Remote Method Invocation (RMI)

Course: Distributed Computing
Instructor: Zulqarnain
Email: Zulqarnain@iiu.edu.pk

- **Introduction to Distributed Computing with RMI**
  - Goals
  - Comparison of Distributed and Nondistributed Java Programs

- **Java RMI Architecture**
  - Interfaces: The Heart of RMI
  - RMI Architecture Layers
    - Stub and Skeleton Layer
    - Remote Reference Layer
    - Transport Layer

- **Naming Remote Objects**

- **Using RMI**
  - Interfaces
  - Implementation
  - Stubs and Skeletons
  - Host Server
  - Client
  - Running the RMI System

- **Parameters in RMI**

- **Parameters in a Single Java™ Virtual Machine**
  - Primitive Parameters
  - Object Parameters
  - Remote Object Parameters

- **RMI Client-Side Callbacks**

- **Distributing and Installing RMI Software**
  - Distributing RMI Classes
  - Automatic Distribution of Classes
  - Firewall Issues

- **Distributed Garbage Collection**

- **Serializing Remote Objects**

Introduction to Distributed Computing with RMI

Remote Method Invocation (RMI) technology, first introduced in JDK™ 1.1, elevates network programming to a higher plane. Although RMI is relatively easy to use, it is a remarkably powerful technology and exposes the average Java developer to an entirely new paradigm--the world of distributed object computing.

This course provides you with an in-depth introduction to this versatile technology. RMI has evolved considerably since JDK 1.1, and has been significantly upgraded under the Java 2 SDK. Where applicable, the differences between the two releases will be indicated.

Goals

A primary goal for the RMI designers was to allow programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs. To do this, they had to carefully map how Java classes and objects work in a single Java™ Virtual Machine (JVM) to a new model of how classes and objects would work in a distributed (multiple JVM) computing environment.

This section introduces the RMI architecture from the perspective of the distributed or remote Java objects, and explores their differences through the behavior of local Java objects. The RMI architecture defines how objects behave, how and when exceptions can occur, how memory is managed, and how parameters are passed to, and returned from, remote methods.

Comparison of Distributed and Nondistributed Java Programs

The RMI architects tried to make the use of distributed Java objects similar to using local Java objects. While they succeeded, some important differences are listed in the table below.

Do not worry if you do not understand all of the difference. They will become clear as you explore the RMI architecture. You can use this table as a reference as you learn about RMI.

	Local Object	Remote Object
Object Definition	A local object is defined by a Java class.	A remote object's exported behavior is defined by an interface that must extend the `Remote` interface.
Object Implementation	A local object is implemented by its Java class.	A remote object's behavior is executed by a Java class that implements the remote interface.
Object Creation	A new instance of a local object is created by the `new` operator.	A new instance of a remote object is created on the host computer with the `new` operator. A client cannot directly create a new remote object (unless using Java 2 Remote Object Activation).
Object Access	A local object is accessed directly via an object reference variable.	A remote object is accessed via an object reference variable which points to a proxy stub implementation of the remote interface.

References	In a single JVM, an object reference points directly at an object in the heap.	A "remote reference" is a pointer to a proxy object (a "stub") in the local heap. That stub contains information that allows it to connect to a remote object, which contains the implementation of the methods.
Active References	In a single JVM, an object is considered "alive" if there is at least one reference to it.	In a distributed environment, remote JVMs may crash, and network connections may be lost. A remote object is considered to have an active remote reference to it if it has been accessed within a certain time period (the lease period). If all remote references have been explicitly dropped, or if all remote references have expired leases, then a remote object is available for distributed garbage collection.
Finalization	If an object implements the `final ize()` method, it is called before an object is reclaimed by the garbage collector.	If a remote object implements the `Unreferenced` interface, the unreferenced method of that interface is called when all remote references have been dropped.
Garbage Collection	When all local references to an object have been dropped, an object becomes a candidate for garbage collection.	The distributed garbage collector works with the local garbage collector. If there are no remote references and all local references to a remote object have been dropped, then it becomes a candidate for garbage collection through the normal means.
Exceptions	Exceptions are either Runtime exceptions or Exceptions. The Java compiler forces a program to handle all Exceptions.	RMI forces programs to deal with any possible `RemoteException` objects that may be thrown. This was done to ensure the robustness of distributed applications.

Java RMI Architecture

The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI architects have succeeded; creating a system that extends the safety and robustness of the Java architecture to the distributed computing world.

