client connects, the MBean creates a `TCPAdapter` object in a new `Thread` to handle the client request, and continues listening.

9.5.1 Writing the code

To complete this new protocol adapter, you need to create three classes and one interface:

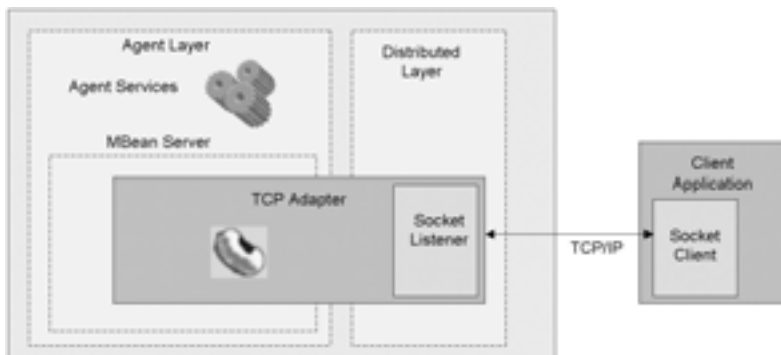


Figure 9.4 The `TCPAdapter` MBean handling incoming requests. Each time a client connects, a new `TCPAdapter` object is created.

- `TCPAdapter`—MBean interface that declares the methods exposed for the adapter
- `TCPAdapter`—Implements the MBean interface and creates the `ServerSocket`
- `TCPAdapter`—Created by the `TCPAdapter` to handle each incoming client
- `TCPAdapter`—Class used to test the TCP adapter

The first step to create the adapter is to write the MBean interface for the `TCPAdapter` MBean. The MBean is a Standard MBean (look back to chapter 4 if necessary), and its interface follows:

```
package jmxbook.ch9;

public interface TCPAdapterMBean
{
    public void setPort( int port );
    public int getPort();
    public boolean start();
    public boolean stop();
}
```

The `TCPAdapterMBean` interface declares two operations and one read/write attribute. The `port` attribute is the port number to which the `ServerSocket` will listen for incoming clients. The `start()` method initiates the `ServerSocket` and tells the MBean to begin listening. Alternatively, the `stop()` method closes the `ServerSocket` and stops the `TCPAdapter` MBean from receiving any new clients. Existing clients will continue to have access to the agent until they close their connection.

Listing 9.8 shows the implementation of the `TCPAdapterMBean` interface in the `TCPAdapter` class.

Listing 9.8 TCPServer.java

```
package jmxbook.ch9;

import javax.management.*;
import java.net.*;

public class TCPServer implements TCPServerMBean,
                                MBeanRegistration, Runnable
{
    private int port = 1555;
    private boolean stopped = false;
    private ServerSocket ss = null;
    private MBeanServer mbs = null;

    public TCPServer()
    {
    }

    public void setPort( int port )
    {
        this.port = port;
    }

    public int getPort()
    {
        return port;
    }

    public boolean start()
    {
        stopped = false;
        Thread t = new Thread( this );
        t.start();
        return true;
    }

    public boolean stop()
    {
        stopped = true;
        return true;
    }

    public void run()
    {
        try
        {
            System.out.println( "Binding to port: " + port );
            ss = new ServerSocket( port );
            while( !stopped )
            {
                Socket client = ss.accept();
                System.out.println( "Client being accepted" );
            }
        }
        catch ( IOException e )
        {
            System.out.println( "Error: " + e );
        }
    }
}
```

←
**Implement
MBeanRegistration and
Runnable interfaces**

←
**Create
ServerSocket
instance**

```

        Thread t = new Thread( new TCPAdapter( client, mbs ) )
        t.start();
    }
    ss.close();
}
catch( Exception e )
{
    e.printStackTrace();
    stopped = true;
}
}

public void postDeregister()
{}
public void postRegister( Boolean done )
{}
public void preDeregister()
{}

public ObjectName preRegister(
    MBeanServer server, ObjectName name )
{
    this.mbs = server;
    return name;
}
}

```

Create TCPAdapter
object to handle
new client

As you can see in the code, upon invocation of the `start()` method, the `TCPServer` MBean runs continuously in a `Thread` until told to stop (via the `stop()` method). Once inside the `run()` method, the MBean opens the `ServerSocket` object to the specified port (the default value is 1555) and begins listening for clients.

When a socket is accepted, the MBean creates a new `TCPAdapter` instance, gives it the new client and a reference to the `MBeanServer`, and runs it in a new `Thread`. Each instance of the `TCPAdapter` class needs a reference to the `MBeanServer` in order to work with MBeans on behalf of its client.

