Assignment No. 1

Aim: Implement multi-threaded client/server Process communication using RMI.

Objectives: To develop a multi-threaded client/server process communication using Java RMI.

Infrastructure:

Software Used: Java, Eclipse IDE, JDK

Theory:
Remote method invocation (RMI) allows a java object to invoke method on an object running on another machine. RMI provide remote communication between java program. RMI is used for building distributed application.

RMI applications often comprise two separate programs, a server and a client.

1. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
2. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to do the following:

- Locate remote objects.
- Communicate with remote objects.
- Load class definitions for objects that are passed around.

The RMI provides remote communication between the applications using two objects stub and skeleton. A remote object is an object whose method can be invoked from another JVM.

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

Remote Interfaces, Objects, and Methods

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called remote

objects.

An object becomes remote by implementing a remote interface, which has the following characteristics:

- A remote interface extends the interface java.rmi.Remote.
- Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, only those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Creating Distributed Applications by Using RMI

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

1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

1. Designing and Implementing the Application Components

First determine your application architecture, including the components of local objects and components accessible remotely.

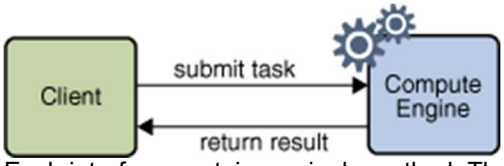
- **Defining the remote interfaces**. A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
- **Implementing the remote objects**. Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.
- **Implementing the clients**. Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

1.1 Writing an RMI Server

The compute engine server accepts tasks from clients, runs the tasks, and returns any results. The server code consists of an interface and a class. The interface defines the methods that can be invoked from the client. Essentially, the interface defines the client's view of the remote object. The class provides the implementation.

1.1.1 Designing a Remote Interface:

At the core of the compute engine is a protocol that enables tasks to be submitted to the compute engine, the compute engine to run those tasks, and the results of those tasks to be returned to the client. This protocol is expressed in the interfaces that are supported by the compute engine.



Each interface contains a single method. The compute engine's remote interface, Compute, enables tasks to be submitted to the engine. The client interface, Task, defines how the compute engine executes a submitted task.

The *compute.Compute* interface defines the remotely accessible part, the compute engine itself.

```
1. package compute;
2.
3. import java.rmi.Remote;
4. import java.rmi.RemoteException;
5.
6. public interface Compute extends Remote {
7.     <T> T executeTask(Task<T> t) throws RemoteException;
8. }
9.
```

By extending the interface java.rmi.Remote, the Compute interface identifies itself as an interface whose methods can be invoked from another Java virtual machine. Any object that implements this interface can be a remote object.
As a member of a remote interface, the `executeTask` method is a remote method, it is capable of throwing *java.rmi.RemoteException.* It is a checked exception, so any code invoking a remote method needs to handle this exception by either catching it or declaring it in its throws clause.

The second interface needed for the compute engine is the Task interface, which is the type of the parameter to the *executeTask* method in the Compute interface. The *compute.Task* interface defines the interface between the compute engine and the work that it needs to do, providing the way to start the work.

```
1. package compute;
2.
3. public interface Task<T> {
4.     T execute();
5. }
6.
```

The Compute interface's *executeTask* method, in turn, returns the result of the execution of the Task instance passed to it. Thus, the *executeTask* method has its own type parameter, T, that associates its own return type with the result type of the passed Task instance.

RMI uses the Java object serialization mechanism to transport objects by value between Java virtual machines. For an object to be considered serializable, its class must implement the *java.io.Serializable* marker interface. Therefore, classes that implement the Task interface must also implement Serializable, as must the classes of objects used for task results.

1.1.2 Implementing a Remote Interface

In general, a class that implements a remote interface should at least do the following:

- Declare the remote interfaces being implemented
- Define the constructor for each remote object
- Provide an implementation for each remote method in the remote interfaces

An RMI server program needs to create the initial remote objects and export them to the RMI runtime, which makes them available to receive incoming remote invocations. This setup procedure can be either encapsulated in a method of the remote object implementation class itself or included in another class entirely. The setup procedure should do the following:

- Create and install a security manager

- Create and export one or more remote objects
- Register at least one remote object with the RMI registry (or with another naming service, such as a service accessible through the Java Naming and Directory Interface) for bootstrapping purposes

```java
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
        }
    }
}

```

1.2 Creating a Client Program

A client needs to call the compute engine, but it also has to define the task to be performed by the compute engine.

The first class, `ComputePi`, looks up and invokes a `Compute` object.