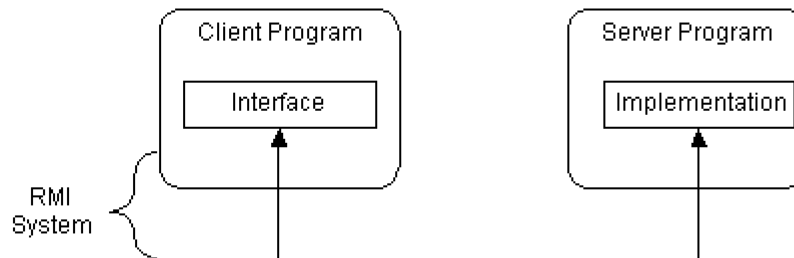
Interfaces: The Heart of RMI

The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

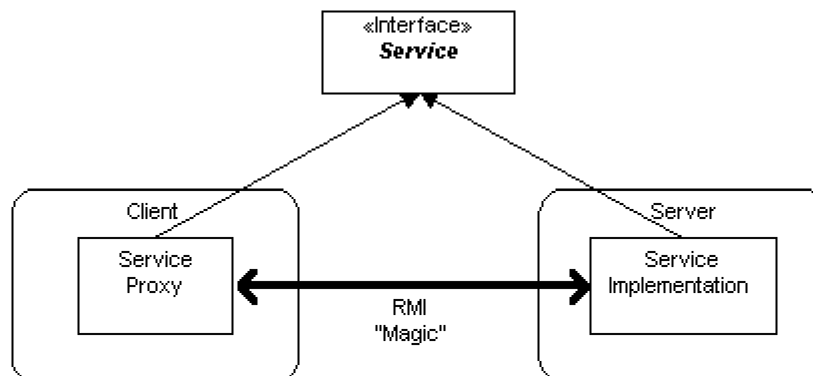
This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service.

Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that *interfaces define behavior* and *classes define implementation*.

While the following diagram illustrates this separation,



remember that a Java interface does not contain executable code. RMI supports two classes that implement the same interface. The first class is the implementation of the behavior, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client. This is shown in the following diagram.



A client program makes method calls on the proxy object, RMI sends the request to the remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

RMI Architecture Layers

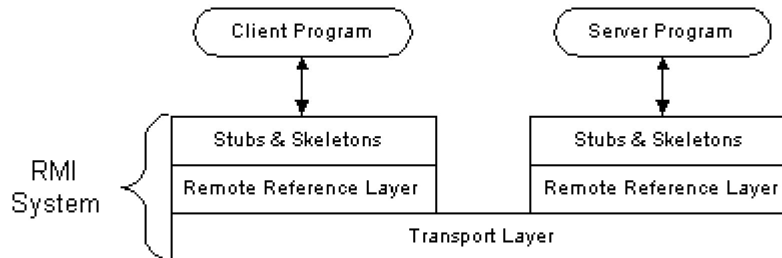
With an understanding of the high-level RMI architecture, take a look under the covers to see its implementation.

The RMI implementation is essentially built from three abstraction layers. 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.

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*.

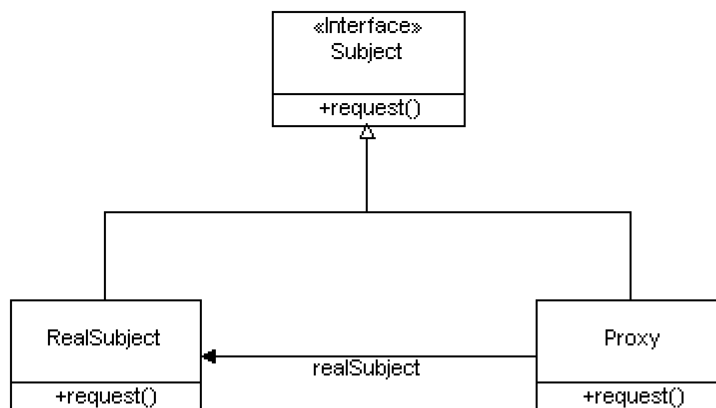
The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



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.

Stub and Skeleton Layer

The stub and skeleton layer of RMI lie just beneath the view of the Java developer. In this layer, RMI uses the Proxy design pattern. In the Proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects. The following class diagram illustrates the Proxy pattern.



In RMI's use of the Proxy pattern, the stub class plays the role of the proxy, and the remote service implementation class plays the role of the `RealSubject`.

A skeleton is a helper class that is generated for RMI to use. The skeleton understands how to communicate with the stub 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.

Remote Reference Layer

The Remote Reference Layers defines and supports the invocation semantics of the RMI connection. This layer provides a `RemoteRef` object that represents the link to the remote service implementation object.

The stub objects use the `invoke()` method in `RemoteRef` to forward the method call. The `RemoteRef` object understands the invocation semantics for remote services.

The JDK 1.1 implementation of RMI provides only one way for clients to connect to remote service implementations: a unicast, point-to-point connection. Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system. (If it is the primary service, it must also be named and registered in the RMI Registry).