The real work of the TCP adapter is done in the `TCPAdapter` class. It defines the commands clients can send, as well as the order in which it expects them to be sent. Table 9.4 lists the possible commands that can be sent by a TCP client.

Table 9.4 The possible commands used by the TCP client

TCPAdapter variable	Actual value
CREATE_MBEAN	create mbean
ARGS	args

Table 9.4 The possible commands used by the TCP client (continued)

TCPAdapter variable	Actual value
GET_ATTRIBUTE	get attribute
SET_ATTRIBUTE	set attribute
INVOKE	invoke
SHUTDOWN	shutdown

Not every message sent from a client will be one of the commands from table 9.4. Other messages might be a classname or argument value, for instance. Table 9.5 lists the tasks the TCP adapter can perform, along with the messages needed to perform the tasks. The messages and commands are listed in the order they must be received. Messages in bold are expected values from the client (for example, **classname** is an actual classname `String`). Those in *italic* are optional.

Table 9.5 Commands to send to complete a function of the adapter

Adapter function	Command order
Create an MBean	CREATE_MBEAN, classname , objectname , <i>ARGS</i> , <i>arglist</i> , <i>siglist</i>
Get attribute	GET_ATTRIBUTE, attname , objectname
Set attribute	SET_ATTRIBUTE, attname , objectname , <i>ARGS</i> , <i>arglist</i> , <i>siglist</i>
Invoke an operation	INVOKE, operation , objectname , <i>ARGS</i> , <i>arglist</i> , <i>siglist</i>

For each object name sent, the adapter expects the whole `String` value (such as a `String` like `JMXBookAgent:name=myValue`). The *arglist* and *siglist* messages are expected to be comma-separated lists of arguments. The *arglist* parameters must correspond to the types in the *siglist* message. In addition, each object value being passed in the *arglist* must be able to create an `Object` from the `String` value. This is similar to what the HTML adapter expects from clients.

Listing 9.9 shows the `TCPAdapter` class. After examining this class, you will add the TCP adapter to your `JMXBookAgent` class and write a simple test program.

Listing 9.9 `TCPAdapter.java`

```
package jmxbook.ch9;

import java.net.*;
import javax.management.*;
import java.io.*;
```

```
import java.lang.reflect.*;
import java.util.*;

public class TCPAdapter implements Runnable
{
    private MBeanServer server = null;
    private Socket socket = null;
    private BufferedReader in = null;
    private PrintWriter out = null;

    public static String SHUTDOWN           = "shutdown";
    public static String CREATE_MBEAN      = "create mbean";
    public static String GET_ATTRIBUTE     = "get_attribute";
    public static String SET_ATTRIBUTE     = "set_attribute";
    public static String INVOKE           = "invoke";
    public static String ARGS              = "args";

    public TCPAdapter( Socket socket, MBeanServer server )
    {
        this.socket = socket;
        this.server = server;
        try
        {
            this.out = new PrintWriter( socket.getOutputStream() );
            this.in = new BufferedReader(
                new InputStreamReader( socket.getInputStream() ) );
            System.out.println( "TCP Adapter CREATED" );
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    public void run()
    {
        try
        {
            System.out.println( "TCP adapter starting..." );
            String line = in.readLine();

            while( !line.equals( SHUTDOWN ) )
            {
                if( line.equals( CREATE_MBEAN ) )
                {
                    try
                    {
                        createMBean( );
                        out.println( "SUCCESS" );
                        out.flush();
                    }
                    catch( Exception e )
                    {

```

❶ Read until shutdown

```
        e.printStackTrace();
        out.println( "ERROR " + e.getMessage() );
        out.flush();
    }
}
else if( line.equals( GET_ATTRIBUTE ) )
{
    try
    {
        out.println( getAttribute( ) );
        out.flush();
    }
    catch( Exception e )
    {
        e.printStackTrace();
        out.println( "ERROR " + e.getMessage() );
        out.flush();
    }
}
else if( line.equals( SET_ATTRIBUTE ) )
{
    try
    {
        setAttribute( );
        out.println( "SUCCESS" );
        out.flush();
    }
    catch( Exception e )
    {
        e.printStackTrace();
        out.println( "ERROR " + e.getMessage() );
        out.flush();
    }
}
else if( line.equals( INVOKE ) )
{
    try
    {
        out.println( invoke() );
        out.flush();
    }
    catch( Exception e )
    {
        e.printStackTrace();
        out.println( "ERROR " + e.getMessage() );
        out.flush();
    }
}

line = in.readLine();
}
```

```

        in.close();
        out.close();
        socket.close();
    }
    catch( Exception e )
    {
        e.printStackTrace();
    }
}

private void createMBean() throws Exception
{
    String classname = null;
    String objectName = null;
    String line = in.readLine();
    String arglist = null;
    String siglist = null;

    classname = line;
    objectName = in.readLine();

    line = in.readLine();
    if( line.equals( ARGS ) )
    {
        arglist = in.readLine();
        siglist = in.readLine();
    }

    String[] sig = createSignature( siglist );
    Object[] args = createObjectList( arglist, sig );

    System.out.println( "NOW CREATING MBEAN" );

    server.createMBean( classname, new ObjectName( objectName ),
                       args, sig );
}

private String getAttribute() throws Exception
{
    String attname = null;
    String objectName = null;
    String line = in.readLine();
    attname = line;

    objectName = in.readLine();

    System.out.println( "GETTING ATTRIBUTE " + attname
                       + " FROM " + objectName );

    Object obj = server.getAttribute( new ObjectName( objectName ),
                                     attname );

    return obj.toString();
}

private void setAttribute() throws Exception

