Java Remote Method Invocation

Based on article by Gopalan Suresh Raj, Mastering Enterprise JavaBeans™ Second Edition by Ed Roman, Scott Ambler and Tyler Jewell (Wiley Computer Publishing), Thinkling in Java" by Bruce Eckel (second edition) and Sun RMI tutorialRemote

Method Invocation (RMI) is the object equivalent of Remote Procedure Calls (RPC). While RPC allows you to call procedures over a network, RMI invokes an object's methods over a network.

RMI was the original way to perform remote method invocations in Java and uses the *java.rmi* package. In Java enterprise RMI-IIOP is used more widely.

RMI has some interesting features not available in RMI-IIOP, such as:

- **distributed garbage collection,**
- **object activation**
  Prior to Java 2, 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. 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.

downloadable class files (One of the most significant capabilities of the Java is the ability to dynamically download Java software from any URL to a JVM running in a separate process, usually on a different physical system. The result is that a remote system can run a program, for example an applet, which has never been installed on its disk. For example, a JVM running from within a web browser can download the bytecodes for subclasses of `java.applet.Applet` and any other classes needed by that applet. Thus, the system on which the browser is running has most likely never run this applet before, nor installed it on its disk. Once all the necessary classes have been downloaded from the server, the browser can start the execution of the applet program using the local resources of the system on which the client browser is running.

Java RMI takes advantage of this capability to download and execute classes and on systems where those classes have never been installed on disk. Using the **RMI API any JVM, not only those in browsers, can download any Java class file including specialized RMI stub classes, which enable the execution of method calls on a remote server using the server system's resources.)**

In the RMI model, the server defines objects that the client can use remotely**. The clients can now invoke methods of this remote object as if it were a local object running in the same virtual machine as the client. RMI hides the underlying mechanism of transporting method arguments and return values across the network**. In Java-RMI,

an argument or return value can be of any **primitive** Java type or any other **Serializable** Java object.

A ***remote procedure call (RPC)*** is a procedural invocation from a process on one machine to a process on another machine. RPCs enable traditional procedures to reside on multiple machines, yet still remain in communication. They are a simple way to perform cross-process or cross-machine networking.

A ***remote method invocation*** in Java takes the RPC concept one step further and allows for distributed *object* communications. RMI and RMI-IIOP allows you to invoke methods on objects remotely, not merely procedures. You can build your networked code as full objects. This yields the benefits of an object-oriented programming, such as inheritance, encapsulation, and polymorphism.

What is Java-RMI and how does it compare with other Middleware Specifications?

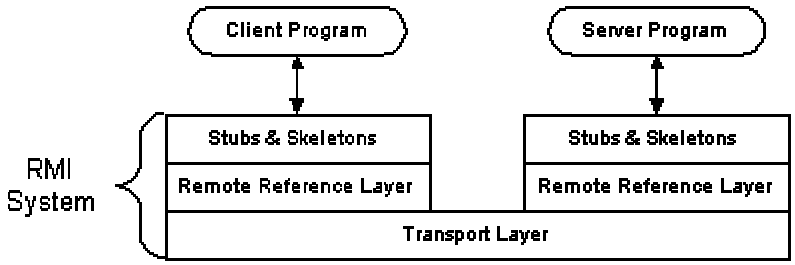
1. Java-RMI is a Java-specific middleware spec that allows client Java programs to invoke server Java objects as if they were local.
2. Java-RMI is tightly coupled with the Java language. Hence there are no separate IDL mappings that are required to invoke remote object methods. This is different from DCOM or CORBA where IDL mappings have to be created to invoke remote methods.
3. Since Java-RMI is tightly coupled with The Java Language, Java-RMI can work with true sub-classes. Neither DCOM nor CORBA can work with true subclasses since they are static object models.
4. Because of this, parameters passed during method calls between machines can be true Java Objects. This is impossible in DCOM or CORBA at present.
5. If a process in an RMI system receives an object of a class that it has never seen before, it can request that its class information be sent over the network.
6. Over and above all this, Java-RMI supports Distributed Garbage Collection that ties into the local Garbage Collectors in each JVM.

RMI Architecture Layers

The RMI implementation is essentially built from three abstraction layers.

1. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.
2. The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via *Remote Object Activation*.

3. The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



By using a layered architecture each of the layers could be enhanced or replaced without affecting the rest of the system. For example, the transport layer could be replaced by a UDP/IP layer without affecting the upper layers.

The RMI advantages

- You don't need to create a custom transaction protocol between client and server. The class files are all that's required.
- Simpler than CORBA.
- No fiddling with sockets, streams or parsing the way you do with CGI.
- Very flexible easy protocol, just like calling local objects to do the work. Once you have the objects defined, you can treat local and remote objects identically.

The RMI disadvantages are:

- Both client and server need access to the latest identical class definitions of the objects (or at least of their interfaces, stubs are automatically downloaded), something a traditional transaction processing or CGI environment does not require. In CGI, the code in client and server is independent. All that ties them together are the various messages exchanged
- Requires Java installed for both client and server. Does not work with any other language the way CORBA does. RMI is a Java-only solution. RMI can use IOP as its low level protocol to give it slightly more compatibility with CORBA
- Higher overhead than other techniques. If you don't keep your wits about you, you may inadvertently send huge reams of dependent pickled objects back and forth as a side effect of your remote procedure calls. Even at the best of times, the transmitted serialised objects are bulky. Objects contain considerable identifier text, versioning, and of course nested referenced objects.
- Requires two copies of the JVM running on the server, one for the registry and one for the server proper.

Stubs and Skeletons

One of the benefits of RMI is an almost illusionary, transparent networking. You can invoke methods on remote objects just as you would invoke a method on any other Java object. In fact, RMI completely masks whether the object you're invoking on is local or remote. This is called ***local/remote transparency***.

To mask that you're invoking on an object residing on a remote host, RMI needs to somehow simulate a local object that you can invoke on. This local object is called a *stub*.

