SE 424: Distributed Systems	**Recitation 3**
Semester 1 5785	**18 Nov 2024**
Lecturer: Michael J. May	**Kinneret College**

Remote Method Invocation (RMI)

In this recitation we will explore the use of RMI in Java and develop some more examples using it. The purpose of the recitation is to become more familiar with the tools that Java offers for Remote Method Invocation, not necessarily to explore all of the features that it provides. More information on Java RMI can be found at:

- Java RMI technology page with examples: `https://www.oracle.com/java/technologies/javase/remote-method-invocation-distributed-computing.html`
- Java RMI short tutorial (startup): `https://www.baeldung.com/java-rmi`
- Java RMI Module JavaDoc page: [`link here`]

For the first half of the recitation we will introduce the Java RMI model for Remote Process Invocation. The Java model is based on the client stub/server stub model that we talked about in class. Time permitting, we will also do a followup example with more features and capabilities (a bank server).

1 A First RMI Application: Greeting Processing

As a simple example, let us work over the major parts of a toy RMI example. This application is a "Hello world" example in which we will send data to a remote object which performs some simple processing of the input. The point is to see what's possible with RMI, not develop a serious application. We will develop the following classes:

- The remote class interface `Greeter` - it extends the `java.rmi.Remote` interface to indicate that it will be used for RMI. The methods defined in the interface are the ones that the remote clients will be able to see.
- One or more classes which implement Greeter. We'll make one called `GreeterImp`. It implements the methods defined so they can be called by others.
- A program that sets up a server with at least one copy of an implementation (`GreeterImp`). The server creates the object of type `GreeterImp` and registers it for access by other objects, even those which are on other computers. As part of the registration, it produces a *stub* which is handed over to an service called the *RMI registry*. The object is identified by a name which others can use to look it up. We'll put this functionality in the `public static void main()` function of another class (`TheServer`) [1].
- A *client* that knows where the server is located and the name of the remote object that it wants. It pulls the remote object using **lookup**, casts it to the class that it wants, and uses it as it would any other object. We'll put this functionality in the `public static void main()` function of a separate package in a class called `GreeterClient`.

The client program must be aware of the interface that the remote object implements. That implies that it is compiled with it available. Java RMI offers a mechanism to dynamically download class definitions and interfaces over the web. We're not going to use that functionality in full in this recitation, but you can read more about it online.

The basic layout of the project is shown in Figure 1.

To keep things clear, we're going to develop the client and server in two separate Intellij projects.

[1] Note that we could have just as easily put the function in `GreeterImp`

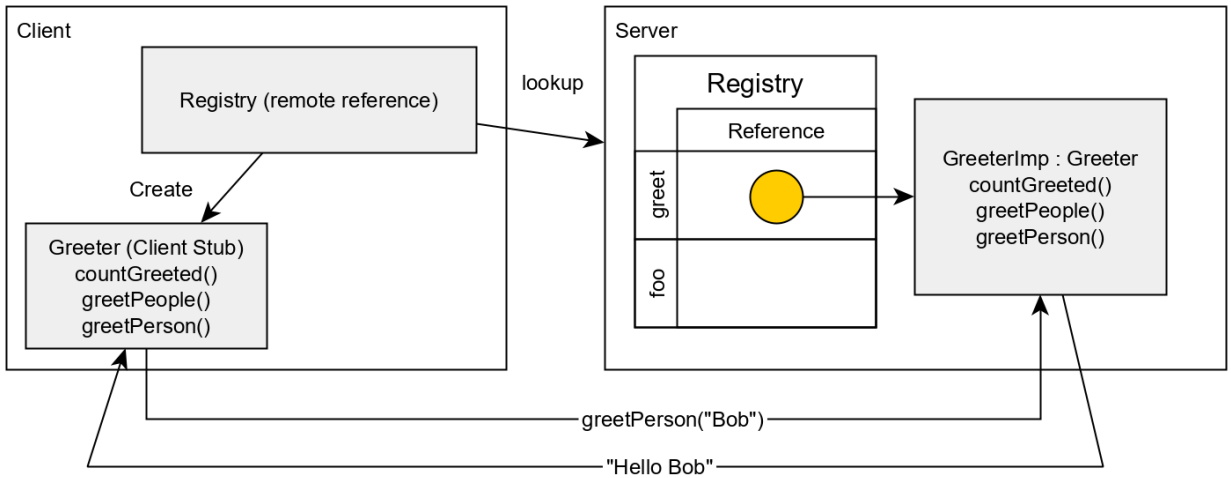

Figure 1: Sketch of Greeter RMI Parts

2 Project 1: The Server

2.1 Interface: Greeter

We'll start by implementing the `Greeter` interface which extends the `Remote` interface. The interface has three methods:

```
public interface Greeter extends Remote {
 /**
  * Counts the number of people greeted so far
  * @return Count of people greeted so far.
  */
 int countGreeted() throws RemoteException;
 /**
  * Sends back of a list of greeting for multiple people
  * @param s List of people to greet divided by ;
  * @return Array of greetings
  */
 String[] greetPeople(String s) throws RemoteException;
 /**
  * Greets a single person
  * @param s Name of the person to greet
  * @return A greeting for the person
  * @throws RemoteException
  */
 String greetPerson (String s) throws RemoteException;
}
```

2.1.1 What to do

Create the interface `Greeter` as above.