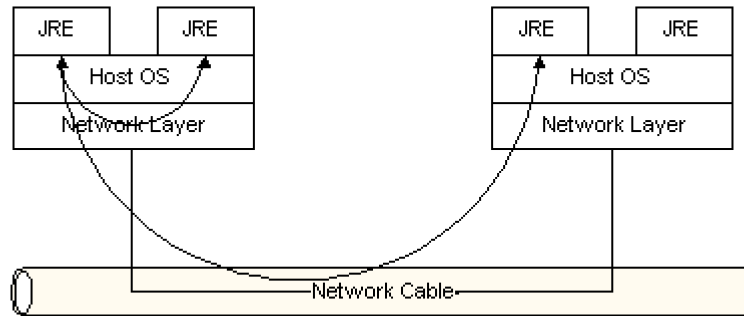
The Java 2 SDK implementation of RMI adds a new semantic for the client-server connection. In this version, RMI supports activatable remote objects. When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects.

Other types of connection semantics are possible. For example, with multicast, a single proxy could send a method request to multiple implementations simultaneously and accept the first reply (this improves response time and possibly improves availability). In the future, Sun may add additional invocation semantics to RMI.

Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. (This is why you must have an operational TCP/IP configuration on your computer to run the Exercises in this course). The following diagram shows the unfettered use of TCP/IP connections between JVMs.



As you know, TCP/IP provides a persistent, stream-based connection between two machines based on an IP address and port number at each end. Usually a DNS name is used instead of an IP address; this means you could talk about a TCP/IP connection between `flicka.magelang.com:3452` and `rosa.jguru.com:4432`. In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections.

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary, stream-based protocol that is only partially `specified` is now in two versions. The first version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server. The second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes. (Note that some alternate implementations, such as BEA Weblogic and NinjaRMI *do not* use JRMP, but instead use their own wire level protocol. ObjectSpace's Voyager does recognize JRMP and will interoperate with RMI at the wire level.)

Sun and IBM have jointly worked on the next version of RMI, called RMI-IIOP, which will be available with Java 2 SDK Version 1.3. The interesting thing about RMI-IIOP is that instead of using JRMP, it will use the Object Management Group (OMG) Internet Inter-ORB Protocol, IIOP, to communicate between clients and servers.

The OMG is a group of more than 800 members that defines a vendor-neutral, distributed object architecture called Common Object Request Broker Architecture (CORBA). CORBA Object Request Broker (ORB) clients and servers communicate with each other using IIOP. With the adoption of the Objects-by-Value extension to CORBA and the Java Language to IDL Mapping proposal, the ground work was set for direct RMI to CORBA integration. This new RMI-IIOP implementation supports most of the RMI feature set, except for:

- java.rmi.server.RMISocketFactory
- UnicastRemoteObject
- Unreferenced
- The DGC interfaces

The RMI transport layer is designed to make a connection between clients and server, even in the face of networking obstacles.

While the transport layer prefers to use multiple TCP/IP connections, some network configurations only allow a single TCP/IP connection between a client and server (some browsers restrict applets to a single network connection back to their hosting server).

In this case, the transport layer multiplexes multiple virtual connections within a single TCP/IP connection.

Naming Remote Objects

During the presentation of the RMI Architecture, one question has been repeatedly postponed: "How does a client find an RMI remote service? " Now you'll find the answer to that question. Clients find remote services by using a naming or directory service. This may seem like circular logic. How can a client locate a service by using a service? In fact, that is exactly the case. A naming or directory service is run on a well-known host and port number.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI). RMI itself includes a simple service called the RMI Registry, `rmiregistry`. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

On a host machine, a server program creates a remote service by first creating a local object that implements that service. Next, it exports that object to RMI. When the object is exported, RMI creates a listening service that waits for clients to connect and request the service. After exporting, the server registers the object in the RMI Registry under a public name.

On the client side, the RMI Registry is accessed through the static class `Naming`. It provides the method `lookup()` that a client uses to query a registry. The method `lookup()` accepts a URL that specifies the server host name and the name of the desired service. The method returns a remote reference to the service object. The URL takes the form:

```
rmi://<host_name>
      [:<name_service_port>]
            /<service_name>
```

where the `host_name` is a name recognized on the local area network (LAN) or a DNS name on the Internet. The `name_service_port` only needs to be specified only if the naming service is running on a different port to the default 1099.

Using RMI

It is now time to build a working RMI system and get hands-on experience. In this section, you will build a simple remote calculator service and use it from a client program.

A working RMI system is composed of several parts.

- Interface definitions for the remote services
- Implementations of the remote services
- Stub and Skeleton files
- A server to host the remote services
- An RMI Naming service that allows clients to find the remote services
- A class file provider (an HTTP or FTP server)
- A client program that needs the remote services

In the next sections, you will build a simple RMI system in a step-by-step fashion. You are encouraged to create a fresh subdirectory on your computer and create these files as you read the text.