It is responsible for accepting method calls locally and *delegating* those method calls to their actual object implementations, which are possibly located across the network. This effectively makes every remote invocation appear to be a local invocation. You can think of a **stub as a placeholder for an object that knows how to look over the network for the real object**. Because you invoke on local stubs, all the nasty networking issues are hidden.

Stubs are only half of the picture. We'd like the remote objects themselves—the objects that are being invoked on from remote hosts—to not worry about networking issues as well. **Just as a client invokes on a stub that is local to that client, your remote object needs to accept calls from a *skeleton* that is local to that remote object**. **Skeletons are responsible for receiving calls over the network (perhaps from a stub) and delegating those calls to the remote object implementation.** This is shown in Figure 1.

Your RMI implementation (that is, your J2EE server) should provide a means to *generate* the needed stubs and skeletons, thus relieving you of the networking burden. Typically, this is achieved through command-line tools. For example, Sun's J2EE reference implementation ships with a tool called ***rmic*** (which stands for the RMI compiler) to generate stub and skeleton classes. As you can see from Figure 1, you should deploy the stub on the client machine and the skeleton on the server machine.

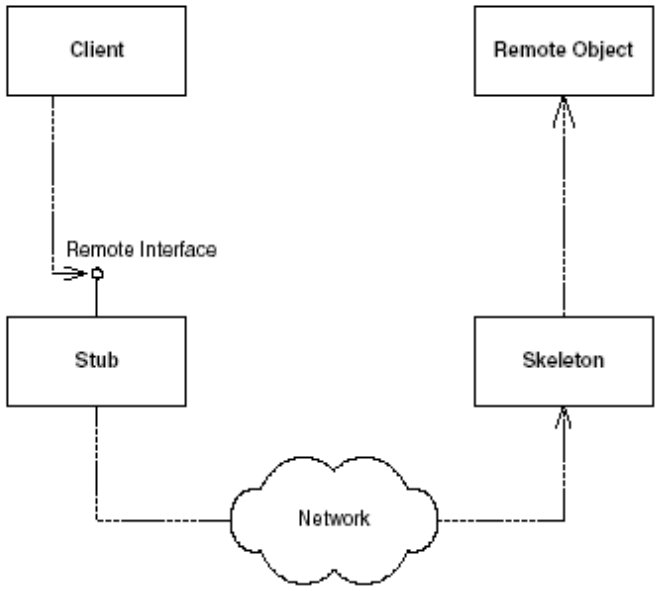


Figure 1 Stubs and skeletons.

A skeleton is a helper class that is generated across the RMI link. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object. You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

Parameter passing conventions

There are two major ways to pass parameters when calling a method**:** ***pass-by-value*** and ***pass-by-reference.*** However, pass-by-reference has a different meaning in RMI. It uses ***Remote Object.***

1. First way: RMI Pass-by-value

Parameters in a Single JVM

The normal semantics for Java technology is pass-by-value. When a parameter is passed to a method, the JVM makes a copy of the value, places the copy on the stack and then executes the method. When the code inside a method uses a parameter, it accesses its stack and uses the copy of the parameter. Values returned from methods are also copies.

When a primitive data type (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, or `double`) is passed as a parameter to a method, the mechanics of pass-by-value are straightforward. The mechanics of passing an object as a parameter are more complex. Recall that an object resides in heap memory and is accessed through one or more reference variables. And, while the following code makes it look like an object is passed to the method `println()`

```
String s = "Test";
System.out.println(s);
```

in the mechanics it is the reference variable that is passed to the method. In the example, a copy of reference variable `s` is made (increasing the reference count to the `String` object by one) and is placed on the stack. Inside the method, code uses the copy of the reference to access the object.

Primitive Parameters

When a primitive data type is passed as a parameter to a remote method, the RMI system passes it by value. R**MI will make a copy of a primitive data type and send it to the remote method.** If a method returns a primitive data type, it is also returned to the calling JVM by value.

Values are passed between JVMs in a standard, machine-independent format. This allows JVMs running on different platforms to communicate with each other reliably.

Object Parameters (that are pass-by-reference in non RMI world)

When an object is passed to a remote method, the semantics change from the case of the single JVM. RMI sends the object itself, not its reference, between JVMs. **It is the *object* that is passed by value, not the reference to the object. Normally, we pass a copy of reference. Similarly, when a remote method returns an object, a copy of the whole object is returned to the calling program.**

Unlike primitive data types, sending an object to a remote JVM is a nontrivial task. A Java object can be simple and self-contained, or it could refer to other Java objects in complex graph-like structure. Because different JVMs do not share heap memory, RMI must send the referenced object and all objects it references. (Passing large object graphs can use a lot of CPU time and network bandwidth.)

RMI uses a technology called ***Object Serialization*** to transform an object into a linear format that can then be sent over the network wire. Object serialization essentially flattens an object and any objects it references. Serialized objects can be de-serialized in the memory of the remote JVM and made ready for use by a Java program.

The actual parameter is *serialized* and passed using a network protocol to the target remote object. Serialization essentially "squeezes" the data out of an object/primitive. On the receiving end, that data is used to build a "clone" of the original object or primitive.

Serialization

Java introduces the concept of *object serialization* to handle this problem. ***Serialization*** is the conversion of a Java object into a bit-blob representation of that object. You can send bit-blobs anywhere. For example, you can use object serialization as an instant file-format for your objects and save them to your hard disk. RMI also uses object serialization to send parameters over the network.

When you're ready to use the object again, you must deserialize the bit blob back into a Java object. Then it's magically usable again.

The Java language handles the low-level details of serialization. In most cases, you don't need to worry about any of it. To tell Java that your object is **serializable**, your object must implement the **java.lang.Serializable** interface. That's all there is to it: Take this one simple step, and let Java handle the rest.

java.lang.Serializable defines no methods at all—it's simply a ***marker interface*** that identifies your object as something that can be serialized and deserialized.

You can provide your own custom serialization by implementing the *writeObject()* method on your object, or provide custom deserialization by implementing *readObject()*. This might be useful if you'd like to perform some sort of compression on your data

before your object is converted into a bit-blob and decompression after the bit-blob is restored to an object.

