Java Remote Method Invocation

Objectives

+ Understand the fundamental purpose of RMI;
+ Understand how RMI works;
+ Be able to implement an RMI client/server application involving .class files that are available locally;

1. Introduction

This section is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system, examining, in particular, the remote invocation paradigms.

- Request-reply protocols represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing. In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation, and also provide direct support for RPC and RMI, discussed below.
- The earliest and perhaps the best-known example of a more programmer-friendly model was the extension of the conventional procedure call model to distributed systems (the remote procedure call, or RPC, model), which allows client programs to call procedures transparently in server programs running in separate processes and generally in different computers from the client.
- In the 1990s, the object-based programming model was extended to allow objects in different processes to communicate with one another by means of remote method invocation (RMI). RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

Note that we use the term ‘RMI’ to refer to remote method invocation in a generic way – this should not be confused with particular examples of remote method invocation such as Java RMI.

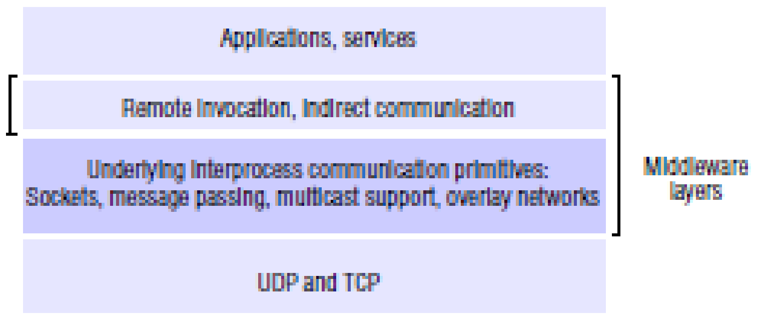
Applications, services
Remote invocation, indirect communication
Underlying interprocess communication primitives: Sockets, message passing, multicast support, overlay networks
UDP and TCP
Middleware layers

Figure 1 Middleware layers

Figure 1 continues our study of middleware concepts by focusing on the layer above interprocess communication.

2. Java RMI

– With all our method calls so far, the objects upon which such methods have been invoked have been local. However, in a distributed environment, it is often desirable to be able to invoke methods on remote objects (i.e. on objects located on other systems). RMI (Remote Method Invocation) provides a platform – independent means of doing just this.

- Under RMI, the networking details required by explicit programming streams and sockets disappear; and the fact that an object is located remotely is almost transparent to the Java programmer.
- Once a reference to the remote object has been obtained, the methods of that object may be invoked in exactly the same way as those of local objects. In the background, of course, RMI will be making use of byte streams to transfer data and method invocations, but all this is handled automatically by the RMI infrastructure.
- RMI has been a core component of Java from the earliest release of the language, but has undergone some evolutionary changes since its original specification.

3. The Basic RMI Process

- Earlier on, we referred to obtaining a reference to a remote object, this was really a simplification of what actually happens.
- The server program that has control of the remote object registers an interface with a naming service, thereby making this interface accessible by client programs. The interface contains the signatures for those methods of the object that the server wishes to make publicly available.

  A client program can then use the same naming service to obtain a reference to this interface in form of what is called a **stub**. This stub is effectively a local surrogate (a 'stand-in' or place holder) for the remote object. On the remote system, there will be another surrogate called **skeleton**.

- When the client program invokes a method of the remote object, it appears as though the method is being invoked directly on the object. What is actually happening, however, is that an equivalent method is being called in the stub. The stub then forwards the call and any parameters to the skeleton on the remote machine. Only primitive types and those reference types that implement the Serializable interface may be used as parameters (the serializing of these parameters is called **marshalling**).
- Upon receipt of the byte stream, the skeleton converts this stream into the original method call and associated parameters (the deserialization of parameters being referred to as **unmarshalling**). Finally, the skeleton calls the implementation of the method on the server. The stages of this process are shown diagrammatically in Figure 2.

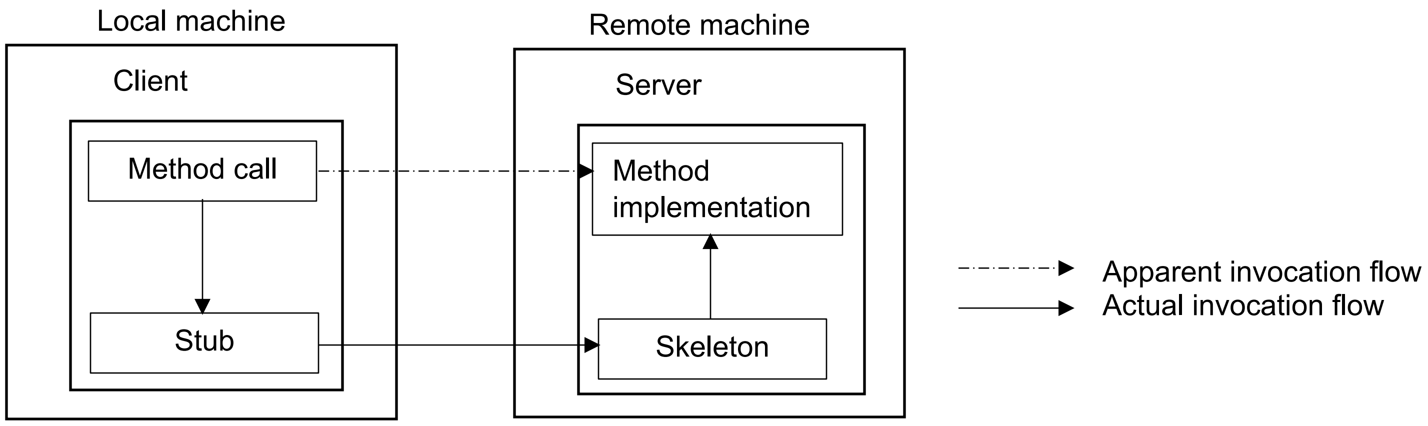

Figure 2: Using RMI to invoke a method of a remote object

3. Implementation Details

– The packages used in the implementation of an RMI client – server application are:

  (i.) java.rmi,
  (ii.) java.rmi.server, and
  (iii.) java.rmi.registry;

  though only the first two need to be used explicitly.