To simplify things, you will use a single directory for the client and server code. By running the client and the server out of the same directory, you will not have to set up an HTTP or FTP server to provide the class files.

Assuming that the RMI system is already designed, you take the following steps to build a system:

1. Write and compile Java code for interfaces
2. Write and compile Java code for implementation classes
3. Generate Stub and Skeleton class files from the implementation classes
4. Write Java code for a remote service host program
5. Develop Java code for RMI client program
6. Install and run RMI system

1. Interfaces

The first step is to write and compile the Java code for the service interface. The `Calculator` interface defines all of the remote features offered by the service:

```
public interface Calculator
          extends java.rmi.Remote {
    public long add(long a, long b)
        throws java.rmi.RemoteException;

    public long sub(long a, long b)
        throws java.rmi.RemoteException;

    public long mul(long a, long b)
        throws java.rmi.RemoteException;

    public long div(long a, long b)
        throws java.rmi.RemoteException;
}
```

Notice this interface extends `Remote`, and each method signature declares that it may throw a `RemoteException` object.

Copy this file to your directory and compile it with the Java compiler:

```
>javac Calculator.java
```

Implementation

Next, you write the implementation for the remote service. This is the `CalculatorImpl` class:

```
public class CalculatorImpl
    extends
      java.rmi.server.UnicastRemoteObject
    implements Calculator {
```

```
    // Implementations must have an
    //explicit constructor
    // in order to declare the
    //RemoteException exception
    public CalculatorImpl()
        throws java.rmi.RemoteException {
        super();
    }

    public long add(long a, long b)
        throws java.rmi.RemoteException {
        return a + b;
    }

    public long sub(long a, long b)
        throws java.rmi.RemoteException {
        return a - b;
    }

    public long mul(long a, long b)
        throws java.rmi.RemoteException {
        return a * b;
    }

    public long div(long a, long b)
        throws java.rmi.RemoteException {
        return a / b;
    }
}
```

Again, copy this code into your directory and compile it.

The implementation class uses `UnicastRemoteObject` to link into the RMI system. In the example the implementation class directly extends `UnicastRemoteObject`. This is not a requirement. A class that does not extend `UnicastRemoteObject` may use its `exportObject()` method to be linked into RMI.

When a class extends `UnicastRemoteObject`, it must provide a constructor that declares that it may throw a `RemoteException` object. When this constructor calls `super()`, it activates code in `UnicastRemoteObject` that performs the RMI linking and remote object initialization.

2. Stubs and Skeletons

You next use the RMI compiler, `rmic`, to generate the stub and skeleton files. The compiler runs on the remote service *implementation* class file.

```
>rmic CalculatorImpl
```

Try this in your directory. After you run `rmic` you should find the file `Calculator_Stub.class` and, if you are running the Java 2 SDK, `Calculator_Skel.class`.

Options for the JDK 1.1 version of the RMI compiler, `rmic`, are:

```
Usage: rmic <options> <class names>

where <options> includes:
  -keep  Do not delete intermediate
          generated source files
  -keepgenerated (same as "-keep")
  -g     Generate debugging info
  -depend   Recompile out-of-date
            files recursively
  -nowarn   Generate no warnings
  -verbose   Output messages about
             what the compiler is doing
  -classpath <path>     Specify where
                          to find input source
                          and class files
  -d <directory>   Specify where to
              place generated class files
  -J<runtime flag> Pass argument
               to the java interpreter
```

The Java 2 platform version of `rmic` add three new options:

```
  -v1.1  Create stubs/skeletons
             for JDK 1.1 stub
                 protocol version
  -vcompat  (default)
            Create stubs/skeletons compatible
                 with both JDK 1.1 and Java 2
                     stub protocol versions
  -v1.2  Create stubs for Java 2 stub protocol
                 version only
```

3. Host Server

Remote RMI services must be hosted in a server process. The class `CalculatorServer` is a very simple server that provides the bare essentials for hosting.

```
import java.rmi.Naming;

public class CalculatorServer {

  public CalculatorServer() {
    try {
      Calculator c = new CalculatorImpl();
      Naming.rebind("
      rmi://localhost:1099/
              CalculatorService", c);
    } catch (Exception e) {
      System.out.println("Trouble: " + e);
    }
  }
```

```
    public static void main(String args[]) {
      new CalculatorServer();
    }
  }
```