Figure 2 shows the serialization/deserialization process, where *writeObject()* is responsible for saving the state of the class, and *readObject()* is responsible for restoring the state of the class. These two methods will be called automatically when an object instance is being serialized or deserialized. If you choose not to define these methods, then the default serialization mechanisms will be applied. The default mechanisms are good enough for most situations.

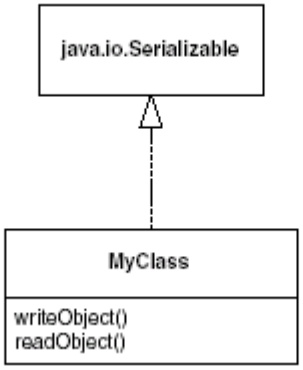


Figure 2 The Java serialization process.

Objects sent as arguments or returned as results across the network must be serializable. That is, they must implement the Serializable interface.

A class that implements the Serializable interface need not implement any special methods (as, for example, you must do if you implement, say, the Runnable interface).

Note that if a class is serializable, all its subclasses inherit this property. Many of the Java API provided classes are already serializable so there is no need to declare any of their subclasses serializable. You can check in the documentation.

However if you create your own class to be sent across the network, and it is only a subclass of, say, Object, you must declare it as implementing the serializable interface.

Another problem can sometimes arise when you serialize your own object creations. All the objects from which your object are composed must also be serializable, or your object cannot move down the net. (References to such non-serializable objects can be declared transient to avoid this problem.)

Marshaling

The term marshaling applies to the process of serializing. When a serialized object is to be sent across the network, it is "marshaled". **The object is converted into a byte stream for transmission. At the receiving end it is "unmarshaled**". The object is

reconstructed in its structured form from the incoming byte stream. In the RMI system, all this complex, low level, stuff is taken care of for you!

Rules for Serialization

Java serialization has the following rules for member variables held in serialized objects:

- Any basic primitive type (int, char, and so on) is automatically serialized with the object and is available when deserialized.

- Java objects can be included with the serialized bit-blob or not; it's your choice. The way you make your choice is as follows:

  - Objects marked with the ***transient*** keyword are not serialized with the object and are not available when deserialized.

  - Any object that is not marked with the transient keyword must implement *java.lang.Serializable*. These objects are converted to bit-blob format along with the original object. If your Java objects are neither transient nor implement *java.lang.Serializable*, a NotSerializableException is thrown when *writeObject()* is called.

Thus, when you serialize an object, you also serialize all nontransient subobjects as well. This means you also serialize all nontransient sub-subobjects (the objects referenced from the subobjects). This is repeated recursively for every object until the entire reference graph of objects is serialized. This recursion is handled automatically by Java serialization, as shown in Figure 3.

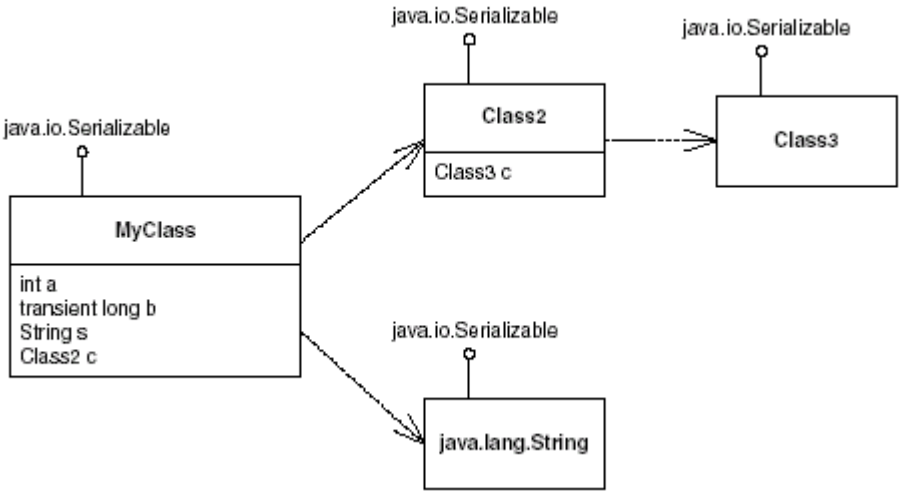


Figure 3 Object serialization recursion.

You simply need to make sure that each of your member objects implements the *java.lang.Serializable* interface. When serializing MyClass, Object Serialization will recurse through the dependencies shown, packaging up the entire graph of objects as a

stream. In this diagram, everything will get serialized except for transient long b, since it is marked as transient.

What Should You Make Transient?

How do you know which member variables should be marked transient and which should not? Here are some good reasons to mark an object as transient:

- The object is large. Large objects may not be suitable for serialization because operations you do with the serialized blob may be very intensive. Examples here include saving the blob to disk or transporting the blob across the network.

- The object represents a resource that cannot be reconstructed on the target machine. Some examples of such resources are database connections and sockets.

- The object represents sensitive information that you do not want to pass in a serialized stream.

Object Serialization and RMI

Java RMI relies on object serialization for passing parameters via remote method invocations. Figure 3 shows what MyObject's object graph could look like. Notice that every field and subfield is a valid type for Java serialization.

