CS502052: Enterprise Systems Development Concepts

Lab 3: Java - Remote Method Invocation (Part 1)

I. Introduction

The **RMI** *(Remote Method Invocation)* is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

II. Java RMI

It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM. RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package `java.rmi`.

1. Architecture of an RMI Application

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.

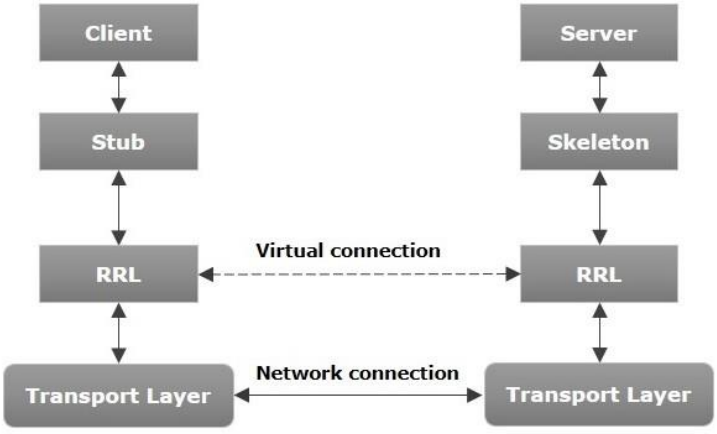


Figure 1 RMI architecture

Now, let's discuss the components of this architecture.

- **Transport Layer** – This layer connects the client and the server. It manages the existing connection and sets up new connections.
- **Stub** –A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.

- **Skeleton** – This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.
- **RRL** *(Remote Reference Layer)* – It is the layer which manages the references made by the client to the remote object.

2. Working of an RMI Application

The following points summarize how an RMI application works:

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

3. Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

4. RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the **RMIregistry** (using ***bind()*** or ***reBind()*** methods). These are registered using a unique name known as **bind name**.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using ***lookup()*** method).

The following illustration explains the entire process:

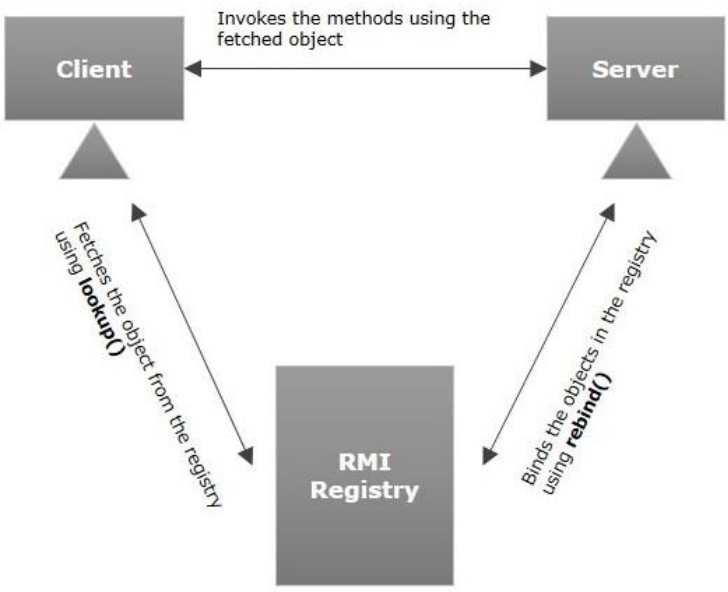


Figure 2 RMI Registry

5. Goals of RMI

Following are the goals of RMI:

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

III. Java RMI Programming

To write an RMI Java application, you would have to follow the steps given below:

- Define the **remote interface**;
- Develop the **implementation class** *(remote object)*;
- Develop the **server program**;
- Develop the **client program**;
- **Compile** the application;
- **Execute** the application.

1. Defining the Remote Interface

A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface.

To create a ***remote interface***:

- Create an interface that extends the predefined interface ***Remote*** which belongs to the package.

- Declare all the business methods that can be invoked by the client in this interface.
- Since there is a chance of network issues during remote calls, an exception named **RemoteException** may occur; throw it.

Following is an example of a remote interface. Here we have defined an interface with the name **Hello** and it has a method called *printMsg()*.

```
package edu.tdt.rmiinterface;

import java.rmi.*;

/**
public interface Hello extends Remote {

    public void printMsg(String name) throws RemoteException;

}
```

Figure 3 Remote Interface

2. Developing the Implementation Class (Remote Object)

We need to implement the remote interface created in the earlier step. We can write an implementation class separately or we can directly make the server program implement this interface.

To develop an ***implementation class***:

- Implement the interface created in the previous step.
- Provide implementation to all the abstract methods of the remote interface.

Following is an implementation class. Here, we have created a class named **ImplHello** and implemented the interface **Hello** created in the previous step, and provided ***body*** for this method which prints a message.

```
package edu.tdt.rmiinterface;

import java.rmi.RemoteException;

/**
public class ImplHello implements Hello {

    @Override
    public void printMsg(String name) throws RemoteException
    {
        System.out.println(name + " is trying to contact!");
    }

}
```

Figure 4 Implementation Class

3. Developing the Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMIregistry**.

To develop a ***server program***:

- Create a client class from where you want to invoke the remote object.
- **Create a remote object** by instantiating the implementation class as shown below.