4. Client

The source code for the client follows:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class CalculatorClient {

    public static void main(String[] args) {
        try {
            Calculator c = (Calculator)
                          Naming.lookup(
                "rmi://remotehost
                       /CalculatorService");
            System.out.println( c.sub(4, 3) );
            System.out.println( c.add(4, 5) );
            System.out.println( c.mul(3, 6) );
            System.out.println( c.div(9, 3) );
        }
        catch (MalformedURLException murle) {
            System.out.println();
            System.out.println(
              "MalformedURLException");
            System.out.println(murle);
        }
        catch (RemoteException re) {
            System.out.println();
            System.out.println(
                          "RemoteException");
            System.out.println(re);
        }
        catch (NotBoundException nbe) {
            System.out.println();
            System.out.println(
                       "NotBoundException");
            System.out.println(nbe);
        }
        catch (
            java.lang.ArithmeticException
                                    ae) {
            System.out.println();
            System.out.println(
             "java.lang.ArithmeticException");
            System.out.println(ae);
        }
    }
```

```
}
```

5. **Running the RMI System**

   You are now ready to run the system! You need to start three consoles, one for the server, one for the client, and one for the RMIRegistry.

   Start with the Registry. You must be in the directory that contains the classes you have written. From there, enter the following:

```
rmiregistry
```

   If all goes well, the registry will start running and you can switch to the next console.

   In the second console start the server hosting the `CalculatorService`, and enter the following:

```
>java CalculatorServer
```

   It will start, load the implementation into memory and wait for a client connection.

   In the last console, start the client program.

```
>java CalculatorClient
```

   If all goes well you will see the following output:

```
1
9
18
3
```

That's it; you have created a working RMI system. Even though you ran the three consoles on the same computer, RMI uses your network stack and TCP/IP to communicate between the three separate JVMs. This is a full-fledged RMI system.

Parameters in RMI

You have seen that RMI supports method calls to remote objects. When these calls involve passing parameters or accepting a return value, how does RMI transfer these between JVMs? What semantics are used? Does RMI support pass-by-value or pass-by-reference? The answer depends on whether the parameters are primitive data types, objects, or remote objects.

Parameters in a Single JVM

First, review how parameters are passed 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.

Now you will see how RMI passes parameters and return values between remote JVMs.

Primitive Parameters

When a primitive data type is passed as a parameter to a remote method, the RMI system passes it by value. RMI 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

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. 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.

Remote Object Parameters

RMI introduces a third type of parameter to consider: remote objects. As you have seen, a client program can obtain a reference to a remote object through the RMI Registry program. There is another way in which 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) {
}
```

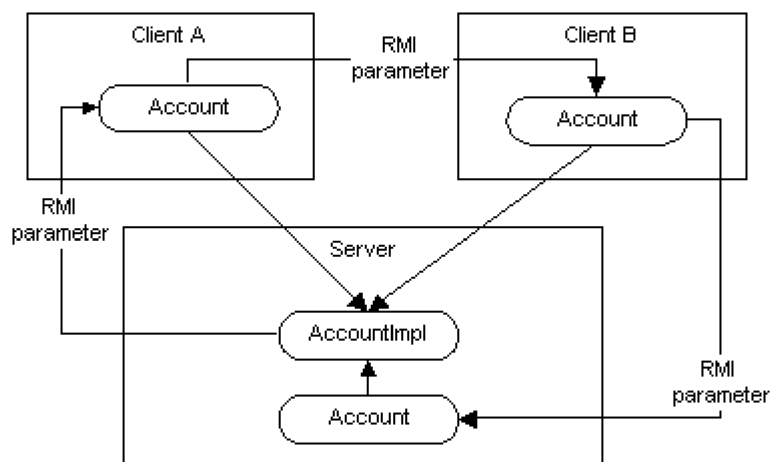
In the implementation of `getAccount()`, the method returns a (local) reference to the remote service.

```
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 diagram 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



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.

RMI Client-side Callbacks

In many architectures, a server may need to make a remote call to a client. Examples include progress feedback, time tick notifications, warnings of problems, etc.

To accomplish this, a client must also act as an RMI server. There is nothing really special about this as RMI works equally well between all computers. However, it may be impractical for a client to extend `java.rmi.server.UnicastRemoteObject`. In these cases, a remote object may prepare itself for remote use by calling the static method

