Building a Generic Compute Engine using RMI

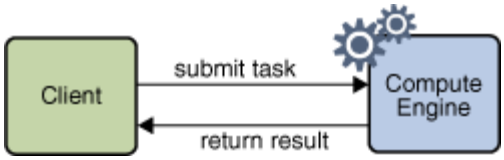
Introduction

This tutorial uses some of the content and the example provided in the official Java RMI tutorial by Oracle, which is located at http://docs.oracle.com/javase/tutorial/rmi/. The tutorial focuses on a simple, yet powerful, distributed application called a *compute engine*. The compute engine is a remote object on the server that takes tasks from clients, runs the tasks, and returns any results. The tasks are run on the machine where the server is running. This type of distributed application can enable a number of client machines to make use of a particularly powerful machine or a machine that has specialized hardware.

The ability to perform arbitrary tasks is enabled by the dynamic nature of the Java platform, which is extended to the network by RMI. RMI dynamically loads the task code into the compute engine's Java virtual machine and runs the task without prior knowledge of the class that implements the task. Such an application, which has the ability to download code dynamically, is often called a *behavior-based application*. Such applications usually require full agent-enabled infrastructures. With RMI, such applications are part of the basic mechanisms for distributed computing on the Java platform. Thus, RMI can also be used to expose services that are consumed by remote Java clients in Service-Oriented Architecture (SOA).

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. The remote communication for this protocol is illustrated in the following figure.



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.

Creating a Project for the Protocol

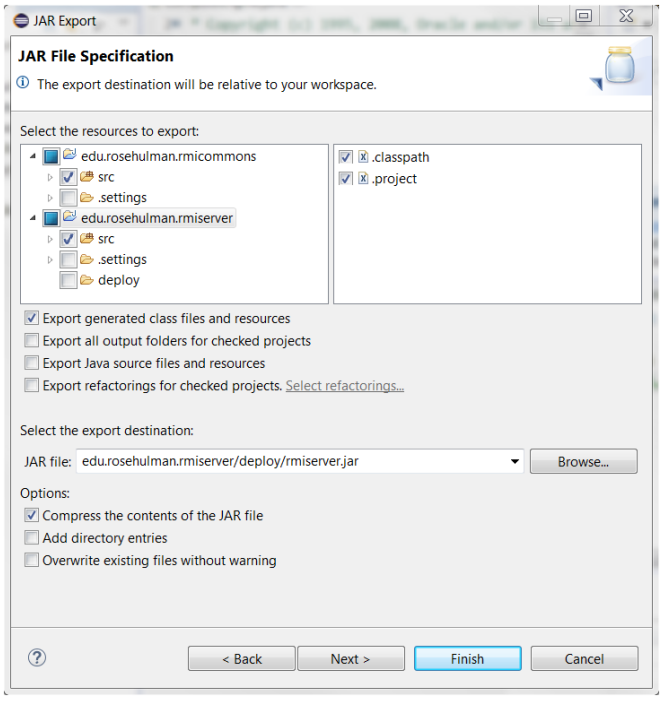
Download the **Resources** bundle from Moodle and unzip it. Go to the **commons** folder. There are two Java files that define the protocol above. We will create a Java Project in Eclipse using the contents in the commons folder.

1. Go to Eclipse -> File -> New Java Project.
2. Name your project: **edu.rosehulman.rmicommons** and use default settings.
3. Create a new package, under the **src** folder and also name it **edu.rosehulman.rmicommons**.
4. Copy paste **Compute.java** and **Task.java** from **Resources/commons** to the newly created package in Eclipse.
5. There should be no errors, but if you see one, then it must be due to package name mismatch. Fix it!
6. This project serves as a common protocol for both client and server.

Creating the Server Project

Go the **Resources/server** folder. There is one java file and other helper/configuration files. The **ComputeEngine.java** represents the server program. Let's create a project for the server program.