- Export the remote object using the method ***exportObject()*** of the class named **UnicastRemoteObject** which belongs to the package **java.rmi.server**.
- Get the RMI registry using the ***getRegistry(port)*** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**. In case you are working on your local machine, you can omit the port parameter. However, in most case, you need to manually define the ***port***, to avoid port conflict.
- Bind the remote object created to the registry using the ***rebind()*** method of the class named **Registry**. To this method, pass a string representing the bind name and the object exported, as parameters.

Following is an example of an RMI server program.

```
public class Server extends ImplHello {

    public Server()
    {

    }

    public static void main(String[] args)
    {
        try
        {
            // Parsing the argument
            int index = 0;
            int port = Integer.parseInt(args[index++]);

            // Instantiating the implementation class
            ImplHello obj = new ImplHello();

            // Exporting the object of implementation class
            // (here we are exporting the remote object to the skeleton)
            Hello skeleton = (Hello) UnicastRemoteObject.exportObject(obj, 0);

            // Binding the remote object (stub) in the registry
            Registry registry = LocateRegistry.getRegistry(port);
            registry.rebind("Hello", skeleton);

            System.err.println("Server ready");

        } catch(Exception e)
        {
            System.err.println(e.toString());
        }
    }

}
```

Figure 5 Server program

4. Developing the Client Program

Write a client program in it, fetch the remote object and invoke the required method using this object.

To develop a client program:

- Create a client class from where your intended to invoke the remote object.
- Get the RMI registry using the ***getRegistry(host, port)*** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**. If you are working on your local machine, the host is *localhost*. The port value in Client must be the same as Server.
- Fetch the object from the registry using the method ***lookup()*** of the class **Registry** which belongs to the package **java.rmi.registry**.

- To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.
- The ***lookup()*** returns an object of type remote, down cast it to the type **Hello**.
- Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

```
public class Client {

    public Client()
    {

    }

    public static void main(String[] args)
    {
        try
        {
            // Parse the input arguments
            if (args.length != 3)
            {
                System.err.println("usage: java Client host port input");
                System.exit(1);
            }

            int index = 0;
            String host = args[index++];
            int port = Integer.parseInt(args[index++]);
            String input = args[index++];

            // Getting the registry
            Registry registry = LocateRegistry.getRegistry(host, port);

            // Looking up the registry for the remote object
            Hello stub = (Hello) registry.lookup("Hello");

            // Calling the remote method using the obtained object
            stub.printMsg(input);

            System.err.println("Remote method invoked");

        } catch(Exception e)
        {
            System.err.println(e.toString());
        }
    }
}
```

Figure 6 Client program

5. Compiling the Application

To compile and run the project, you should do the steps below.

Build the Java project

To build the Java project from *.java* to *.class* files, in NetBeans IDE, open menu *Run* → *Clean and Build Project*, or press the shortcut *Shift + F11*.

Start the Server

- Navigate to the project directory and open the *"build/classes"* folder.
- Open the Command Prompt windows in *"build/classes"*, the directory may look like this:

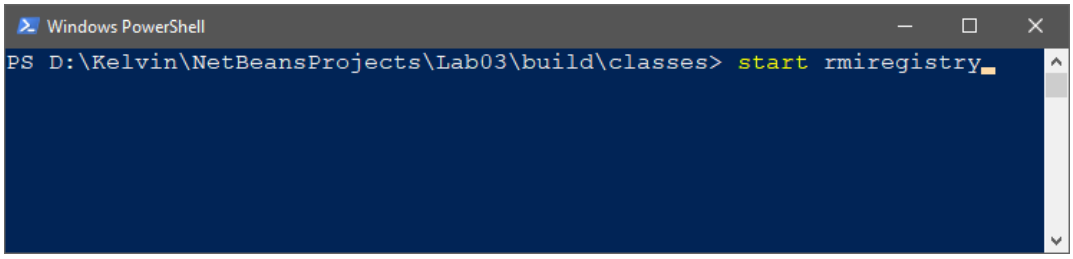


Figure 7 Start RMI Registry

- Start the RMI Registry by executing the command **`start rmiregistry 1100`**. In case MacOS, executing the command **`rmiregistry 1100`** in Terminal.
- This will start an RMI Registry on a separate window/terminal, and don't close this.

Run the Server class

- Open a new Command Prompt/Terminal window at *"build/classes"* path.
- Run the *Server.class* by executing: **`java edu.tdt.rmiserver.Server 1100`**.

