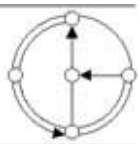
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



SS 2006 Prof. Gustavo Alonso / Angela Nicoară / Patrick Stüedi

Computer Networks
Exercise 4

Start date: May 5, 2006
Due date: May 12, 2006

1. Introduction to Remote Procedure Call (RPC) / Java RMI

RPC is a powerful technology for creating distributed client-server based applications. It enables the communication between two processes - the client process and the server process, which may reside on the same computer or on different computers. The client process makes a remote procedure call to a procedure that is implemented by the server and retrieves the result.

Java RMI is a Java API for performing remote procedure calls (RPC). It allows to invoke methods on objects that exist on the same machine or a different one. The server implements a remote interface that provides the methods that can be called from the clients.

Java / RMI Settings:

To use RMI, the J2SE SDK (Software Development Kit) must be installed and the Java environment needs to be set up:

- Download and install the J2SE 1.4.2 SDK or J2SE 1.5.0 SDK from the http://java.sun.com website.

- Set the PATH environment variable. The **<SDK_HOME>** is the home directory where J2SE SDK is installed.

  ```
  bash$ export PATH=/<SDK_HOME>/bin:$PATH
  ```

- Set the CLASSPATH environment variable:

  ```
  bash$ export CLASSPATH=/<SDK_HOME>/jre/lib/rt.jar:$CLASSPATH
  ```

- After the J2SE SDK has been installed, the following Java tools can be found under the **`<SDK_HOME>/bin`** directory:
  - **`java –`** the tool that launches a Java application
  - **`javac –`** the Java programming language compiler
  - **`rmic –`** the Java RMI compiler
  - **`rmiregistry -`** the Java Remote Object Registry

RMI Example:

In what follows, we will give an example of a RMI client and server. The server exposes a remote method `int addOne(int i)` to the clients. With the input arguments `i` from clients, `addOne()` simply returns `i+1`.

To create and run the RMI example, the following steps are necessary:

1.1. Writing a RMI server:

- Define the remote interface, in this case, `Calculator.java`:

```
package example;

import java.rmi.*;

public interface Calculator extends Remote {
    public int addOne(int i) throws RemoteException;
}
```

Figure 1. The remote interface `Calculator.java`

The interface must be public and extend the `java.rmi.Remote` interface. The `addOne(int i)` method of the server class, which implements this interface, is called from the remote client. Each remote method in the interface must declare that it throws `java.rmi.RemoteException` when a failure occurs during the remote invocation. Other exceptions may also be thrown. Catching and handling the exceptions are up to the clients that use the remote methods.

- Create the `CalculatorImpl.java` class that implements the remote interface `Calculator.java`:

```
package example;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculatorImpl implements Calculator {

        public CalculatorImpl() throws RemoteException {
            UnicastRemoteObject.exportObject(this);
        }

        public int addOne(int i) throws RemoteException {
            return i+1;
        }
}
```

Figure 2. The remote class `CalculatorImpl.java`

The `UnicastRemoteObject.exportObject(this)` method exports the remote object, so that it is made available to accept incoming calls from clients. In this way, it is ensured that objects of the class can be used as remote objects, and can be invoked remotely.

- Create the server class `CalculatorServer.java` to handle the client's requests:

```
package example;

import java.net.*;
import java.rmi.*;

public class CalculatorServer {

  public static void main(String[] args){
      try{
          CalculatorImpl c = new CalculatorImpl();
          Naming.rebind("calculator" , c);
          System.out.println("Calculator Server Ready!");
      }
      catch (RemoteException e) {
          System.out.println("Exception in CalculatorImpl.main: " + e);
      }
      catch (MalformedURLException e) {
          System.out.println("A malformed URL has occurred: " + e);
      }
  }
}
```

Figure 3. The server class `CalculatorServer.java`

This class provides the rest of the code that makes up the server program: a `main` method that creates an instance of the remote object and registers it with the naming facility. The `java.rmi.Naming` interface is used for binding, or registering, and looking up remote objects in the registry (i.e., a server that relates objects with names). `Naming.rebind("calculator", c)` binds the calculator object with the name "calculator". The binding name is case-sensitive. Calling the `rebind` method makes a remote call to the RMI registry on the local host. Together with the host name or the IP address where the server is running, a URL address "**rmi://host/calculator**" is used to address the calculator object.

The clients uses the URL "**rmi://host/calculator**" to get the reference to the remote object and invoke the method. The URL includes the host name (**host**), or IP address, on which the registry and remote object is being run and a name (**calculator**) that identifies the remote object in the registry.

