

RMI - Remote Method Invocation

16th February 2005

Abstract

The Java Remote Method Invocation (RMI) enables an object running in Java Virtual Machine (JVM) to call a method on another object running in a separate JVM. This worksheet provides an introduction to RMI programming its structure is based on a cut down version of the official Java RMI tutorial [2]. In this worksheet you will create a simple RMI client-server application.

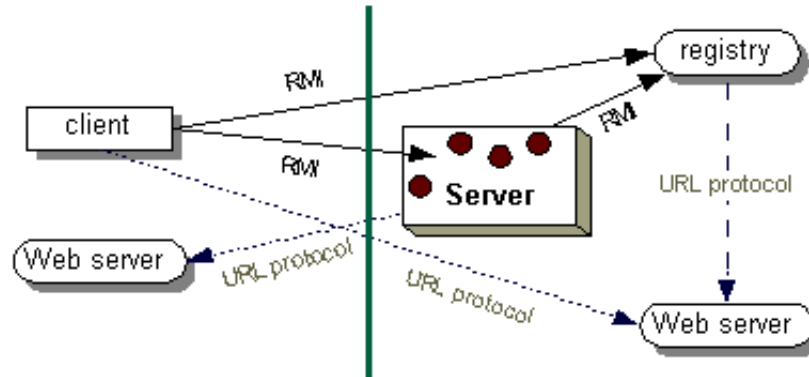
1 Introduction [2]

RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to:

- Locate remote objects: Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, the `rmiregistry`, or the application can pass and return remote object references as part of its normal operation.
- Communicate with remote objects: Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.
- Load class bytecodes for objects that are passed around: Because RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object's code, as well as for transmitting its data.

The following illustration depicts an RMI distributed application that uses the registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing Web server to load class bytecodes, from server to client and from client to server, for objects when needed.



1.1 Advantages of Dynamic Code Loading

One of the central and unique features of RMI is its ability to download the bytecodes (or simply code) of an object's class if the class is not defined in the receiver's virtual machine. The types and the behavior of an object, previously available only in a single virtual machine, can be transmitted to another, possibly remote, virtual machine. RMI passes objects by their true type, so the behavior of those objects is not changed when they are sent to another virtual machine. This allows new types to be introduced into a remote virtual machine, thus extending the behavior of an application dynamically. The compute engine example in this chapter uses RMI's capability to introduce new behavior to a distributed program.

1.2 Remote Interfaces, Objects, and Methods

Like any other application, a distributed application built using Java RMI is made up of interfaces and classes. The interfaces define methods, and the classes implement the methods defined in the interfaces and, perhaps, define additional methods as well. In a distributed application some of the implementations are assumed to reside in different virtual machines. Objects that have methods that can be called across virtual machines are remote objects.

An object becomes remote by implementing a remote interface, which has the following characteristics.

- A remote interface extends the interface `java.rmi.Remote`.

- Each method of the interface declares `java.rmi.RemoteException` in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a nonremote object when the object is passed from one virtual machine to another. Rather than making a copy of the implementation object in the receiving virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the caller, the remote reference. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This allows a stub to be cast to any of the interfaces that the remote object implements. However, this also means that only those methods defined in a remote interface are available to be called in the receiving virtual machine.

1.3 Creating Distributed Applications Using RMI

When you use RMI to develop a distributed application, you follow these general steps.

1. Design and implement the components of your distributed application.
2. Compile sources and generate stubs.
3. Make classes network accessible.
4. Start the application.

1.4 Design and Implement the Application Components

First, decide on your application architecture and determine which components are local objects and which ones should be remotely accessible. This step includes:

- Defining the remote interfaces: A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. Part of the design of such interfaces is the determination of any local objects that will be used as parameters and return values for these methods; if any of these interfaces or classes do not yet exist, you need to define them as well.
- Implementing the remote objects: Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces (either local or remote) and other methods (which are available only locally). If any local classes are to be used as parameters or return values to any of these methods, they must be implemented as well.

- Implementing the clients: Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

1.5 Compile Sources and Generate Stubs

This is a two-step process. In the first step you use the `javac` compiler to compile the source files, which contain the implementation of the remote interfaces and implementations, the server classes, and the client classes. In the second step you use the `rmic` compiler to create stubs for the remote objects. RMI uses a remote object's stub class as a proxy in clients so that clients can communicate with a particular remote object.

1.6 Make Classes Network Accessible

In this step you make everything—the class files associated with the remote interfaces, stubs, and other classes that need to be downloaded to clients—accessible via a Web server.