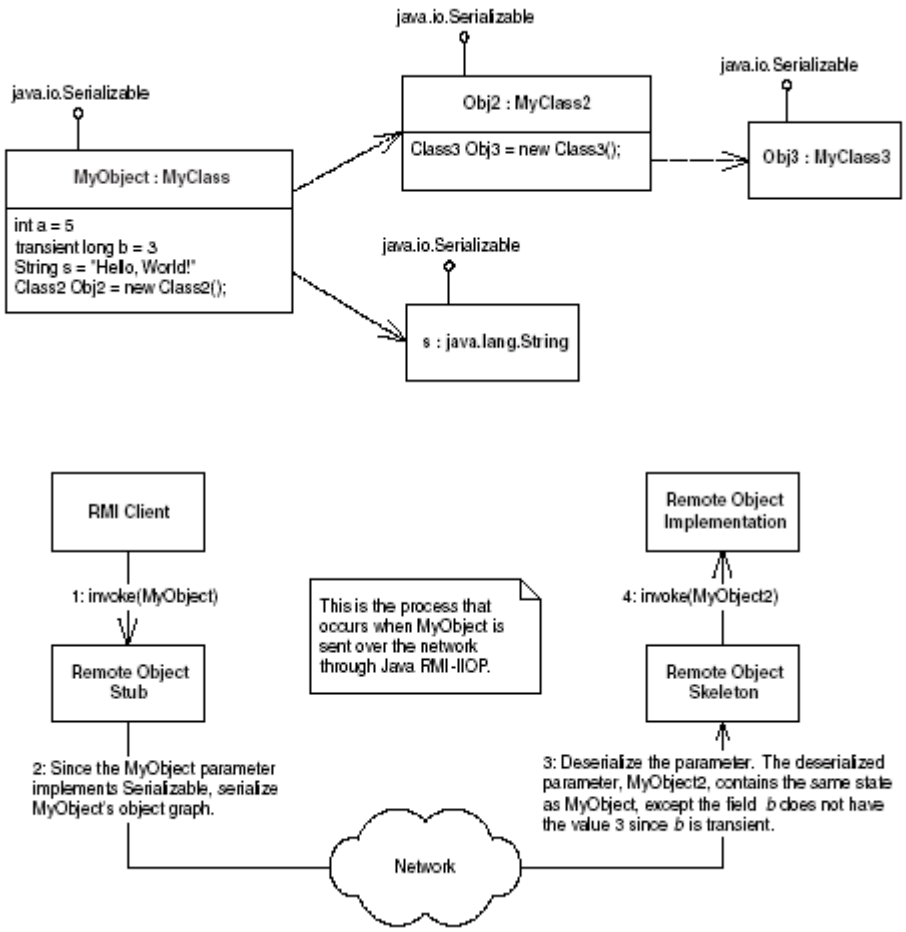


Figure 3 Java RMI and object serialization.

Figure 3 shows how RMI handles ***pass-by-value***, where an entire graph of objects is serialized into a bit-blob, sent across the network, and then deserialized on the target machine. But passing parameters by-value can lead to inefficiencies.

Remote method invocations are by no means simple. These are just some of the issues that arise: **Marshalling and unmarshalling.** RMIs (as well as RPCs) allow you to pass parameters, including Java primitives and Java objects, over the network. But what if the target machine represents data differently than the way you represent data? For example, what happens if one machine uses a different binary standard to represent numbers? The problem becomes even more apparent when you start talking about objects. What happens if you send an object reference over the wire? That pointer is not usable on the other machine because that machine's memory layout is completely different from yours.

***Marshalling* and *unmarshalling* is the process of massaging parameters so that they are usable on the machine being invoked on remotely**. It is the packaging and unpackaging of parameters so that they are usable in two heterogeneous environments.

RMI way pass-by-reference using Remote Object Parameters

RMI introduces a third type of parameter to consider: remote objects.

A client can obtain a remote reference, it can be returned to the client from a method call.

In the following code, the `BankManager` service `getAccount()` method is used to obtain a remote reference to an `Account` remote service.

```
BankManager bm;
Account     a;
try {
    bm = (BankManager)Naming.lookup("rmi://BankServer/BankManagerService");
    a  = bm.getAccount( "jGuru" );
    // Code that uses the account
}
catch (RemoteException re) {
}
```

In the implementation of `getAccount()`, the method returns a (local) reference to the remote service.

```
public Account getAccount(String accountName) {
    // Code to find the matching account
    AccountImpl ai =
     // return reference from search
    return AccountImpl;
}
```

When a method returns a local reference to an exported remote object, RMI does not return that object. Instead, it substitutes another object (the remote proxy for that service) in the return stream.

The following Figure 4 illustrates how RMI method calls might be used to:

- Return a remote reference from Server to Client A
- Send the remote reference from Client A to Client B
- Send the remote reference from Client B back to Server

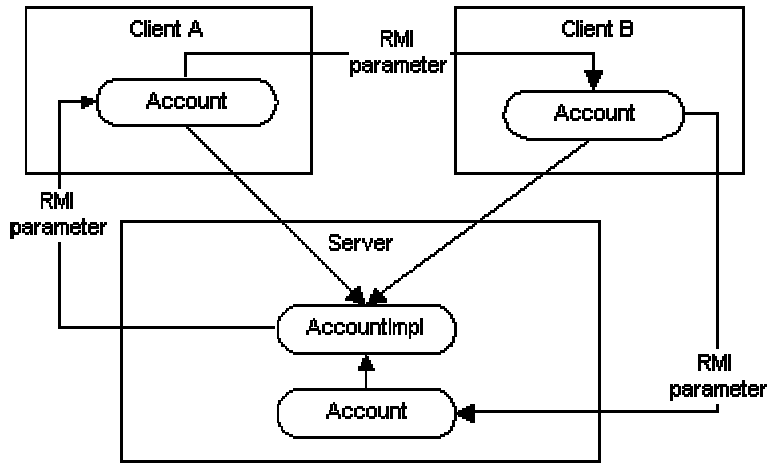


Figure 4.

Notice that when the `AccountImpl` object is returned to Client A, the `Account` proxy object is substituted. Subsequent method calls continue to send the reference first to Client B and then back to Server. During this process, the reference continues to refer to one instance of the remote service.

It is particularly interesting to note that when the reference is returned to Server, it is not converted into a local reference to the implementation object. While this would result in a speed improvement, maintaining this indirection ensures that the semantics of using a remote reference is maintained.