2.2 Class: GreeterImp

We then create a class `GreeterImp` that implements `Greeter` and extends `UnicastRemoteObject`:

```
public class GreeterImp extends UnicastRemoteObject implements Greeter {
...
}
```

It will implement the greeting processing functionality of `greetPeople` and `greetPerson` and return a count of greetings processed so far in `countGreeted`. We could just as well as have `GreeterImp` implement `Remote` directly, but then we'd miss the chance for polymorphism where we might offer multiple implementations of how to process greetings.

2.2.1 Remote Object State

`GreeterImp`'s methods will keep track of the count of greetings sent so far using a counter. It also will keep track of who has been greeted previously. If a new person's name is sent (*e.g.* Henry), the greeting will be "Hello there Henry!". From then one, whenever the name is sent again, it will receive the greeting "Welcome back Henry!". The object will keep its state in a `Vector<String>` object.

See Section 3.2 for a sample output to see how the server should respond.

2.2.2 What to do

Implement the logic for the three methods in `GreeterImp`.

2.3 Class: TheServer

The last part necessary for the server is the code which sets up the remote object for access by others. It must create an instance of `GreeterImp` and register it with a `Registry`.

2.3.1 The RMI Registry

The RMI Registry is a Java service that allows you to post RMI accessible objects from other computers. The default port for the registry service is 1099. There are two ways to start it up:

1. Start it up manually from the command line. The rmiregistry.exe file can be found in the Java JDK's bin directory.
2. Start it up via code. Include the following line in your server:

   ```
   Registry registry = LocateRegistry.createRegistry(1099);
   ```

   Note that this creates a new registry on port 1099. If you already have an instance running on 1099, it should throw an error.

Once the registry is created, you can use it to publish objects for access via RMI.

2.3.2 Writing the code

The critical lines of code in the server are as follows:

```
// make a greeter object
GreeterImp greeter = new GreeterImp();
// get a hook to the local registry
Registry registry = LocateRegistry.createRegistry(1099);
// now bind to it to the name 'greet'
registry.rebind("greet", greeter);
```

The above code creates a Registry accessible on port 1099 and registers the object we want to publish (greeter) with it under the name "greet".

3 Project 2: The Client

The client side of the RMI system is simpler than the server side. It receives names from the user and passes them to the remote object for processing.

As with other client/server applications, we need to give the client the IP address of the server as well as the name of the object which can be accessed remotely. We'll pass the parameters to the client application using the `String[] args` parameter in the `public static void main()` method. The parameters can be entered in the *Program arguments* section of the *Run Configurations* window for the client as shown in Figure 2.