1.2. Creating a RMI client:

Before a client can call the remote method `addOne(int)`, it needs to retrieve the remote reference to the calculator object. RMI provides a `lookup()` method in the `java.rmi.Naming` class to get the reference to the remote object:

```
Calculator cal = (Calculator) Naming.lookup("rmi://host/calculator");
```

The client class example is as follows:

```
package example;

import java.rmi.*;
import java.net.*;

public class CalculatorClient {

  public static void main(String args[]) {
    if (args.length == 0 || !args[0].startsWith("rmi:")){
      System.out.println("Usage:
         java calculatorClient rmi://host/calculator number");
      return;
    }

    try {
      Calculator cal = (Calculator ) Naming.lookup(args[0]);
      int input = (new Integer(args[1])).intValue();
      int output = cal.addOne(input);
      System.out.println("The output of addOne(" + input + ")" + " is " + output);
    }
    catch (MalformedURLException e) {
      System.out.println(args[0] + " is not a valid RMI URL");
    }
    catch (RemoteException e) {
      System.out.println("Remote object throw exception: " + e);
    }
    catch (NotBoundException e) {
      System.out.println("Cannot find the requested remote object on the server");
    }
  }
}
```

Figure 4. The remote client class `CalculatorClient.java`

The client constructs a name (`calculator`) used to look up a calculator remote object. The argument of the `lookup` method (`args[0]`) is the URL address of the remote host on which the calculator object runs. The URL passed to the `Naming.lookup` method has the same URL syntax as the URL passed in the `Naming.rebind` call, which was discussed earlier.

1.3. Compiling and running the example:

- Compile the client and the server programs:

```
bash$ javac example/*.java
```

- Generate the stub and skeleton class files for remote objects:

```
bash$ rmic example.CalculatorImpl
```

Running **`rmic`** on the `CalculatorImpl` class will generate the stub and the skeleton classes for the `CalculatorImpl` remote object (`CalculatorImpl_Skel.class` and `CalculatorImpl_Stub.class)` in the `example` directory.

Note: In J2SE 1.5.0 SDK, by default, **`rmic`** does not generate any skeleton classes.

- Starting the server: Before starting the server, the **`rmiregistry`** must be started and leave it running in the background:

```
bash$ rmiregistry &
bash$ java example.CalculatorServer
```

Note: Make sure that the classpath used to start the **`rmiregistry`** includes the stubs and skeleton class files generated in the previous step.

- Running the client:

```
bash$ java example.CalculatorClient rmi://host/calculator 100
```

- The result is then displayed:

```
The Output of addOne(100) is 101
```

2. Task: Implementation of a server-client application with RMI

The objective of this assignment is to implement a server and two clients that exchange messages through the server, as shown in Figure 5.

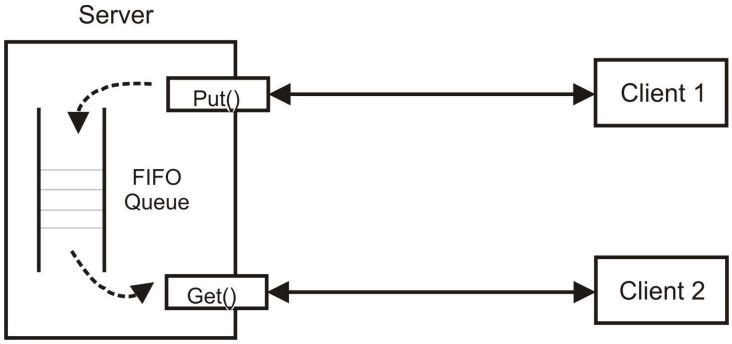


Figure 5: The Message Server

- The server **`MessagePoolServer`** implements two remote methods defined in the interface class **`MessagePool`**:

```
import java.rmi.*;

public interface MessagePool extends Remote {
    public void put(String msg) throws RemoteException;
    public String get() throws RemoteException;
}
```

Figure 6: The remote interface **`MessagePool.java`**

- the **`put()`** method accepts a String message from the client, and stores it into the FIFO (First In First Out) queue in the server. In case the queue is full, the **`put()`** operation will fail, and a **`QueueFullException`** exception will be thrown.

In case the message from the client is null, the server will throw a **`MessageNullException`** exception.

- the **`get()`** method retrieves the message out of the queue to the client which invoked it. The retrieved message will be deleted from the queue. In case the queue is empty, the **`get()`** operation will fail and a **`QueueEmptyException`** exception will be thrown.

- The FIFO message queue **`MessageQueue`** has a maximum size of 100 messages. Messages are strings with the maximum length of 100 characters.

- Implement two clients: **`MessagePutClient.java`** generates messages periodically (1 message per 1 second), and **`MessageGetClient.java`** retrieves messages periodically (1 message per 2 seconds). The message could be the timestamp of the client or any random generated string.

- Make sure all cases are handled:
  - Retrieving from an empty queue / Adding a message to a full queue
  - Trying to add a message to a full queue

- For this assignment the program should be single threaded.