```

**Implement
createMBean()**

2

```

{
    String attName = null;
    String objectName = null;
    String line = in.readLine();
    String arglist = null;
    String siglist = null;

    attName = line;
    objectName = in.readLine();

    line = in.readLine();
    arglist = in.readLine();
    siglist = in.readLine();

    String[] sig = createSignature( siglist );
    Object[] args = createObjectList( arglist, sig );

    System.out.println( "SETTING ATTRIBUTE " + attName
                       + " FROM " + objectName );
    server.setAttribute( new ObjectName( objectName ),
                       new Attribute( attName, args[0] ) );
}

private String invoke() throws Exception
{
    String operation = null;
    String objectName = null;
    String line = in.readLine();
    String arglist = null;
    String siglist = null;

    operation = line;
    objectName = in.readLine();

    line = in.readLine();
    if( line.equals( ARGS ) )
    {
        arglist = in.readLine();
        siglist = in.readLine();
    }

    String[] sig = createSignature( siglist );
    Object[] args = createObjectList( arglist, sig );

    System.out.println( "INVOKING OPERATION " + operation
                       + " FROM " + objectName );
    Object result = server.invoke( new ObjectName( objectName ),
                                operation, args, sig );
    return result.toString();
}

private String[] createSignature( String siglist )
{
    if( siglist == null )
        return null;
}

```

**Implement
invoke()**

3

```

StringTokenizer toks = new StringTokenizer( siglist, "," );
String[] result = new String[ toks.countTokens() ];

int i = 0;
while( toks.hasMoreTokens() )
{
    result[ i++ ] = toks.nextToken();
}

return result;
}

private Object[] createObjectList( String objects,
                                   String[] sig ) throws Exception
{
    if( objects == null )
        return null;

    Object[] results = new Object[ sig.length ];
    StringTokenizer toks = new StringTokenizer( objects, "," );

    int i = 0;
    while( toks.hasMoreTokens() )
    {
        String object = toks.nextToken();
        Class conSig[] = { Class.forName( sig[i] ) };
        Object[] conParams = { object };

        Class c = Class.forName( sig[i] );
        Constructor con = c.getConstructor( conSig );
        results[ i ] = con.newInstance( conParams );
        i++;
    }

    return results;
}
}

```

- ❶ Once the TCPAdapter object has been created by the TCPServer MBean, it is continuously in its run() method until it reads the SHUTDOWN message from the client. Upon receiving that message, the adapter object closes the socket and stops communication.

In the run() method, the adapter reads messages from the client until it finds one of its four available tasks (CREATE_MBEAN, GET_ATTRIBUTE, SET_ATTRIBUTE, OR INVOKE). When it reads a valid command, it invokes the appropriate private method to complete the task. Because the output and input streams are class variables, they can be used in every method.

- ❷ The createMBean() method allows clients to create new MBeans in the agent. If any exceptions occur during this process, the method fails and returns an error

to the client. When creating an MBean, clients should expect either the `SUCCESS` message or an error returned.

To complete this task, the client must send the classname of the MBean, a `String` value for an object name, and the arguments and signature if a constructor with arguments is to be used. If arguments are sent, the method breaks them into the needed `Object` and `String` arrays for the MBean server's `createMBean()` method. After acquiring all the necessary information from the client, the `createMBean()` method invokes the `createMBean()` method on the MBean server to complete the task. If no exception is thrown, the task completes.

- 3 The `invoke()` method works similarly to the `createMBean()` method. For this task, the adapter must gather the operation name, object name, and possible arguments from the client in order to invoke an MBean operation. After doing so, the `invoke()` method calls the `invoke()` method of the MBean server and prepares the return value to be sent back to the client. The return value is put into `String` form via the `toString()` method to be sent over the socket. No object serialization is used.