1.7 Start the Application

Starting the application includes running the RMI remote object registry, the server, and the client.

2 Creating the Server

An RMI server essentially consists of two components:

- The Remote Interface
- The `UnicastRemoteObject` Class

The remote interface defines the services that the RMI server will provide. The `UnicastRemoteObject` class provides an implementation of the services defined in the remote interface.

2.1 The Remote Interface

The purpose of the remote interface must is to define the services that the RMI server will provide. Think of this interface as the contract between the client and the server. The client does not need to know how the method is to be implemented instead they only need know the method signature. Remote interfaces must adhere to a few basic rules:

- The interface must extend from `java.rmi.Remote` interface. This basically means that any class that implements this interface will become a remote object.

- Any methods defined in the interface must be declared as throwing `java.rmi.RemoteException`.
- Any parameters passed or returned must be `Serializable`. This means that if you create a custom class and which to exchange instances of it as parameters using RMI then the class must implement the `java.io.Serializable` interface. By implementing this interface you are marking a class of being capable of conversion into a stream of bytes that can be reconstructed to form an exact copy of the 'serialized' object. Are there are types of objects that could not be serialized?

Below is an example interface that defines one method called `getCurrentTime()` that returns a `String` and takes no parameters.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyRemoteInterface extends Remote {

    public String getCurrentTime() throws RemoteException;

}
```

Enter this code into a text editor and save with the name `MyRemoteInterface.java`. This service definition is implemented in the following section.

2.2 The UnicastRemoteObject

The remote interface we created in the previous section provides a definition of the service that the server will provide the client. In order to make this usable we must implement this service definition (remote interface). We do this by creating an implementation class that adheres to the following rules:

- It must declare the remote (service definition) interface as being implemented.
- It must provide an implementation for each of the remote methods in the remote interface.
- It must define the constructor for the remote object.

The RMI server must all do the following:

- Create and install a security manager.
- Create [1..N] numbers of the remote object.
- Register [1..N] of the remote objects on the RMI registry (Note: It is possible to use other naming services but for this tutorial we will only be looking at the RMI registry).

Have a look at the following code for our server implementation class:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class MyRMIServer extends UnicastRemoteObject implements
    MyRemoteInterface {

    public MyRMIServer() throws RemoteException {
        super();
    }

    public String getCurrentTime() throws RemoteException {
        Date now = new Date();
        return now.toString();
    }

    public void myMethod() {
        System.out.println("Local Method");
    }

    public static void main(String []args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "//localhost/MyRMIObject";
        try {
            MyRemoteInterface obj = new MyRMIServer();
            Naming.rebind(name, obj);
        } catch (Exception e) {
            System.err.println("Exception e = "+e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Enter this code into a text editor and save with the name `MyRMIServer.java`. The following subsections form an edited version of the RMI tutorial.

2.2.1 Declare the Remote Interfaces Being Implemented

The implementation class for the compute engine is declared as:

```
public class MyRMIServer extends UnicastRemoteObject implements MyRemoteInterface
```

`UnicastRemoteObject` is a convenience class, defined in the RMI public API, that can be used as a superclass for remote object implementations. The superclass `UnicastRemoteObject` supplies implementations for a number of `java.lang.Object` methods (`equals`, `hashCode`, `toString`) so that they are defined

appropriately for remote objects. `UnicastRemoteObject` also includes constructors and static methods used to export a remote object, that is, make the remote object available to receive incoming calls from clients.

The `MyRMIServer` example defines a remote object class that implements only a single remote interface and no other interfaces. The `MyRMIServer` class also contains a method that can be called only locally.

2.2.2 Define the Constructor

The `MyRMIServer` class has a single constructor that takes no arguments. The code for the constructor is:

```
public MyRMIServer() throws RemoteException {
    super();
}
```

This constructor simply calls the superclass constructor, which is the no-argument constructor of the `UnicastRemoteObject` class. Although the superclass constructor gets called even if omitted from the constructor, we include it for clarity.

During construction, a `UnicastRemoteObject` is exported, meaning that it is available to accept incoming requests by listening for incoming calls from clients on an anonymous port.

The no-argument constructor for the superclass, `UnicastRemoteObject`, declares the exception `RemoteException` in its throws clause, so the `MyRMIServer` constructor must also declare that it can throw `RemoteException`. A `RemoteException` can occur during construction if the attempt to export the object fails—due to, for example, communication resources being unavailable or the appropriate stub class not being found.

2.2.3 Provide Implementations for Each Remote Method

The `MyRemoteInterface` interface contains a single remote method, `getCurrentTime()`, which is implemented as follows:

```
public String getCurrentTime() throws RemoteException {
    Date now = new Date();
    return now.toString();
}
```

This method implements the protocol between the `MyRMIServer` and its clients.

2.2.4 Passing Objects in RMI

Arguments to or return values from remote methods can be of almost any type, including local objects, remote objects, and primitive types. More precisely, any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a primitive data type, a remote

object, or a serializable object, which means that it implements the interface `java.io.Serializable`.

A few object types do not meet any of these criteria and thus cannot be passed to or returned from a remote method. Most of these objects, such as a file descriptor, encapsulate information that makes sense only within a single address space. Many of the core classes, including those in the packages `java.lang` and `java.util`, implement the `Serializable` interface.

The rules governing how arguments and return values are passed are as follows.

- Remote objects are essentially passed by reference. A remote object reference is a stub, which is a client-side proxy that implements the complete set of remote interfaces that the remote object implements.
- Local objects are passed by copy, using object serialization. By default all fields are copied, except those that are marked `static` or `transient`. Default serialization behavior can be overridden on a class-by-class basis.

Passing an object by reference (as is done with remote objects) means that any changes made to the state of the object by remote method calls are reflected in the original remote object. When passing a remote object, only those interfaces that are remote interfaces are available to the receiver; any methods defined in the implementation class or defined in nonremote interfaces implemented by the class are not available to that receiver.

In remote method calls objects—parameters, return values, and exceptions—that are not remote objects are passed by value. This means that a copy of the object is created in the receiving virtual machine. Any changes to this object’s state at the receiver are reflected only in the receiver’s copy, not in the original instance.

2.2.5 Implement the Server’s main Method

The most involved method of the `MyRMIServer` implementation is the main method. The main method is used to start the `MyRMIServer` and therefore needs to do the necessary initialization and housekeeping to prepare the server for accepting calls from clients. This method is not a remote method, which means that it cannot be called from a different virtual machine. Since the main method is declared `static`, the method is not associated with an object at all but rather with the class `MyRMIServer`.

2.2.6 Create and Install a Security Manager

The first thing that the main method does is to create and to install a security manager, which protects access to system resources from untrusted downloaded code running within the virtual machine. The security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations.

All programs using RMI must install a security manager, or RMI will not download classes (other than from the local class path) for objects received as parameters, return values, or exceptions in remote method calls. This restriction ensures that the operations performed by downloaded code go through a set of security checks.

The `MyRMIServer` uses a security manager supplied as part of the RMI system, the `RMISecurityManager`. This security manager enforces a similar security policy as the typical security manager for applets; that is to say, it is very conservative as to what access it allows. An RMI application could define and use another `SecurityManager` class that gave more liberal access to system resources or use a policy file that grants more permissions.

Here's the code that creates and installs the security manager:

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
}
```

2.2.7 Make the Remote Object Available to Clients

Next, the main method creates an instance of the `MyRMIServer`. This is done with the statement:

```
MyRemoteInterface engine = new MyRMIServer();
```

As mentioned, this constructor calls the `UnicastRemoteObject` superclass constructor, which in turn exports the newly created object to the RMI runtime. Once the export step is complete, the `MyRMIServer` remote object is ready to accept incoming calls from clients on an anonymous port, one chosen by RMI or the underlying operating system. Note that the type of the variable `engine` is `MyRemoteInterface`, not `MyRMIServer`. This declaration emphasizes that the interface available to clients is the `MyRemoteInterface` interface and its methods, not the `MyRMIServer` class and its methods.

Before a caller can invoke a method on a remote object, that caller must first obtain a reference to the remote object. This can be done in the same way that any other object reference is obtained in a program, such as getting it as part of the return value of a method or as part of a data structure that contains such a reference.

The system provides a particular remote object, the RMI registry, for finding references to remote objects. The RMI registry is a simple remote object name service that allows remote clients to get a reference to a remote object by name. The registry is typically used only to locate the first remote object an RMI client needs to use. That first remote object then provides support for finding other objects.

The `java.rmi.Naming` interface is used as a front-end API for binding, or registering, and looking up remote objects in the registry. Once a remote object is registered with the RMI registry on the local host, callers on any host can

look up the remote object by name, obtain its reference, and then invoke remote methods on the object. The registry may be shared by all servers running on a host, or an individual server process may create and use its own registry, if desired.