In other words, **RMI *simulates* a pass-by reference convention, which means the arguments are not copied over. Rather, the server modifies the client's copy of the parameter.**

If you want to pass a parameter by-reference, the parameter must itself be a remote object. **The parameter is thus an object that is callable remotely**. **When the client calls the server, the RMI runtime sends a stub to that remote object to the server. *The server can perform a callback on that stub, which connects the server to the remote object living on the client machine*.**

Figure 5 shows the process that occurs when *MyRemoteObject,* an instance of *My RemoteClass,* is sent over the network through Java RMI/RMI-IIOP.

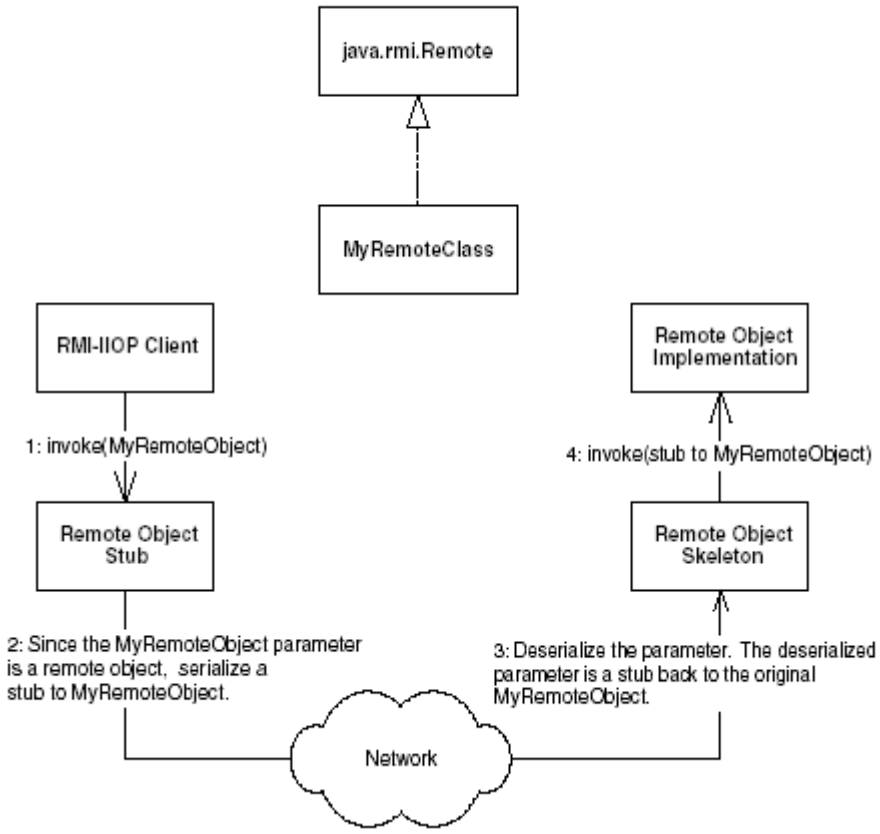


Figure 5. Pass-by-reference with Java RMI/RMI-IIOP.

The best way to understand this paradigm is by analogy. In Java programming language, when you pass an object as a parameter, the object reference is copied. In RMI, when you pass an object as a parameter, the stub is copied. Both of these strategies achieve pass-by-reference because they are cloning the thing that points to the object, rather than the object itself.

Because Java RMI stubs are also serializable, they are passable over the network as a bit-blob. This is why strickly specking *all* parameters in Java RMI-IIOP are passed by-value.

Thus, Java RMI-IIOP only *simulates* pass-by reference by passing a serializable stub, rather than serializing the original object. By making your parameters remote objects, you can effectively avoid the network lag in passing large objects.

The actual parameter, which *is itself a remote object*, is represented by a proxy. The proxy keeps track of where the actual parameter lives, and anytime the target method uses the formal parameter, *another remote method invocation occurs* to "call back" to the actual parameter. This can be useful if the actual parameter points to a large object (or graph of objects) and there are few call backs.

Pass-by-reference does not make a copy. With pass-by reference, any modifications to parameters made by the remote host affect the original data. The flexibility of both the pass-by-reference and pass-by value models is advantageous, and RMI supports both.

Unlike a local Java call, an RMI invocation passes local objects in parameters by value, instead of by reference. Why pass by value? Because a reference to a local object is only useful with in a single virtual machine. On the other hand, every remote object is passed by reference and not by value - just like CORBA.

Local method invocation

Arguments to a method are always "pass by value", as in C. That is, the method receives copies of the actual parameters. However, in many cases, these arguments are references to objects. Objects are effectively "pass by reference" because using the (copy) of the reference, you can change the public members of the actual object. The object itself is not copied.

With RMI, even if the arguments for a remote method (on the server) are references to (local on the client) objects, what arrives at the server are copies of the actual objects referered to, not the references to those objects, as would be the case for a local call.

Therefore, changes made to the object on the remote server are not reflected by on the original object still sitting on the client machine. The only way to get a change back from the remote method is via the return value of the method.

The situation is more like it would be in a functional language such as Scheme or ML.

One more time

1. There are two kinds of object in RMI, ordinary ones that don't implement the Remote interface, and ones that do.

Ones that do, have **RMIC** stubs on both client (called a stub) and server. When you pass an object as a parameter to a remote procedure, it may go either by reference or by value. Only the server has the actual code for the class. RMI is a peer to peer protocol, so any station may act both as client and server.

Ordinary objects go by value. The object may travel in either direction, client to server or server to client, and may be passed as a parameter or returned as a result. The object's data fields are pickled by serialisation into an ObjectStream and reconstituted on the other end. You end up with two independent objects, one local and one remote. Changes made to one won't be reflected in the other.

2. When you pass an object that implements Remote as a parameter, it goes by reference. It may travel in either direction, client to server or server to client, and may be passed as a parameter or returned as a result. A dummy proxy object is created at the other end, but the master copy of the object with its data fields remains behind. When the other ends wants to find out about data in the object, or change data in the object, (by invoking its methods), it invokes the interface methods of the local proxy object which in turn transparently invokes the methods of the remote master object and returns the results.