```
UnicastRemoteObject.exportObject (<remote_object>)
```

Distributing and Installing RMI Software

RMI adds support for a Distributed Class model to the Java platform and extends Java technology's reach to multiple JVMs. It should not be a surprise that installing an RMI system is more involved than setting up a Java runtime on a single computer. In this section, you will learn about the issues related to installing and distributing an RMI based system.

For the purposes of this section, it is assumed that the overall process of designing a DC system has led you to the point where you must consider the allocation of processing to nodes. And you are trying to determine how to install the system onto each node.

Distributing RMI Classes

To run an RMI application, the supporting class files must be placed in locations that can be found by the server and the clients.

For the server, the following classes must be available to its class loader:

- Remote service interface definitions
- Remote service implementations
- Skeletons for the implementation classes (JDK 1.1 based servers only)
- Stubs for the implementation classes
- All other server classes

For the client, the following classes must be available to its class loader:

- Remote service interface definitions
- Stubs for the remote service implementation classes
- Server classes for objects used by the client (such as return values)

- All other client classes

Once you know which files must be on the different nodes, it is a simple task to make sure they are available to each JVM's class loader.

Automatic Distribution of Classes

The RMI designers extended the concept of class loading to include the loading of classes from FTP servers and HTTP servers. This is a powerful extension as it means that classes can be deployed in one, or only a few places, and all nodes in a RMI system will be able to get the proper class files to operate.

RMI supports this remote class loading through the `RMIClassLoader`. If a client or server is running an RMI system and it sees that it must load a class from a remote location, it calls on the `RMIClassLoader` to do this work.

The way RMI loads classes is controlled by a number of properties. These properties can be set when each JVM is run:

```
java [ -D<PropertyName>=<PropertyValue> ]+
<ClassFile>
```

The property `java.rmi.server.codebase` is used to specify a URL. This URL points to a `file:`, `ftp:`, or `http:` location that supplies classes for objects that are sent *from* this JVM. If a program running in a JVM sends an object to another JVM (as the return value from a method), that other JVM needs to load the class file for that object. When RMI sends the object via serialization of RMI embeds the URL specified by this parameter into the stream, alongside of the object.

Note: RMI does not send class files along with the serialized objects.

If the remote JVM needs to load a class file for an object, it looks for the embedded URL and contacts the server at that location for the file.

When the property `java.rmi.server.useCodebaseOnly` is set to `true`, then the JVM will load classes from either a location specified by the `CLASSPATH` environment variable or the URL specified in this property.

By using different combinations of the available system properties, a number of different RMI system configurations can be created.

Closed. All classes used by clients and the server must be located on the JVM and referenced by the `CLASSPATH` environment variable. No dynamic class loading is supported.

Server based. A client applet is loaded from the server's `CODEBASE` along with all supporting classes. This is similar to the way applets are loaded from the same HTTP server that supports the applet's web page.

Client dynamic. The primary classes are loaded by referencing the `CLASSPATH` environment variable of the JVM for the client. Supporting classes are loaded by the

`java.rmi.server.RMIClassLoader` from an HTTP or FTP server on the network at a location specified by the server.

