Lab: Remote Method Invocation

Preamble

This lab uses the IntelliJ IDEA project template provided on Campus, which contains code snippets to help you with the exercises. The project contains several packages named `exercise1`, `exercise2`, etc. each corresponding to one of the exercises below. To do an exercise, copy/paste the classes from `exercise1` into the corresponding package and modify them if needed.

Note: should you need it, you can activate RMI traces by launching the JVM with the following parameter: -Djava.rmi.server.logCalls=true

1 Local Execution

Package `exercice1` provides a simple example of an RMI client / server application. First, study the code of the application without executing it and answer the following questions: **Q1.** How many programs (i.e. main methods) does the application contain? **Q2.** For how long do those programs execute? **Q3.** In which order should you launch them?

Launch the programs one after the other and observe the messages they output in their console. Now, launch the client program again. **Q4.** What do you observe in the server's console? Explain what you see.

Q5. Perform the following steps and explain what you see at each step, in particular in the client's console:

1. stop the registry and launch the client again
2. launch the registry again, then the client
3. stop the server, launch it again, then the client
4. stop the server and launch the client again

Q6. What conclusion can you draw from this exercise regarding the test of a distributed application?

2 Distributed Execution

We now want to execute the client/server application of exercise 1 on distinct machines, as in real life. **Q1.** Where should you deploy the programs of the application? **Q2.** Which are the classes that both the client and the server program use? **Q3.** Which constants do you need to modify in the client and/or the server?

Select a classmate to execute the application on distinct machines and select the role – client or server – you will play. On each machine, copy the relevant classes of exercise 1 into package `exercise2`. **Caution**: (1) Do not copy the classes that are not relevant to your role: see questions Q1 and Q2. (2) After copying the classes in package `exercise2`, make sure all the imports refer to package `exercise2`, not `exercise1`. Modify the constants in the client and/or the server (see question Q3).

Notes:

1- Check that your server is listening on the right IP address, i.e. the one corresponding to your wifi network interface. If not, set the java.rmi.server.hostname system property to the appropriate value on the server. Example: -Djava.rmi.server.hostname=192.168.1.10.
2- If the local wifi network does not allow computers to communicate with one another, use your phone as a hotspot through which the two machines are connected.
3- To specify launch parameters or system properties in IntelliJ, select go to *Run>Edit Configurations...* In the window that opens, select the right *Main class*, then enter your *VM options* and/or *Program parameters* in the appropriate field. Note: system properties definitions (-D…=…) must be passed as VM options.

3 Stateful Server

The server we used so far is stateless, since it does not have any attribute. This exercise demonstrate that a RMI server can also be stateful.

Based on the example given in exercise 1, develop your own client / server application. The server is a property repository, i.e. a set of (key, value) pairs. The server is accessed by multiple clients that set and/or retrieve property values (see the provided PropertyRepository interface). Note that the server has to handle the fact that several clients may access it simultaneously.

Test your application with multiple clients that either set or get property values.

4 Generic Server

So far, the servers we developed were dedicated to a specific task: sorting string lists, storing (key, value) pairs. However, we can define a generic server that can execute a generic task. The server implement the Computer interface provided. It can execute any task that implements the Task interface provided.

This construct is very flexible: the task to execute does not need to be known when the server is compiled or even when the server is launched. The task will be known at the very last moment, when the client request the server to execute it.

Develop a client / server application using the Task and Computer interfaces provided. Test your application in a distributed setting and have your client submit two distinct tasks of your choice to the server.

5 Dynamic Class Loading

Copy the classes of exercise 1 into package `exercise5`. Again, check that the all the imports are correct. **Q1.** Modify the server code so that the reverseSort() method returns an object of type MyList (the class is provided) instead of an ArrayList as in sort().

Launch the application all over again (registry, server, client) on two distinct machines. **Q2.** What do you observe on the client? Try to interpret the problem by reading the error message. Why does the call to sort() succeeds and the call to reverseSort() fails?

Fix the problem using dynamic class loading, as follows:

1. *Allow RMI to download missing classes from the server*. As a safety measure, RMI won't download any class until a Security Manager is configured in the local JVM. To configure a (empty, i.e. allow-nothing) security manager, simply define the system property named java.security.manager: -Djava.security.manager
2. Define a security policy that (i) allows RMI to connect to the server, (ii) allows the downloaded code to execute with all permissions (just as local code). This policy is defined in the provided `security.policy.txt` file. Instruct the JVM to use the file as follows: -Djava.security.manager=src/exercise5/security.policy.tx*t*
3. *Launch a HTTP server on the server*. The HTTP server will answer the code download requests sent by the client-side RMI runtime. Any HTTP server can do and as a convenience, a simplistic HTTP server is provided in the package `exercise5.classserver`. By default, the server is listening on port 2001 and looks for class files in the project's root directory.
4. *Setup the RMI server to instruct the client* (i) which HTTP server to use to download missing classes, (ii) the root directory containing the classes. This information is supplied through the java.rmi.server.codebase system property. The setup for the las is as follows: -Djava.rmi.server.codebase=http://<@IP>:2001/bin

Q3. Launch the application all over again (registry, HTTP server, RMI server and RMI client) with the appropriate setup and check that it runs correctly. **Q4**. Comment and/or un-comment the various sections of the policy file and observe the consequences on the client. Explain what you observe.