  Of course, if the application class already have a superclass (other than Object), extending Thread is not an option, since Java does not support multiple inheritance.

– The basic steps are listed below:
  1. Create the interface.
  2. Define a class that implements this interface.
  3. Create the server process.
  4. Create the client process.

– We will use a very simple application as an example to illustrate and appreciate the four steps listed above. This example application will simply display a greeting message to any client that uses the appropriate interface registered with the naming service to invoke the associated method implementation on the server.

I. Create the interface

– This interface should import the package java.rmi and must extend the interface Remote, which (like Serializable) is a 'tagging' interface that contains no methods. The interface definition for this example must specify the signature for the method (in this case) getGreeting, which is to be made available to clients. This method must declare that it throws a RemoteException. The contents of this file are shown below:

Listing 1: The Interface definition

```
import java.rmi.*;

public interface Hello extends Remote {

   public String getGreeting() throws RemoteException;

}
```

II. Define a class that implements this interface

– The implementation class should import two Java packages java.rmi and java.rmi.server. The implementation class must extend class RemoteObject or one of RemoteObject's subclasses. In practice, most implementations extend the subclass UnicastRemoteObject, since this class supports point – to – point communication using TCP streams. The implementation class must also implement our interface Hello, of course, by providing an executable body for the single interface method getGreeting.

– In addition, we must provide a constructor for our implementation object (even if we simply give this constructor an empty body, as below). Like the method(s) declared in the interface, this constructor must declare that it throws a RemoteException.

– Finally, we shall adopt the common convention of appending the suffix Impl onto the name of our interface to form the name of the implementation class.

```
Listing 2: The class that implements the interface

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

public class HelloImpl extends UnicastRemoteObject implements Hello{

      public HelloImpl() throws RemoteException{

            //no action needed here.
      }

      public String getGreeting() throws RemoteException{
            return (“ Hello there! ”);
      }
}
```

III. Create the server process

– The server creates object(s) of the above implementation class and registers them with a naming service called the registry. It does this by using the static method rebind of a class Naming (from package java.rmi). This method has two arguments:
   (i.) a String that hold the name of the remote object as a URL with protocol rmi;
   (ii.) a reference to the remote object (as an argument of type Remote).

   + The method establishes a connection between the object’s name and its reference clients will then be able to use the remote object’s name to retrieve a reference to that object via the registry.
   + The URL string, as well as specifying a protocol of rmi and name for the object, specifies the name of the remote object’s host machine. For simplicity’s sake, we shall use localhost (which is what RMI assumes by default anyway). The default port for RMI is 1099, though we can change this to any port if we wish.

– The code for our server process is shown below and contains just one method: main. To cater for the various types of exception that may be generated, this method declares that it throws Exception.

```
Listing 3: This implements the server process

import java.rmi.*;

public class HelloServer{

      private static final String HOST = “localhost”;

      public static void main(String [] args) throws Exception{
            //create a reference to an implementation object
            HelloImpl temp = new HelloImpl();

            //create the string URL holding the object’s name
            String rmiObjectName = “rmi:” + HOST + “/Hello”;

            //could omit host name here, since localhost would
            //be assumed default
```

```
                //Bind the object reference to the name
                Naming.rebind(rmiObjectName, temp);

                //Display a message so that we know the process
                //has been completed
                System.out.println("Binding complete ... \n");
        }

                //no action needed here.
        }

        public String getGreeting() throws RemoteException{
                return (" Hello there! ");
        }
}
```

IV. Create the client process

- The client obtains a reference to the remote object from the registry. It does this by using method lookup of class Naming, supplying as an argument to this method the same URL that the server did when binding the object reference to the object's name in the registry.
  + Since lookup returns a Remote reference, this reference must be typecast into an Hello reference (not an HelloImpl reference!). Once the Hello reference has been obtained, it can be used to call the solitary method that was made available in the interface.

```
Listing 4: This implements the client process

import java.rmi.*;

public class HelloClient{

        private static final String HOST = "localhost";

        public static void main(String [] args) {

                try{
                        //Obtain a reference to the object from the registry
                        //and typecast it into the appropriate type ...
                        Hello greeting = (Hello) Naming.lookup("rmi:" + HOST + "/Hello");

                        //use the above reference to invoke the
                        //remote object's method ...
                        System.out.println("Message received: " + greeting.getGreeting());

                }
                catch(ConnectionException conEx){
                        System.out.println("Unable to connect to server!");
                        System.exit(1);
                }
                catch(Exception ex){
                        ex.printStackTrace();
                        System.exit(1);
                }
        }

}
```

5. Compilation and Execution

– There are several steps that need to be carried out, as described below:

I. Compile all files with javac
This is straightforward

```
javac Hello.java
javac HelloImpl.java
javac HelloServer.java
javac HelloClient.java
```

II. Start the RMI registry
Type the following command:

```
rmiregistry
```

When this is executed, the only indication that anything has happened is a change in the command window's tile.

III. Open a new command prompt window and run the server
From the new window, invoke the Java interpreter

```
java HelloServer
```

IV. Open a third command prompt window and run the client
Again, invoke the Java interpreter:

```
Java HelloClient
```

Since the server process and the RMI registry will continue to run indefinitely after the client process has finished, they will need to be closed down by entering Ctrl + C in each of their windows.