Server-dynamic. The primary classes are loaded by referencing the `CLASSPATH` environment variable of the JVM for the server. Supporting classes are loaded by the `java.rmi.server.RMIClassLoader` from an HTTP or FTP server on the network at a location specified by the client.

Bootstrap client. In this configuration, *all* of the client code is loaded from an HTTP or FTP server across the network. The only code residing on the client machine is a small bootstrap loader.

Bootstrap server. In this configuration, *all* of the server code is loaded from an HTTP or FTP server located on the network. The only code residing on the server machine is a small bootstrap loader.

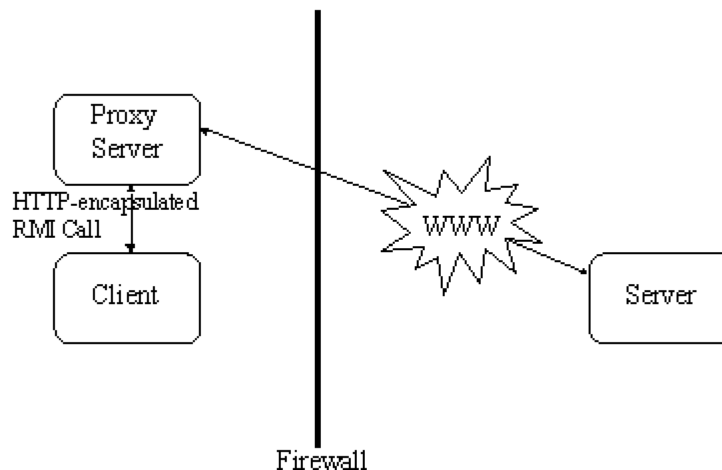
The exercise for this section involves creating a bootstrap client configuration. Please follow the directions carefully as different files need to be placed and compiled within separate directories.

Firewall Issues

Firewalls are inevitably encountered by any networked enterprise application that has to operate beyond the sheltering confines of an Intranet. Typically, firewalls block all network traffic, with the exception of those intended for certain "well-known" ports.

Since the RMI transport layer opens dynamic socket connections between the client and the server to facilitate communication, the JRMP traffic is typically blocked by most firewall implementations. But luckily, the RMI designers had anticipated this problem, and a solution is provided by the RMI transport layer itself. To get across firewalls, RMI makes use of HTTP tunneling by encapsulating the RMI calls within an HTTP POST request.

Now, examine how HTTP tunneling of RMI traffic works by taking a closer look at the possible scenarios: the RMI client, the server, or both can be operating from behind a firewall. The following diagram shows the scenario where an RMI client located behind a firewall communicates with an external server.

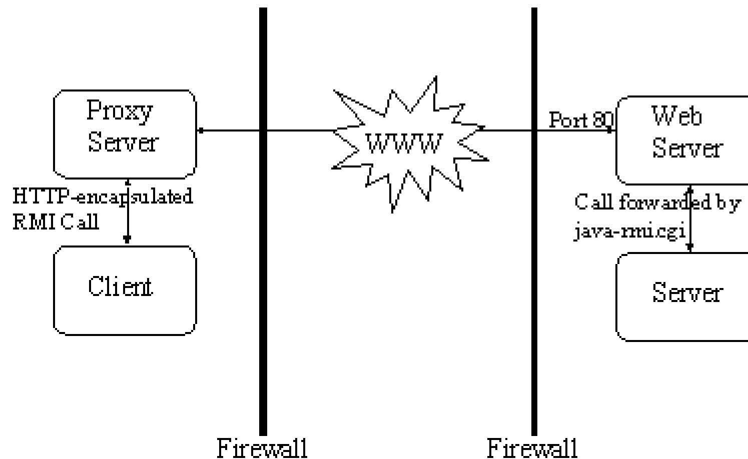


In the above scenario, when the transport layer tries to establish a connection with the server, it is blocked by the firewall. When this happens, the RMI transport layer automatically retries by encapsulating the JRMP call data within an HTTP POST request. The HTTP POST header for the call is in the form:

`http://hostname:port`

If a client is behind a firewall, it is important that you also set the system property `http.proxyHost` appropriately. Since almost all firewalls recognize the HTTP protocol, the specified proxy server should be able to forward the call directly to the port on which the remote server is listening on the outside. Once the HTTP-encapsulated JRMP data is received at the server, it is automatically decoded and dispatched by the RMI transport layer. The reply is then sent back to client as HTTP-encapsulated data.

The following diagram shows the scenario when both the RMI client and server are behind firewalls, or when the client proxy server can forward data only to the well-known HTTP port 80 at the server.



In this case, the RMI transport layer uses one additional level of indirection! This is because the client can no longer send the HTTP-encapsulated JRMP calls to arbitrary ports as the server is also behind a firewall. Instead, the RMI transport layer places JRMP call inside the HTTP packets and send those packets to port 80 of the server. The HTTP POST header is now in the form

`http://hostname:80/cgi-bin/java-rmi?forward=<port>`

This causes the execution of the CGI script, `java-rmi.cgi`, which in turn invokes a local JVM, unbundles the HTTP packet, and forwards the call to the server process on the designated port. RMI JRMP-based replies from the server are sent back as HTTP REPLY packets to the originating client port where RMI again unbundles the information and sends it to the appropriate RMI stub.

Of course, for this to work, the `java-rmi.cgi script`, which is included within the standard JDK 1.1 or Java 2 platform distribution, must be preconfigured with the path of the Java interpreter and located within the web server's `cgi-bin` directory. It is also equally

important for the RMI server to specify the host's fully-qualified domain name via a system property upon startup to avoid any DNS resolution problems, as:

```
java.rmi.server.hostname=host.domain.com
```

It should be noted that notwithstanding the built-in mechanism for overcoming firewalls, RMI suffers a significant performance degradation imposed by HTTP tunneling. There are other disadvantages to using HTTP tunneling too. For instance, your RMI application will no longer be able to multiplex JRMP calls on a single connection, since it would now follow a discrete request/response protocol. Additionally, using the `java-rmi.cgi` script exposes a fairly large security loophole on your server machine, as now, the script can redirect any incoming request to any port, completely bypassing your firewalling mechanism. Developers should also note that using HTTP tunneling precludes RMI applications from using callbacks, which in itself could be a major design constraint. Consequently, if a client detects a firewall, it can always disable the default HTTP tunneling feature by setting the property:

```
java.rmi.server.disableHttp=true
```

Distributed Garbage Collection

One of the joys of programming for the Java platform is not worrying about memory allocation. The JVM has an automatic garbage collector that will reclaim the memory from any object that has been discarded by the running program.

One of the design objectives for RMI was seamless integration into the Java programming language, which includes garbage collection. Designing an efficient single-machine garbage collector is hard; designing a distributed garbage collector is very hard.

The RMI system provides a reference counting distributed garbage collection algorithm based on Modula-3's Network Objects. This system works by having the server keep track of which clients have requested access to remote objects running on the server. When a reference is made, the server marks the object as "dirty" and when a client drops the reference, it is marked as being "clean."

The interface to the DGC (distributed garbage collector) is hidden in the stubs and skeletons layer. However, a remote object can implement the `java.rmi.server.Unreferenced` interface and get a notification via the `unreferenced` method when there are no longer any clients holding a live reference.

In addition to the reference counting mechanism, a live client reference has a lease with a specified time. If a client does not refresh the connection to the remote object before the lease term expires, the reference is considered to be dead and the remote object may be garbage collected. The lease time is controlled by the system property `java.rmi.dgc.leaseValue`. The value is in milliseconds and defaults to 10 minutes.

Because of these garbage collection semantics, a client must be prepared to deal with remote objects that have "disappeared."

Serializing Remote Objects

When designing a system using RMI, there are times when you would like to have the flexibility to control where a remote object runs. Today, when a remote object is brought to life on a particular JVM, it will remain on that JVM. You cannot "send" the remote object to another machine for execution at a new location. RMI makes it difficult to have the option of running a service locally or remotely.

The very reason RMI makes it easy to build some distributed application can make it difficult to move objects between JVMs. When you declare that an object implements the `java.rmi.Remote` interface, RMI will prevent it from being serialized and sent between JVMs as a parameter. Instead of sending the implementation class for a `java.rmi.Remote` interface, RMI substitutes the stub class. Because this substitution occurs in the RMI internal code, one cannot intercept this operation.

There are two different ways to solve this problem. The first involves manually serializing the remote object and sending it to the other JVM. To do this, there are two strategies. The first strategy is to create an `ObjectInputStream` and `ObjectOutputStream` connection between the two JVMs. With this, you can explicitly write the remote object to the stream. The second way is to serialize the object into a `byte` array and send the `byte` array as the return value to an RMI method call. Both of these techniques require that you code at a level below RMI and this can lead to extra coding and maintenance complications.

In a second strategy, you can use a delegation pattern. In this pattern, you place the core functionality into a class that:

- Does *not* implement `java.rmi.Remote`
- *Does* implement `java.io.Serializable`

Then you build a remote interface that declares remote access to the functionality. When you create an implementation of the remote interface, instead of reimplementing the functionality, you allow the remote implementation to defer, or delegate, to an instance of the local version.

Now look at the building blocks of this pattern. Note that this is a very simple example. A real-world example would have a significant number of local fields and methods.

```
// Place  functionality in a local object
public class LocalModel
implements java.io.Serializable
{
  public String getVersionNumber()
  {
    return "Version 1.0";
  }
}
```

Next, you declare an `java.rmi.Remote` interface that defines the same functionality:

```
interface RemoteModelRef
    extends java.rmi.Remote
{
```

```
  String getVersionNumber()
    throws java.rmi.RemoteException;
}
```

The implementation of the remote service accepts a reference to the `LocalModel` and delegates the real work to that object:

```
public class RemoteModelImpl
    extends
      java.rmi.server.UnicastRemoteObject
    implements RemoteModelRef
  {
  LocalModel lm;

  public RemoteModelImpl (LocalModel lm)
      throws java.rmi.RemoteException
  {
    super();
    this.lm = lm;
  }

  // Delegate to the local
  //model implementation
  public String getVersionNumber()
      throws java.rmi.RemoteException
  {
    return lm.getVersionNumber();
  }
}
```

Finally, you define a remote service that provides access to clients. This is done with a `java.rmi.Remote` interface and an implementation:

```
interface RemoteModelMgr extends java.rmi.Remote
{
  RemoteModelRef getRemoteModelRef()
    throws java.rmi.RemoteException;

  LocalModel     getLocalModel()
    throws java.rmi.RemoteException;
}
public class RemoteModelMgrImpl
    extends
     java.rmi.server.UnicastRemoteObject
    implements RemoteModelMgr
  {
  LocalModel lm;
  RemoteModelImpl rmImpl;

  public RemoteModelMgrImpl()
      throws java.rmi.RemoteException
  {
    super();
  }
```

```
  public RemoteModelRef getRemoteModelRef()
      throws java.rmi.RemoteException
    {
    // Lazy instantiation of delgatee
    if (null == lm)
    {
      lm = new LocalModel();
    }

    // Lazy instantiation of
    //Remote Interface Wrapper
    if (null == rmImpl)
    {
      rmImpl = new RemoteModelImpl (lm);
    }

    return ((RemoteModelRef) rmImpl);
  }

  public LocalModel getLocalModel()
      throws java.rmi.RemoteException
  {
    // Return a reference to the
    //same LocalModel
    // that exists as the delagetee
    //of the RMI remote
    // object wrapper

    // Lazy instantiation of delgatee
    if (null == lm)
    {
      lm = new LocalModel();
    }

    return lm;
  }
}
```