Note: Microsoft Internet Explorer does not support RMI directly. You are better off using the Java Plug-in, or pre-installing the RMI class library, or running as an application rather than an Applet

A simpler alternative to RMI is to send raw serialized objects, perhaps compressed, back and forth across a socket connection. Both ends need matching class definitions for the objects. This gives you the ability to transmit clean, compact, internal format data, but not to run arbitrary methods at the remote end. To control the other end, you would have to send objects with embedded command fields in them. This saves you much parsing and data validation over the traditional text streams. Sender and receiver simply agree on a common class definition to represent a given message or group of messages.

Summary

We have the following rules for passing objects using Java RMI/RMI-IIOP:

- All Java basic primitives are passed by-value when calling methods remotely. This means copies are made of the parameters. Any changes to the data on the remote host are not reflected in the original data.

- If you want to pass an object over the network by-value, it must implement java.lang.Serializable. Anything referenced from within that object must follow the rules for Java serialization. Again, any changes to the data on the remote host are not reflected in the original data.

- If you want to pass an object over the network by-reference, it must be a remote object, and it must implement **java.rmi.Remote**. A stub for the remote object is serialized and passed to the remote host. The remote host can then use that stub to invoke callbacks on your remote object. There is only one copy of the object at any time, which means that all hosts are calling the same object.

Network or machine instability

With a single JVM application, a crash of the JVM brings the entire application down. But consider a distributed object application, which has many JVMs working together to solve a business problem. In this scenario, a crash of a single JVM should not cause the distributed object system to grind to a halt. To enforce this, remote method invocations need a standardized way of handling a JVM crash, a machine crash, or network instability. When some code performs a remote invocation, the code should be informed of any problems encountered during the operation. RMI performs this for you, abstracting out any JVM, machine, or network problems from your code.

As you can see, there's a lot involved in performing RMIs. RMI contains measures to handle many of these nasty networking issues for you. This reduces the total time spent dealing with the distribution of your application, allowing you to focus on the core functionality.

The Steps involved in developing and deploying a Java/RMI **Server** using JRMP (Java Remote Method Protocol) are

1. **Develop your Remote Interface**
2. **Implement your Java/RMI Server**
3. **Implement an application that creates your server**
4. **Develop your policy file**
5. **Compile the files, generate stubs & skeletons, run the RMIRegistry and startup your server**

The Remote Interface

We begin our exploration of RMI by reviewing one of object-oriented design's great programming practices—the separation of the interface of code from its implementation.

The interface defines the exposed information about an object, such as the names of its methods and what parameters those methods take. It's what the client works with. The interface masks the implementation from the viewpoint of clients of the object, so clients deal only with the end result: the methods the object exposes.

The implementation is the core programming logic that an object provides. It has some very specific algorithms, logic, and data.

By separating interface from implementation, you can vary an object's proprietary logic without changing any client code. For example, you can plug in a different algorithm that performs the same task more efficiently.

RMI makes extensive use of this concept. **All networking code you write is applied to interfaces, *not* implementations**. In fact, you *must* use this paradigm in RMI—you do

not have a choice. **It is impossible to perform a remote invocation directly on an object implementation**. You can operate solely on the interface to that object's class.

Therefore, when using RMI, you must build a custom interface, called a ***remote interface***. This remote interface should ***extend*** the interface ***java.rmi.Remote***. Your interface should have within it a copy of each method your remote object exposes.

With RMI, you can never fully separate your application from the network. At some point, you'll need to deal with remote exceptions being thrown due to networking issues. Some may consider this a limitation of RMI because the network is not entirely seamless: Remote exceptions force you to differentiate a local method from a remote method. But in some ways, this is an advantage of RMI as well. Interlacing your code with remote exceptions forces you to think about the network and encourages distributed object developers to consider issues such as the network failing, the size of parameters going across the network, and more.

Server. Step 1. Develop your Remote Interface

```
// PKGenerator.java.

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

/**
* The remote interface for the remote object.  Clients use this
* remote interface to perform any operations on the remote object.
*/
public interface PKGenerator extends Remote {
    public long generate(int base) throws RemoteException;
}
```

The Remote Object Implementation

Remote objects **are networked object implementations that can be called by another JVM. They *implement* a remote interface and thus expose methods that can be invoked by remote clients.**

The physical locations of remote objects and the clients that invoke them are not important. For example, it is possible for a client running in the same address space as a remote object to invoke a method on that object. It's also possible for a client across the Internet to do the same thing. To the remote object, both invocations appear to be the same.

To make your object a remote object available to be invoked on by remote hosts, your remote class must perform *one* of the following steps:

Extend the class *java.rmi.server.UnicastRemoteObject*;.
java.rmi.server.UnicastRemoteObject is a base class from which you can derive your remote objects. When your remote object is constructed, it automatically calls the `UnicastRemoteObject` 's constructor, which makes the object available to be called remotely.

Don't extend *java.rmi.server.UnicastRemoteObject*. Perhaps your remote object class needs to inherit implementation from another custom class. In this case, because Java does not allow for multiple implementation inheritance, you cannot extend `UnicastRemoteObject`. If you do this, you must manually export your object so that it is available to be invoked on by remote hosts. To export your object, call *java.rmi.server.UnicastRemoteObject.exportObject().*

Now let's create the remote object class. This class implements the *IPKGenerator* interface

```
//PKGeneratorImpl.java.

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**
* The remote object which generates primary keys
*/
public class PKGeneratorImpl extends UnicastRemoteObject implements PKGenerator {
    private static long i= System.currentTimeMillis();

    /*
    * Our remote object's constructor
    */
    public PKGeneratorImpl(String name) throws Exception, RemoteException {
        try {
            /* Rebinds the specified name to a new remote object. */
            java.rmi.Naming.rebind(name, this);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }

    /*
    * Generates a unique primary key */
    public synchronized long generate(int factor) throws RemoteException {
        return (i++)*factor;
    }
}}
```

In our remote object constructor, the **superclass makes our object available to be called remotely**. **This makes the object available at a random port number**.

Once the remote object's constructor is complete, this object is available forever for any virtual machine to invoke on; that is, until someone calls *unexportObject()*.