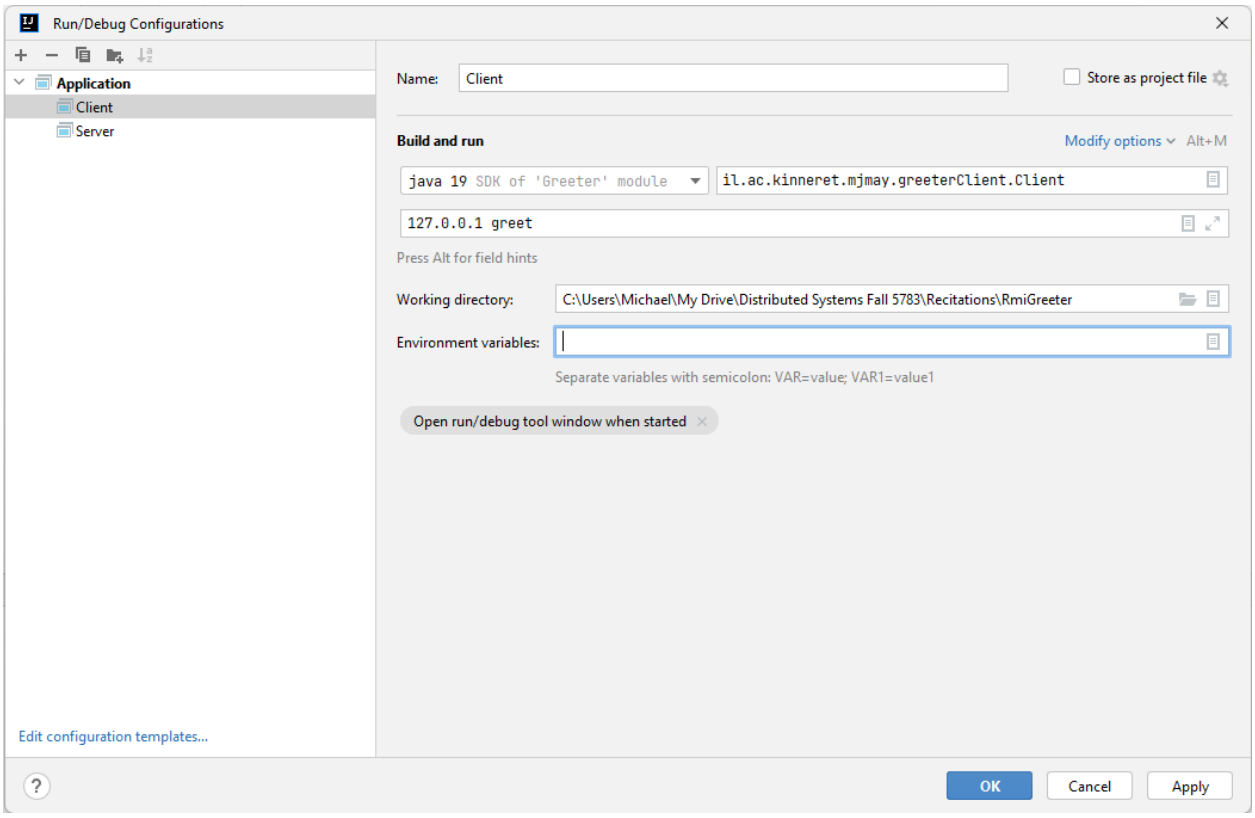

Figure 2: Client Parameters

The client begins by searching the registry of the server for the desired object:

```
Registry registry = LocateRegistry.getRegistry(remoteHost); // args[0]
Greeter greetingServer = (Greeter) registry.lookup(remoteObjectName); //args[1]
```

In the first line, a reference to the registry of the server (IP address is stored in `remoteHost`) is retrieved and stored in `registry`. In the second line, the requested object is looked up using `registry.lookup(remoteObjectName)` (the name of the object is stored in `remoteObjectName`). It's then cast into a `Greeter` type which can be used for sending sentences for processing.

Note that the client is assumed to recognize the type `Greeter` at compile time. This is necessary since it subsequently uses methods defined in it. We accomplish this by having the client depend on the server's

project. We could also have compiled the class definitions into a JAR file and introduced it as a library dependency.

3.1 Client Behavior

After retrieving the remote object instance, we have the user input a bunch of names, pass them to the remote object, and then display the results. To emphasize the fact that the remote object is actually on the server and not on the client, we have the remote object echo the names it receives to the console and have the client echo the current state of the remote object after processing.

The client follow the following pseudocode:

```
while the user hasn't chosen to quit {
 Prompt for input
 Print the total number of people greeted so far using countGreeted()
 If the input is a single name {
  Send the name to the server using greetPerson()
  Output the greeting from the server
 }
 else if the input is a list of names (; delimits) {
  Send the list of name to the server using greetPeople()
  Output the list of greetings from the server
 }
Print the total number of people greeted so far using countGreeted()
}
```

3.2 Sample Client Output

A sample input/output trace from the client is shown below. It took place after the remote object had already taken care of some client requests before.

```
Bound to greet
Before sending 8 people have been greeted
Enter a people to greet delimited by ; (blank to quit): Alice;Bob;Cid;Dan
Hello there Alice!
Hello there Bob!
Hello there Cid!
Hello there Dan!
Greeted 12 people so far

Enter a people to greet delimited by ; (blank to quit): Ed
Hello there Ed!
Greeted 13 people so far

Enter a people to greet delimited by ; (blank to quit): Fanny;Gail;Harry;Alice;Bob;Cid
Hello there Fanny!
Hello there Gail!
Hello there Harry!
Welcome back Alice!
Welcome back Bob!
Welcome back Cid!
Greeted 19 people so far

Enter a people to greet delimited by ; (blank to quit):
Bye!
```

4 Working Between Computers

If you want to have the RMI application work between different computers, you need to ensure that the client has been compiled with a reference to the class that is to be transmitted by RMI. That means you'll need to compile the Server and Client into JAR files and distribute them where they need to be.

If you don't compile the client with the correct version of the server object (for instance what may happen if the server and client are written by different people or you take the sample code for the server and try to use it with your own client), the RMI system will not transfer the object correctly.