The `MyRMIServer` class creates a name for the object with the statement:

```
String name = "//localhost/MyRMIObject";
```

This name includes the host name, host, on which the registry (and remote object) is being run and a name, `MyRemoteInterface`, that identifies the remote object in the registry. The code then needs to add the name to the RMI registry running on the server. This is done later (within the try block) with the statement:

```
Naming.rebind(name, obj);
```

Calling the `rebind` method makes a remote call to the RMI registry on the local host. This call can result in a `RemoteException` being generated, so the exception needs to be handled. The `MyRMIServer` class handles the exception within the try/catch block. If the exception is not handled in this way, `RemoteException` would have to be added to the throws clause (currently nonexistent) of the main method.

Note the following about the arguments to the call to `Naming.rebind`.

- The first parameter is a URL-formatted `java.lang.String` representing the location and the name of the remote object. You will need to change the value of `host` to be the name, or IP address, of your server machine. If the host is omitted from the URL, the host defaults to the local host. Also, you don't need to specify a protocol in the URL. Optionally a port number may be supplied in the URL; for example, the name `//host:1234/objectname` is legal. If the port is omitted, it defaults to 1099. You must specify the port number only if a server creates a registry on a port other than the default 1099. The default port is useful in that it provides a well-known place to look for the remote objects that offer services on a particular host.
- The RMI runtime substitutes a reference to the stub for the remote object reference specified by the argument. Remote implementation objects, such as instances of `MyRMIServer`, never leave the VM where they are created, so when a client performs a lookup in a server's remote object registry, a reference to the stub is returned. As discussed earlier, remote objects in such cases are passed by reference rather than by value.
- Note that for security reasons, an application can bind, unbind, or rebind remote object references only with a registry running on the same host. This restriction prevents a remote client from removing or overwriting any of the entries in a server's registry. A lookup, however, can be requested from any host, local or remote.

Once the server has registered with the local RMI registry it's ready to start handling calls and then the main method exits. It is not necessary to have a thread wait to keep the server alive. As long as there is a reference to the `MyRMIServer` object in another virtual machine, local or remote, the `MyRMIServer` object will not be shut down, or garbage collected. Because the program binds a reference in the registry, it is reachable from a remote client, the registry itself! The RMI system takes care of keeping the `MyRMIServer` process up. The `MyRMIServer` is available to accept calls and won't be reclaimed until its binding is removed from the registry, and no remote clients hold a remote reference to the `MyRMIServer` object.

The final piece of code in the `MyRMIServer` main method deals with handling any exception that might arise. The only exception that could be thrown in the code is a `RemoteException`, thrown either by the constructor of the `MyRMIServer` class or by the call to the RMI registry to bind the object to the name. In either case the program can't do much more than exit after printing an error message. In some distributed applications it is possible to recover from the failure to make a remote call. For example, the application could choose another server and continue operation.

2.3 Build the Server

Now you compile the `MyRMIServer.java` source file, generate a stub for the `MyRMIServer` class, and make that stub network accessible. To create stub (and optionally skeleton files), run the `rmic` compiler on the fully qualified class names of the remote object implementations that must be found in the class path. The `rmic` command takes one or more class names as input and produces as output class files of the form `ClassName_Stub.class` and `ClassName_Skel.class`. (Note: A skeleton file will not be generated if you run `rmic` with the `-v1.2` option.) This option should be used only if all of your clients will be running JDK 1.2 or compatible versions.

We compile this class using the following command:

```
javac MyRMIServer.java  
  
rmic MyRMIServer
```

Note: If you get any class not found exceptions make sure that both the interface and class are in the current directory. If that still doesn't work try:
`javac -classpath . MyRMIServer.java`