<u>Issues with Our Primary Key Generation Algorithm</u>
Our primary key generation algorithm is to simply increment a number each time someone calls our server. This generator overcomes <u>two common challenges</u> when writing an RMI implementation:
1. <u>Threading</u>. RMI allows many clients to connect to a server at once. Thus, our remote object implementation may have many threads running inside of it.

But when generating primary keys, we never want to generate a duplicate key because our keys are not unique and thus would not be good candidates to use in a

database. Therefore, it is important to have the synchronized block around the *generate()* method, so that only one client can generate a primary key at once.

2. JVM crashes. We must protect against a JVM crash (or hardware failure). Thus, we initialize our generator to the current time (the number of milliseconds that have elapsed since 1970). This is to ensure that our primary key generator increases monotonically (that is, primary keys are always going up in value) in case of a JVM crash. Note that we haven't considered daylight savings time resulting in duplicate keys. If we were to use this code in production, we would need to account for that.

What makes Java-RMI tick

Since both the client and the server may reside on different machines/processes, there needs to be a mechanism that can establish a relationship between the two. Java-RMI uses a network-based registry program called *RMIRegistry* to keep track of the distributed objects.

Note: The RMI Registry is an RMI server itself!!!

The RMI Registry

A special program called the Registry runs on the server. This has absolutely nothing to do with the accursed Windows registry. The RMI Registry is a totally separate EXE from the server JVM**. The objects themselves live in the server's address space, not the registry's**. The registry allows server objects to register themselves as available to the clients. Clients can find a registered object by asking for it by name. Since each client gets given a reference to the single copy of the registered object running on the server, that object would be very busy. Typically it is a factory object that spins off other objects and threads that do the actual work. The registry is just to get started. **Once you have a reference, you no longer need the registry. You can pass your reference onto others so they can use it without ever talking to the registry**. Why is the Registry a separate program?

- So that the Registry could run on a separate server, thus sharing the workload.
- After all the clients have found all the Objects they need, you can shut down the Registry and reclaim the resources it was using.

The server object makes methods available for remote invocation by binding it to a name in the RMI Registry. The client object can thus check for the availability of a certain server object by looking up its name in the registry. The RMI Registry thus acts as a central management point for Java-RMI. The RMI Registry is thus a simple name repository. It does not address the problem of actually invoking remote methods.

The RMI registry is an application that runs as a background process on a remote server. It contains a table of named services and remote objects, serving as the communications gateway between the client, for example applet, and the remote object.

The remote object also executes as a background process on the remote server and registers, or *binds*, itself to the RMI registry using a service name. The registry stores the remote object's service name and a reference to the remote object. (The reference is actually to the remote object's skeleton.)

Finally, an application loads the remote object's bindery from the RMI registry. It uses this to understand the structure of the remote object's method parameters. From this point on, the client simply calls, or invokes, the remote object's methods as if they were local to it. All arguments are passed between the applet and remote object's methods using a technique called object serialization.

Server. Step 3. Implement an Application that creates your Server

PKGeneratorServer.java

```
/**
* PKGeneratorServer.java
* Implement an Application that creates Java RMI Server
*/

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import pkgenerator.*;

public class PKGeneratorServer {
    public static void main(String[] args) throws Exception {
        if(System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        pkgenerator.PKGeneratorImpl myObject= new
        pkgenerator.PKGeneratorImpl("PKGENERATOR");

        System.out.println( "RMI PKGeneratorServer ready...");
    }
}
```

Server. Step 4. Develop your security policy file

policy.all

```
grant {
    permission java.security.AllPermission "", "";
};
```

A Note about Security

The JDK 1.2 security model is more sophisticated than the model used for JDK 1.1. JDK 1.2 contains enhancements for finer-grained security and requires code to be granted specific permissions to be allowed to perform certain operations.

In JDK 1.1 code in the class path is trusted and can perform any operation; downloaded code is governed by the rules of the installed security manager. If you run this example in JDK 1.2, you need to specify a policy file when you run your server and client. Here is a general policy file that allows downloaded code, from any code base, to do two things:

- Connect to or accept connections on unprivileged ports (ports greater than 1024) on any host
- Connect to port 80 (the port for HTTP)

Here is the code for the general policy file:

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

If you make your code available for downloading via HTTP URLs, you should use the preceding policy file when you run this example. However, if you use file URLs instead, you can use the following policy file. Note that in Windows-style file names, the backslash character needs to be represented by two backslash characters in the policy file.

```
grant {
    permission java.net.SocketPermission "*:1024-65535",
        "connect,accept";
    permission java.io.FilePermission
        "c:\\home\\ann\\public_html\\classes\\-", "read";
    permission java.io.FilePermission
        "c:\\home\\jones\\public_html\\classes\\-", "read";
};
```

This example assumes that the policy file is called `java.policy` and that it contains the appropriate permissions. If you run this example on JDK 1.1, you will not need to use a policy file, since the `RMISecurityManager` provides all of the protection you need.

Server. Step 5. Compile the Files, generate the stubs & skeletons, startup the RMIRegistry and Run the Server

```
C:\Projects\Forte\...\RMI\PKGenerator>javac *.java
C:\Projects\Forte\...\RMI\PKGenerator>rmic PKGeneratorImpl
C:\Projects\Forte\...\RMI\PKGenerator>start rmiregistry
C:\Projects\Forte\...\RMI\PKGenerator>java -Djava.security.policy=policy.all
PKGeneratorServer

RMI PKGeneratorServer ready...
```

The Steps involved in developing a Java/RMI Client are

1. **Develop your Java/RMI Client**
2. **Develop your policy file**
3. **Compile the files and run the client**