1. Go to Eclipse -> File -> New Java Project.
2. Name your project: **edu.rosehulman.rmiserver** and use default settings.
3. Create a new package, under the **src** folder and also name it **edu.rosehulman.rmiserver**.
4. Copy paste **ComputeEngine.java** (only) from **Resources/server** to the newly created package in Eclipse.
5. There should be some errors, because the server could not find **Compute** and **Task** classes defined in the **rmicommons** project.
6. Let's fix that now. Open **ComputeEngine.java** in your editor. Click on the error, i.e, line #44, "ComputeEngine implements Compute" part.
7. While the cursor is on the highlighted "Compute" part, press **CTRL + 1** -> **Fix Project Setup …** -> Choose the default option (adding the **rmicommons** project to the build path) -> Press Ok.
8. This should fix all compile errors.
9. Now create a **deploy** folder at the same level as **src**.
10. Copy **RunServerCmd.txt** and **server.policy** to the **deploy** folder from **Resources/server**.
11. We need to export the server project as a jar to run as an RMI server application.
12. Right click on the **rmiserver** project -> Export … -> Java -> JAR file. Press **Next**. **Carefully** choose and enter the following settings in the specification window and click **Next**.

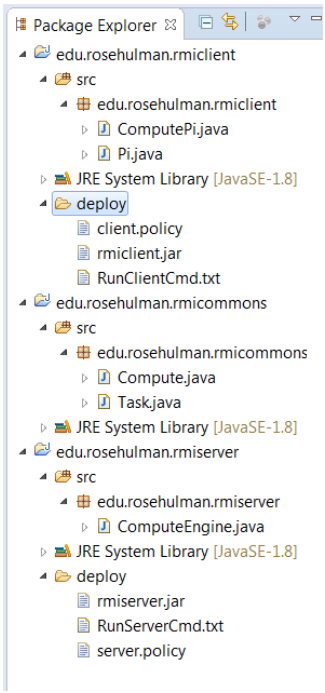


13. Use the default settings for JAR Packaging Options and click **Next**.
14. In the JAR Manifest Specification window, we will choose the class that should run when user, for instance double-clicks the Jar file to run it. Click **Browse** under **Main Class:** -> Select ComputeEngine and press Ok -> Click **Finish**.
15. Ignore the jar-signing warning and check your deploy folder to verify that **rmiserver.jar** was created.

Creating the Client Project

For the client project you will repeat all of the steps detailed in creating the server project. Name your client project the following: **edu.rosehulman.rmiclient**. You will copy **ComputePi.java** and **Pi.java** from **Resources/client** to the client project. **Please note that when you export the client project, you must also export the common project in the same jar as we did in the server project.**

At this point, your workspace should have the following directory structure:



Please create a pdf file and append your answers to the following questions:

Implementing Common Interfaces

Read the code in **rmicommons** and answer the following question:
1. What is the job of **Compute** and **Task** interfaces?

2. The **Compute** interface extends the **Remote** interface but the **Task** interface does not. When a client creates these objects, explain how they get sent to the server? [Hint: Slide# 9]

Implementing Server

Read the code in **rmiserver** and answer the following questions:
3. Which object is equivalent to a Broker (of the Broker Architecture) in **ComputeEngine** and why?

4. When a client gets hold of a stub of **ComputeEngine**, which method of the class will be remotely called by the client?

5. List all of the high-level steps involved in creating a RMI server object.

Implementing Client

Read the code in **rmiclient** and answer the following questions:
6. List all of the high-level steps involved in creating a RMI client.