2.3.1 The Stub

Below is a decompiled Stub so you can see what is going behind the scenes:

```
// Decompiled with Jad  
  
import java.lang.reflect.Method;  
import java.rmi.*;
```

```

import java.rmi.server.*;

public final class MyRMIServer_Stub extends RemoteStub
    implements MyRemoteInterface, Remote {

    public MyRMIServer_Stub(RemoteRef remoteref) {
        super(remoteref);
    }

    static Class _mthclass$(String s){
        try {
            return Class.forName(s);
        } catch(ClassNotFoundException classnotfoundexception) {
            throw new NoClassDefFoundError(classnotfoundexception.getMessage());
        }
    }

    public String getCurrentTime() throws RemoteException {
        try {
            Object obj = super.ref.invoke(this, $method_getCurrentTime_0,
                null, 0x1be1d648023f2512L);
            return (String)obj;
        } catch(RuntimeException runtimeexception) {
            throw runtimeexception;
        } catch(RemoteException remoteexception) {
            throw remoteexception;
        } catch(Exception exception) {
            throw new UnexpectedException("undeclared checked exception", exception);
        }
    }

    private static final long serialVersionUID = 2L;
    private static Method $method_getCurrentTime_0;

    static {
        try {
            $method_getCurrentTime_0 = (MyRemoteInterface.class).getMethod("getCurrentTime",
new Class[0]);
        } catch(NoSuchMethodException _ex) {
            throw new NoSuchMethodError("stub class initialization failed");
        }
    }
}

```

3 Creating the Client

The client code is shown below, enter it into a text editor and save as MyRMI-Client:

```

import java.rmi.*;

public class MyRMIClient {

```

```

public MyRMIClient() {
}

public static void main(String []args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }
    try {
        String name = "//localhost/MyRMIObject";
        MyRemoteInterface obj = (MyRemoteInterface) Naming.lookup(name);
        System.out.println("Time is "+obj.getCurrentTime());
    } catch (Exception e) {
        System.err.println("Exception e = "+e.getMessage());
        e.printStackTrace();
    }
}
}
}

```

3.1 The code in more detail

Like the `MyRMIServer` server, the client begins by installing a security manager. This is necessary because RMI could be downloading code to the client. In this example the `MyRMIServer` stub is downloaded to the client. Any time code is downloaded by RMI, a security manager must be present. As with the server, the client uses the security manager provided by the RMI system for this purpose.

After installing a security manager, the client constructs a name used to look up a `MyRemoteInterface` remote object. The client uses the `Naming.lookup` method to look up the remote object by name in the remote host's registry. When doing the name lookup, the code creates a URL that specifies the host where the compute server is running. The name passed in the `Naming.lookup` call has the same URL syntax as the name passed in the `Naming.rebind` call.

Next, the client calls the `getCurrentTime()` method on the `MyRemoteInterface` object and prints its value to the screen.

3.2 Build the Client

To build the client run the following command:

```
javac MyRMIClient.java
```

4 Running the Examples

4.1 Policy Files

Before you can run the examples you need to create a policy file such as the one shown below:

```
grant {
```

```
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```

Save this file as `java.policy` and put it in the same directory as your source and class files.

4.2 RMI Registry

First you need to start the RMI registry. To do this under use:

```
rmiregistry
```

Then you need to start the server:

```
java -Djava.security.policy=java.policy -Djava.rmi.server.hostname=localhost MyRMIServer
```

Then finally run the client:

```
java -Djava.security.policy=java.policy -Djava.rmi.server.hostname=localhost MyRMIClient
```

4.3 Output

The output should look something like:

```
java -Djava.security.policy=java.policy -Djava.rmi.server.hostname=localhost MyRMIClient
Time is Sat Feb 12 19:02:43 GMT 2005
```

5 Exercises

1. Create a class that will act as a dataholder that will be passed between an RMI client and server. Hint: What interface will this object need to implement?
2. Create a new remote interface that passes the class created in ex1 between the client and server.
3. Create the client and server.

5.1 Optional

Prior to the release of the Java™ 2 SDK, an instance of a `UnicastRemoteObject` could be accessed from a program that (1) created an instance of the remote object, and (2) ran all the time. Now with the introduction of the class `java.rmi.activation.Activatable` and the RMI daemon, `rmid`, programs can be written to register information about remote object implementations that should be created and execute "on demand", rather than running all the time.

The RMI daemon, `rmid`, provides a Java virtual machine* (JVM) from which other JVM instances may be spawned.

To complete the following you will need to reference: <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/activation.html>

- Read up on remote object activation, then create an activatable object, a activatable `UnicastRemoteObject` class and work through the previous exercises.

6 Questions

- How does socket programming compare to RMI programming? What are the benefits and weaknesses of each approach?
- How does RMI compare to RPC?

References

- [1] Default Policy Implementation and Policy File Syntax - <http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html>
- [2] RMI tutorial - <http://java.sun.com/docs/books/tutorial/rmi/index.html>
- [3] RMI homepage - <http://java.sun.com/products/jdk/rmi/>
- [4] Object Activation - <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/activation.html>