Client. Step 1. Develop your Java/RMI Client

```
//PKGeneratorClient.java
/**
* PKGeneratorClient.java
* Develop your Java/RMI Client
*/
import java.rmi.*;
import java.rmi.registry.*;

public class PKGeneratorClient {
    public static void main(String[] args) {
       try {
           if(System.getSecurityManager() == null) {
               System.setSecurityManager(new RMISecurityManager());
           }
           PKGenerator generator=
(PKGenerator)Naming.lookup("rmi://localhost/PKGENERATOR");

           System.out.println("The unique number is= " + generator.generate(2));
       }
       catch(Exception e) {
           System.out.println(e);
       }
    }
}
```

Client. Step 2. Develop your security policy file

policy.all

```
grant {
    permission java.security.AllPermission "", "";
};
```

Client. Step 3. Compile the files and run the client

```
C:\Projects\Forte\_...\RMI\PKGenerator\client>copy ..\PKGenerator.class .
        1 file(s) copied.

C:\Projects\Forte\…\PKGenerator\client>copy ..\policy.all .
        1 file(s) copied.
C:\Projects\Forte\...\RMI\PKGenerator\client>copy ..\PKGeneratorImpl_Stub.class .
        1 file(s) copied.

C:\Projects\Forte\...\RMI\PKGenerator\client>javac *.java

C:\Projects\Forte\… \RMI\PKGenerator\client>java -Djava.security.policy=policy.all
PKGeneratorClient
The unique number is= 2085143700732
```

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Java Remote Method Invocation

The Steps involved in developing and deploying a Java/RMI **Server** using JRMP (Java Remote Method Protocol) are

1. **Develop your Remote Interface**
2. **Implement your Java/RMI Server**
3. **Implement an application that creates your server**
4. **Develop your policy file**
5. **Compile the files, generate stubs & skeletons, run the RMIRegistry and startup your server**

Server. Step 1. Develop your Remote Interface

//StockMarket.java

```
package simpleStocks;
/**
* StockMarket.java
* Server side . Step 1. Develop your Remote Interface
*
*/

import java.rmi.*;

public interface StockMarket extends java.rmi.Remote {
    float getPrice(String symbol) throws RemoteException;
}
```

Server. Step 2. Implement your Java RMI Server

StockMarketImpl.java

```
package simpleStocks;

/**
* Implement your Java RMI Server
*/

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class StockMarketImpl extends UnicastRemoteObject implements StockMarket {
    public StockMarketImpl(String name) throws RemoteException {
        try {
            /* Rebinds the specified name to a new remote object. */
            java.rmi.Naming.rebind(name, this);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }

    public float getPrice(String symbol) {
        float price= 0;

        for(int i = 0; i < symbol.length(); i++) {
            price +=(int)symbol.charAt(i);
        }
        price /= 5;
```

```
        return price;
    }
}
```

Server. Step 3. Implement an Application that creates your Server

StockMarketServer.java

```
/**
* StockMarketServer.java
* Implement an Application that creates Java RMI Server
*/

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import simpleStocks.*;

public class StockMarketServer {
    public static void main(String[] args) throws Exception {
        if(System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        simpleStocks.StockMarketImpl myObject= new impleStocks.StockMarketImpl("NASDAQ");

        System.out.println( "RMI StockMarketServer ready...");
    }
}
```

Server. Step 4. Develop your security policy file

policy.all

```
grant {
    permission java.security.AllPermission "", "";
};
```

Server. Step 5. Compile the Files, generate the stubs & skeletons, startup the RMIRegistry and Run the Server

```
E:\MyProjects\StockRMI\SimpleStocks>
E:\MyProjects\StockRMI\SimpleStocks>javac *.java
E:\MyProjects\StockRMI\SimpleStocks>cd..
E:\MyProjects\StockRMI>rmic SimpleStocks.StockMarketImpl
E:\MyProjects\StockRMI>javac *.java
E:\MyProjects\StockRMI>start rmiregistry
E:\MyProjects\StockRMI>java -Djava.security.policy=policy.all
StockMarketServer
RMI StockMarketServer ready...
```

```
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>javac -d . StockMarket.java
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>javac -d . StockMarketImpl.java
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>mkdir jars
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>jar -cvf jars\stockMarket.jar
simpleStocks\*.*
added manifest
adding: simpleStocks/StockMarket.class(in = 260) (out= 198)(deflated 23%)
adding: simpleStocks/StockMarketImpl.class(in = 835) (out= 512)(deflated 38%)
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>javac -classpath .;jars\stocMarket.jar
StockMarketServer.java
```

```
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>rmic simpleStocks.StockMarketImpl
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>start rmiregistry
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>java -Djava.security.policy=policy.all
StockMarketServer
RMI StockMarketServer ready...
```

The Steps involved in developing a Java/RMI Client are

1. **Develop your Java/RMI Client**
2. **Develop your policy file**
3. **Compile the files and run the client**

Client. Step 1. Develop your Java/RMI Client

StockMarketClient.java

```
/**
* StockMarketClient.java
* Develop your Java/RMI Client
*/
import java.rmi.*;
import java.rmi.registry.*;
import simpleStocks.*;

public class StockMarketClient {
    public static void main(String[] args) {
       try {
           if(System.getSecurityManager() == null) {
               System.setSecurityManager(new RMISecurityManager());
           }
           StockMarket market= (StockMarket)Naming.lookup("rmi://localhost/NASDAQ");

           System.out.println("The price of MY COMPANY is " + market.getPrice("ADP"));
       }
       catch(Exception e) {
           System.out.println(e);
       }
    }
}
```

Client. Step 2. Develop your security policy file

policy.all

```
grant {
    permission java.security.AllPermission "", "";
};
```

Client. Step 3. Compile the files and run the client

```
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>javac -classpath .\jars\stockMarket.jar
StockMarketClient.java
C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>java -Djava.security.policy=policy.all
StockMarketClient

The price of MY COMPANY is 159.2

C:\Projects\Forte\_CST407_2\TestAllWinter03\RMI>
```

```
E:\MyProjects\StockRMI>
E:\MyProjects\StockRMI>javac *.java
E:\MyProjects\StockRMI>java -Djava.security.policy=policy.all
                                                    StockMarketClient
The price of MY COMPANY is 159.2
E:\MyProjects\StockRMI>
```