7. Why is the PI class implementing both **Task** and **Serializable** interfaces?

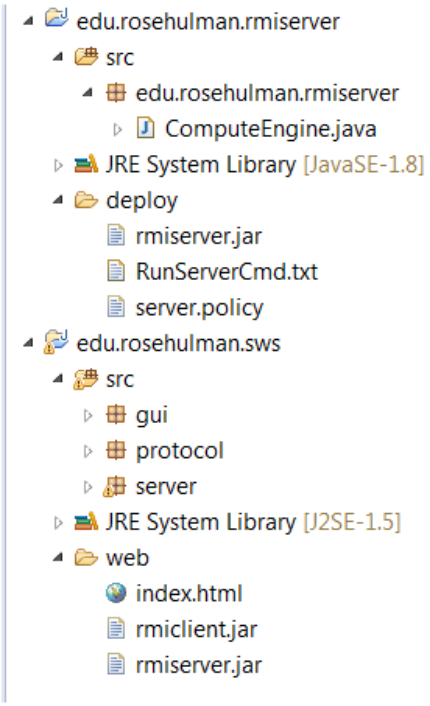
Running the Project

As you have already noticed, a JVM on the server side so far has no way of knowing about the classes that the client has developed in order to create an object out of the marshaled data. E.g., **PI,** which is a subtype of **Task**. RMI can use HTTP to transport these class definitions to the JVM on the server and vice versa. Hence, we need to setup a web server on both client and server machines for the RMI to work.

Setting up the Web Server

I have developed a simple web server for this course that you can use for this lab to transport class definition between client and server.

1. Import **Resources/web-server/sws.zip** as a Java Project into your Eclipse IDE.
2. Run the project as a Java Application (use **gui.WebServer** as the application entry point).
3. Select the **web** directory as the root directory to serve.
4. Click the **Start Simple Web Server** button.
5. Open your browser and type: http://localhost:8080. You should show the "Test Page Successful!" message.
6. Now copy **rmiserver.jar** and **rmiclient.jar** in the **deploy** directories of the rmiserver and rmiclient projects, respectively to the web directory so that our simple web server can serve these programs to a remote machine.
7. The directory structure of the **sws** project should look like the following:



Running the RMI Registry Service

In command line, **cd** to the **web** directory of the **sws** project, and type the following command:

web>*start rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false* [Windows - **Allow** any firewall messages]
web# *rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false &* [Linux/Unix/Mac]

This server is the broker that client will contact to look for service interfaces and to which the server will register its service interfaces.

Running the RMI Server

In the command prompt, **cd** to the **deploy** directory of the **rmiserver** project and type the following command (Also available in **RunServerCmd.txt** in that directory):

```
java -Djava.rmi.server.useCodebaseOnly=false -Djava.rmi.server.codebase="http://localhost:8080/rmiserver.jar
http://localhost:8080/rmicommons.jar" -Djava.rmi.server.hostname=localhost -
Djava.security.policy=server.policy -jar rmiserver.jar
```

If you want your server to be available for use over the local network, then use your **IP-address** instead of **localhost**. The **server.policy** file in the directory grants all privileges to the **rmiserver** to perform the task specified by the client programs.

Running the RMI Client

Open another command line, **cd** to the **deploy** directory of the **rmiclient** project, and type the following command (See **RunClientCmd.txt** in that directory):

```
java -Djava.rmi.server.useCodebaseOnly=false -Djava.rmi.server.codebase="http://localhost:8080/rmiclient.jar"
-Djava.security.policy=client.policy -jar rmiclient.jar localhost 45
```

If you want to use somebody else's server, then replace the first **localhost** by your IP-address and the second **localhost** by the remote server's IP-address. You should see the value of **PI** as the final output (45 – decimal points). Add the snapshot of the console output to the pdf.

Going Further

Without modifying your **rmiserver** or **rmicommons** projects, create a new client project that does the following:

1. Reads an integer value **n** from the command line or console.
2. Sends a **Task** to the **rmiserver** to **Compute** the first **n** prime numbers (use any algorithms).
3. The result of the task returned to the client must have the following (that the client prints on the console):
    a. The IP address of the server running the computation
    b. The IP address of the client requesting the computation
    c. The list of the **n** primes.
4. You will first test this programs (both client and server) on your machine.
5. You will then find a classmate whose machine will act as a server. (Also, return the favor by being the server for that classmate). Add the snapshot of the console output to the pdf.

Deliverables

You will turn in the following on Moodle **(Two Separate Files)**:

1. You will turn in a **pdf** with the answers to the questions above as well as snapshots of the console output after completing the **Running the Client** section and the snapshot of the console output after completing the #5 of the **Going Further** section.
2. Bundle all of your projects as one **zip** file and turn it in.