RPC/RMI

Overview

- **Idea**
  - A procedure (RPC)/method (RMI) is executed on a (possibly) different process (or) than the one in which it was invoked

- **Benefits**
  - Allows a normal programming model in a distributed setting
  - Hides the details of sending and receiving messages, marshaling and unmarshaling, finding the server, security policies, etc.
  - Simplifies the programmers work
RPC/RMI

Overview

- RMI applications often comprise two separate programs: server and client.
- A typical **server** program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
- A typical **client** program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.
RMI

Architecture

**Stub** A stub for a remote object acts as a client’s local representative or proxy for the remote object. The stub hides the serialization of parameters and the network-level communication. When a stub’s method is invoked, it does the following:

- initiates a connection with the remote JVM containing the remote object
- marshals (writes and transmits) the parameters to the remote JVM
- waits for the result of the method invocation
- unmarshals (reads) the return value or exception returned
- returns the value to the caller

**Skeleton** In the remote JVM, each remote object may have a corresponding skeleton. When a skeleton receives an incoming method invocation it does the following:

- Unmarshals (reads) the parameters for the remote method
- Invokes the method on the actual remote object implementation
- marshals (writes and transmits) the result (return value or exception) to the caller
RMI

Architecture

- **Remote Reference Layer (RRL)** Interpret and manage references made from clients to the remote service objects and vice versa

- **Transport Layer** Is based on TCP/IP connections between machines in a network. Provides basic connectivity, as well as some firewall penetration strategies
RMI
Principles

- As both, the server and the clients run JVM, the marshaling is done using the Java serializer. As a result, the passed object has to implement the `Serializable` interface.
- The definition of an objects class can be downloaded by the receiver (from the sender) if the class is not defined in the receiver’s JVM
  - All of the types and behaviors of an object can be transmitted to a remote JVM
  - New types and behaviors can be introduced into a remote JVM, thus dynamically extending the behavior of an application
- To obtain references to remote objects, there exists a registry, called the RMI Registry, which is a remote object that maps names to remote objects
RMI

Normal Execution Pattern
Development Principles

Using RMI to develop a distributed application involves these general steps:

1. Write the code
   1. Write the remote interface
   2. Write the server code
   3. Write the client code

2. Compile the code

3. Make the classes network accessible

4. Start the RMI server and the application
Example

Overview

We will now develop a client/server system in which the client submit a computationally expensive task to the server. The server computes the task and returns the answer to the client.
First, we develop the remote interface. The remote interface has to be available to both, server and client as either source files or jar file. The remote interface is consist of two interfaces. **Compute** which is the actual remote object and **Task** which will be submitted to the server by the client. As a remote object, **Compute** has to extend the `java.rmi.Remote` class.
Example
 Compute.java

```java
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

The server implements Compute and publish the implementation to the **RMI Registry**. Then the client is able to receive a reference to the Compute object from the registry and invokes the remote method `executeTask`
package compute;

different interface Task<T> {
   T execute();
}

Different kinds of tasks can be run by a Compute object as long as they are implementations of the Task type. The classes that implement this interface can contain any data needed for the computation of the task and any other methods needed for the computation.
A class that implements a remote interface needs to provide an implementation for each remote method in the interface.

The server program needs to create the remote objects and export them to the RMI runtime, making them available to receive incoming remote invocations.

A security manager must be created and installed so the RMI runtime knows what can be executed on the server.

Remote objects are passed by reference from a client.

- Passing a remote object by reference means that any changes made to the state of the object by remote method invocations are reflected in the original remote object. When a remote object is passed, only those interfaces that are remote interfaces are available to the receiver. Any methods defined in the implementation class or defined in non-remote interfaces implemented by the class are not available to that receiver.

Other parameters that are not remote objects are passed by value.
package server;
import compute.Compute;
import compute.Task;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Compute {
    public <T> T executeTask(Task<T> t) throws RemoteException {
        return t.execute();
    }
    public Server() {
        super();
    }
    ...
}
... public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new Server();
        Compute skeleton =
            (Compute) UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, skeleton);
        System.out.println("server.Server bound");
    } catch (Exception e) {
        System.err.println("server.Server exception: ");
        e.printStackTrace();
    }
}
}
Example

Pi.java

The client wishes to compute a computationally expensive task (computing \( \pi \) to a specified number of decimal places). \textbf{Pi} has to implement the \textbf{Task} interface as well as the \textbf{Serializable} interface.

```java
package client;
import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {
    private static final long serialVersionUID = 227L;
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);
    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;
    private final int digits;
    public Pi(int digits) {
        this.digits = digits;
    }
    public BigDecimal execute() {
        return computePi(digits);
    }
    ...
```

Dolev Adas (Revised by Yehonatan Buchnik)
public static BigDecimal computePi(int digits) {
    int scale = digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    BigDecimal arctan1_239 = arctan(239, scale);
    BigDecimal pi = arctan1_5.multiply(FOUR).subtract(arctan1_239).multiply(FOUR);
    return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
}
public static BigDecimal arctan(int inverseX, int scale) {
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 = BigDecimal.valueOf(inverseX * inverseX);
    numer = BigDecimal.ONE.divide(invX, scale, roundingMode);
    result = numer;
    int i = 1;
    do {
        numer = numer.divide(invX2, scale, roundingMode);
        int denom = 2 * i + 1;
        term = numer.divide(BigDecimal.valueOf(denom), scale, roundingMode);
        if ((i % 2) != 0) {
            result = result.subtract(term);
        } else {
            result = result.add(term);
        }
        i++;
    } while (term.compareTo(BigDecimal.ZERO) != 0);
    return result;
}
Example

Client

The client receives a reference to the remote object and invokes the remote method with the 
**Pi** object (which is an implementation of **Task**). Although it looks like an ordinary method
invocation the computation is actually done by the server.

The arguments for the client program are the host IP and the required accuracy of the
computation.
package client;
import compute.Compute;
import java.math.BigDecimal;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute stub = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = stub.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
Example

Compilation

Compile the remote interface files:

```
javac compute/Compute.java compute/Task.java
```

Compile the server’s source file:

```
javac server/Server.java
```

Finally, compile the client files:

```
javac client/Client.java client/Pi.java
```
Example

Run the system

First we have to create a file that specifies the server and the client security properties:

Create `server.policy` file:

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

Do the same for `client.policy`:

```java
grant {
  permission java.security.AllPermission;
};
```
Run the system

Create the following hierarchy for the server (in order to simplify the system we place client/Pi.class also with the server, although it doesn’t have to)

```
+---server
    +---Server.class
+---compute
    +---Task.class
    +---Compute.class
+---client
    +---Pi.class
+---server.policy
```
Example

Run the system

Create the following hierarchy for the client

```plaintext
+---client
   +---Pi.class
   +---Client.class
+---compute
   +---Task.class
   +---Compute.class
+---client.policy
```
Example

Run RMI Registry

Run the **RMI Registry** [For Windows] from within the server’s directory

```bash
rmiregistry &
[start rmiregistry]
```
Example

Run Server

Run from within the server’s directory

```
java -cp .
-Djavax.rmi.server.hostname=127.0.0.1
-Djavax.security.policy=server.policy
server.Server
```
Example

Run Client

Run from within the client’s directory

```
java -cp .
-Djava.security.policy=client.policy
client.Client localhost 45
```
Conclusion

For more details see documentation