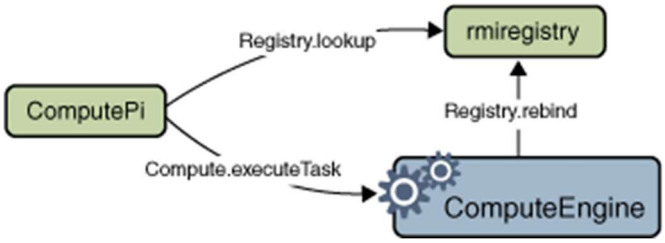
```java
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    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 comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
```

```
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Next, the client creates a new Pi object, passing to the Pi constructor the value of the second command-line argument, args[1], parsed as an integer. This argument indicates the number of decimal places to use in the calculation. Finally, the client invokes the executeTask method of the Compute remote object. The object passed into the executeTask invocation returns an object of type BigDecimal, which the program stores in the variable result. Finally, the program prints the result. The following figure depicts the flow of messages among the ComputePi client, the rmiregistry, and the ComputeEngine.



2. Compiling the Example Programs

In a real-world scenario in which a service such as the compute engine is deployed, a developer would likely create a Java Archive (JAR) file that contains the Compute and Task interfaces for server classes to implement and client programs to use. Next, a developer, perhaps the same developer of the interface JAR file, would write an implementation of the Compute interface and deploy that service on a machine available to clients.

2.1 Building a JAR File of Interface Classes: First, you need to compile the interface source files in the `compute` package and then build a JAR file that contains their class files.

2.2 Building the Server Classes: The engine package contains only one server-side implementation class, ComputeEngine, the implementation of the remote interface Compute.

2.3 Building the Client Classes: The client package contains two classes, ComputePi, the main client program, and Pi, the client's implementation of the Task interface.

3. Running the Program

Before starting the compute engine, you need to start the RMI registry. The RMI registry is a simple server-side bootstrap naming facility that enables remote clients to obtain a reference to an initial remote object. It can be started with the rmiregistry command. Before you execute rmiregistry, you must make sure that the shell or window in which you will run rmiregistry either has no CLASSPATH environment variable set or has a CLASSPATH environment variable that does not include the path to any classes that you want downloaded to clients of your remote objects.

3.1 To start the registry on the server, execute the rmiregistry command. This command produces no output and is typically run in the background.

```
start rmiregistry
```

By default, the registry runs on port 1099.

Once the registry is started, you can start the server. You need to make sure that both the compute.jar file and the remote object implementation class are in your class path. When you start the compute engine, you need to specify, using the java.rmi.server.codebase property, where the server's classes are network accessible.

```
Eg. java -cp c:\home\ann\src;c:\home\ann\public_html\classes\compute.jar
    -Djava.rmi.server.codebase=file:/c:/home/ann/public_html/classes/compute.jar
    -Djava.rmi.server.hostname=mycomputer.example.com
```

```
-Djava.security.policy=server.policy
   engine.ComputeEngine
```

3.2 Starting the Client

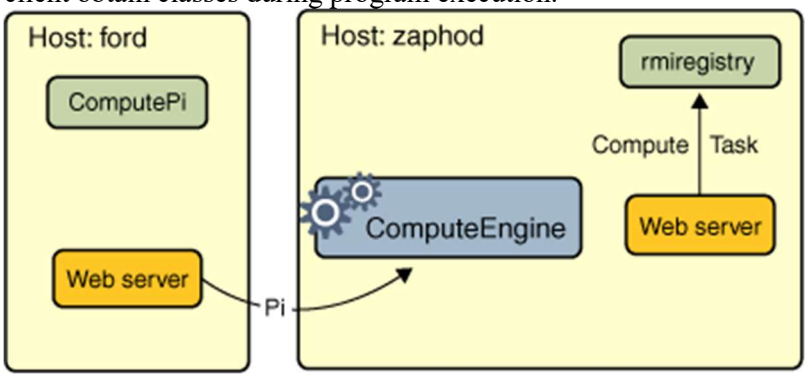
Once the registry and the compute engine are running, you can start the client, specifying the following:

- The location where the client serves its classes (the `Pi` class) by using the `java.rmi.server.codebase` property
- The `java.security.policy` property, which is used to specify the security policy file that contains the permissions you intend to grant to various pieces of code
- As command-line arguments, the host name of the server (so that the client knows where to locate the `Compute` remote object) and the number of decimal places to use in the π calculation

Start the client on another host (a host named mysecondcomputer, for example) as follows:

```
java -cp c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
     -Djava.rmi.server.codebase=file:/c:/home/jones/public_html/classes/
     -Djava.security.policy=client.policy
        client.ComputePi mycomputer.example.com 45
```

The following figure illustrates where the rmiregistry, the ComputeEngine server, and the ComputePi client obtain classes during program execution.



When the `ComputeEngine` server binds its remote object reference in the registry, the registry downloads the `Compute` and `Task` interfaces on which the stub class depends. These classes are downloaded from either the `ComputeEngine` server's web server or file system, depending on the type of codebase URL used when starting the server.

Because the `ComputePi` client has both the `Compute` and the `Task` interfaces available in its class path, it loads their definitions from its class path, not from the server's codebase.

Finally, the `Pi` class is loaded into the `ComputeEngine` server's Java virtual machine when the `Pi` object is passed in the `executeTask` remote call to the `ComputeEngine` object. The `Pi` class is loaded by the server from either the client's web server or file system, depending on the type of codebase URL used when starting the client.

Conclusion:

We implemented a multi-thread client/server process communication using RMI.