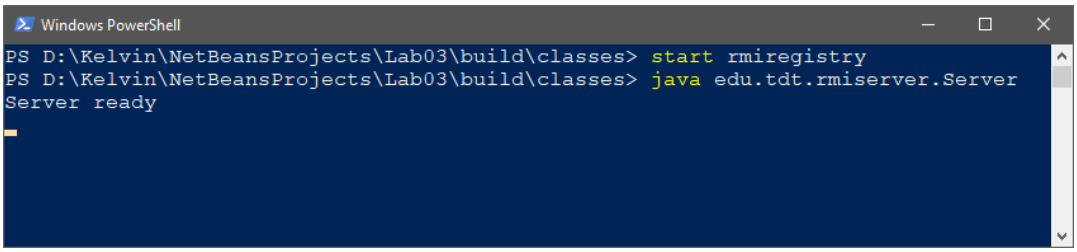
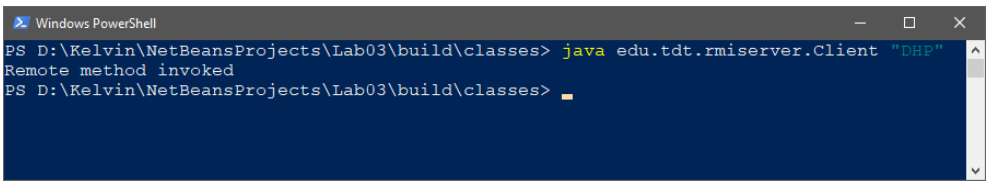


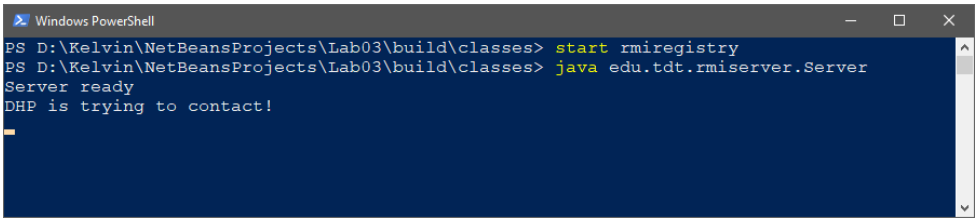
Figure 8 Executing Server.class

Run the Client class

- Open another Command Prompt/Terminal window in *"build/classes"*. (*)
- Run the *Client.class* by executing: **`java edu.tdt.rmiserver.Client localhost 1100 "DHP"`**.



- After successfully executing the above command, in the Command Prompt/Terminal of **Server** will output a new line, as follows.



Verification

Let's compare the two windows, the *Server* and the *Client*, to see how the RMI works.

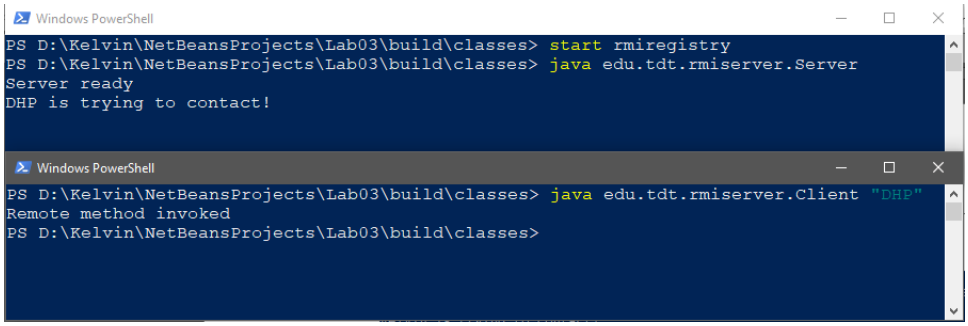


Figure 9 The Server and The Client

IV. Review

Before we go to the Exercises section, we will tell you which files you have to implement for a minimal RMI client/server program. To write a minimal RMI client/server program you need to write the following.

- An interface for the remote object: **Remote.java**
  - Import: java.rmi.RemoteException
- An implementation of the remote object: **RemoteImpl.java**
  - Import: java.rmi.RemoteException
- A server that will run the remote object: **Server.java**
  - Import: java.rmi.registry.* và java.rmi.server.*
- A client to access the server: **Client.java**
  - Import: java.rmi.registry.*

V. Exercises

1. Based on the sample program, instead of passing only the name, you need to modify the entire in case the user wants to pass another parameter, for example, the user's age. You are required to program a new RMI application to meet this request.
2. Program a calculator RMI client/server program which including two public methods:
   - `public long add (long a, long b)`
   - `public long sub (long a, long b)`
3. (*) You are provided a list[1] contains 151,671 surnames of US citizens from 2000, which occur more than 100 times in the nation. You need to program a RMI application to meet the following requirements:
   - Your program must have a ***Person*** class.
   - A user wants to find whether an input surname is available. Command format:

```
java Client <host> <port> find "surname"
```

[1] http://bit.ly/names_census_gov

- Input a *pctwhite*, return all the surnames have *pctwhite* smaller than a given number. Command format:

```
java Client <host> <port> pctwhite <number>
```

- A user wants to list all ***Person*** whose *count* is larger than a given number. Command format:

```
java Client <host> <port> list <number>
```