Implementation of RMI

Aim

implement RMI system

Theoretical background

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[1] (JVM) to a new model of how classes and objects would work in a distributed (multiple JVM) computing environment.

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

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.

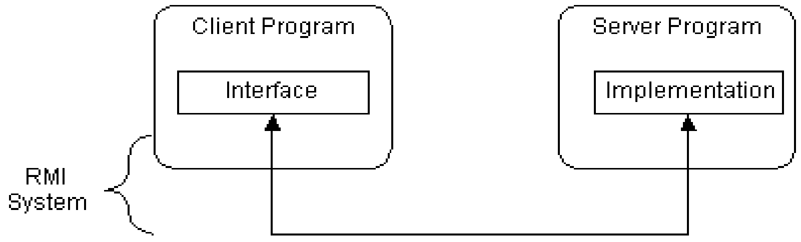
Interfaces: The Heart of RMI

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.

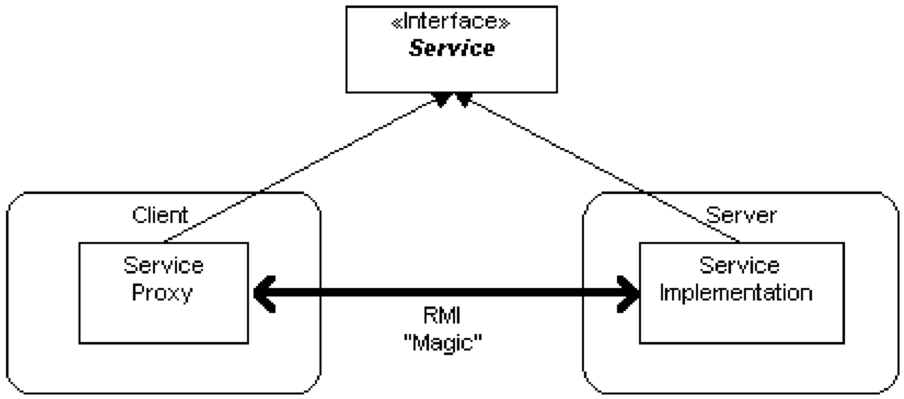
Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that *interfaces define behavior* and *classes define implementation*.

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



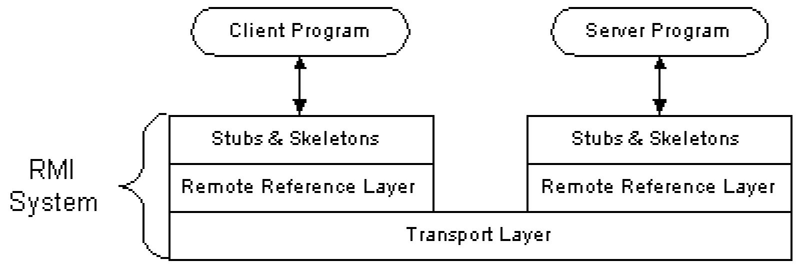
remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

RMI Architecture Layers

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

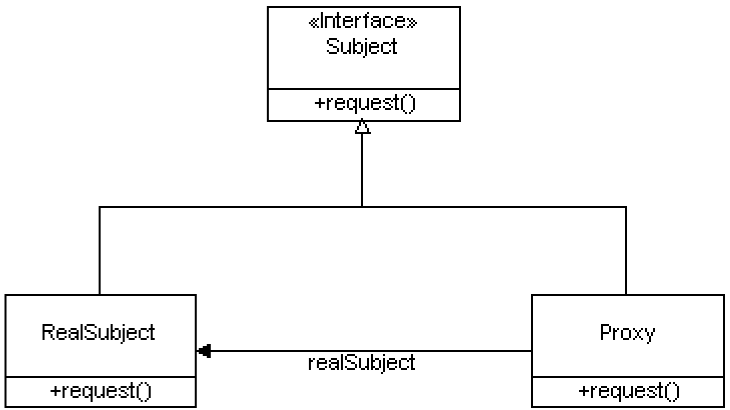
In RMI's use of the Proxy pattern, the stub class plays the role of the proxy, and the remote service implementation class plays the role of the `RealSubject`.

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 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.) Some other changes with the Java 2 SDK are that RMI service interfaces are not required to extend from `java.rmi.Remote` and their service methods do not necessarily throw `RemoteException`.

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.

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.

(*Well-known* meaning everyone in an organization knowing what it is).

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. (Details about how to use HTTP and FTP servers as class file providers will be covered in the section on Distributing and Installing RMI Software)

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 sub(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
```

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

1. 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.

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`.

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();
               }
      }
```

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://localhost
                        /CalculatorService");
            System.out.println( c.sub(4, 3) );

        }
        catch (MalformedURLException murle) {
            System.out.println();
            System.out.println("MalformedURLException");
             }
        catch (RemoteException re) {
            System.out.println();
            System.out.println( "RemoteException");
            }
        catch (NotBoundException nbe) {
            System.out.println();
            System.out.println( "NotBoundException");
         }
        catch (java.lang.ArithmeticException ae) {
            System.out.println();
            System.out.println("java.lang.ArithmeticException");
        }
    }
}
```

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
```

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