Adding the adapter to the JMXBookAgent class

Before you write the test program, let's add the code to the `JMXBookAgent` class that will create TCP adapter when the agent is started. Listing 9.10 shows the new `startTCPAdapter()` method for the agent class.

Listing 9.10 The `startTCPAdapter()` method

```
protected void startTCPAdapter()
{
    TCPAdapter tcp = new TCPAdapter();
    ObjectName adapterName = null;

    try
    {
        adapterName = new ObjectName(
            "JMXBookAgent:name=TCPAdapter");
        server.registerMBean( tcp, adapterName);
        tcp.start();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

Be sure to invoke the new method from the agent's constructor. In addition, you will need to import the `jmxbook.ch9` package. It is like all the other connectivity methods of the agent—it creates the MBean, registers it on the MBean server, and invokes its `start()` method. Use the HTML adapter (via a web browser) if you need to change the port value of the adapter.

9.5.2 Testing the TCP adapter

With everything else completed, it is time to write a simple test program for the TCP adapter (see listing 9.11). The test program is defined by the class `TCPTester`, and it performs all four tasks available to the adapter.

Listing 9.11 `TCPTester.java`

```
package jmxbook.ch9;

import java.net.*;
import javax.management.*;
import java.io.*;

public class TCPTester
{
    public TCPTester( String port ) throws Exception
    {
        Socket s = new Socket( "localhost", Integer.parseInt( port ) );
        PrintWriter print = new PrintWriter( s.getOutputStream() );

        //create a Hello World MBean

        print.println( TCPAdapter.CREATE_MBEAN );
        print.flush();
        print.println( "jmxbook.ch2.HelloWorld" );
        print.flush();
        print.println( "JMXBookAgent:name=TCPCreatedHW" );
        print.flush();
        print.println( TCPAdapter.ARGS );
        print.flush();
        print.println( "This is my greeting" );
        print.flush();
        print.println( "java.lang.String" );
        print.flush();

        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                s.getInputStream() ) );

        System.out.println( in.readLine() );
        Thread.sleep(10000);

        //reset the greeting
    }
}
```

```
print.println( TCPAdapter.SET_ATTRIBUTE );
print.flush();
print.println( "Greeting" );
print.flush();
print.println( "JMXBookAgent:name=TCPCreatedHW" );
print.flush();
print.println( TCPAdapter.ARGS );
print.flush();
print.println( "This is my greeting after being changed" );
print.flush();
print.println( "java.lang.String" );
print.flush();

Thread.sleep(10000);

//get the greeting
print.println( TCPAdapter.GET_ATTRIBUTE );
print.flush();
print.println( "Greeting" );
print.flush();
print.println( "JMXBookAgent:name=TCPCreatedHW" );
print.flush();

System.out.println( in.readLine() );

//invoke printGreeting
print.println( TCPAdapter.INVOKE );
print.flush();
print.println( "printGreeting" );
print.flush();
print.println( "JMXBookAgent:name=TCPCreatedHW" );
print.flush();
print.println( TCPAdapter.SHUTDOWN );
print.flush();

System.out.println( in.readLine() );

}

public static void main(String args[]) throws Exception
{
    TCPTester t = new TCPTester( args[0] );
}
}
```

The output on the agent and from the test program should tell you everything you need to know. In addition, you can check out the HTML adapter view to see the results of running the test program.

9.6 Summary

Previous chapters made it clear how JMX uses protocol adapters and connectors to provide connectivity to JMX agents through all manner of technologies and protocols. This chapter covered the RMI connector more thoroughly, discussed SNMP, and showed you how to write a Jini connector and a TCP adapter.

The RMI connector is contributed to developers in Sun Microsystems' JMX Reference Implementation. It lets you connect to remote JMX agents using Java RMI. In addition, it provides excellent handling of remote notification delivery and heartbeat functionality.

The Jini connector you wrote took the RMI connector one step further by allowing you to connect to a JMX agent using the Jini network technology. The connector still operates over Java RMI, but clients do not have to know the address of a potential JMX agent. Using the Jini discovery mechanism, you were able to provide an agent discovery capability to remote clients.

Finally, you created a TCP adapter to provide access to JMX agents from non-Java clients. Even though the TCP adapter is limited by its inability to translate complex objects to simple commands, it does provide the core functionality of a JMX agent to TCP clients. In fact, the TCP adapter is much like the HTML adapter.

Chapter 9 not only provided you with the examples for agent connectivity, it also showed you some guidelines for writing your own custom classes in order to provide connectivity for other technologies or protocols that you have in your environment.

Agent services provide important functionality to every JMX agent. Chapter 10 covers the first of four agent services that are present in every JMX-compliant agent: the M-let service, which is used to load MBeans from remote locations outside an agent's codebase.