Lab 2 : Java RMI

1. Goals

In this lab you will work with a high-level mechanism for distributed communication. You will discover that Java RMI provides a mechanism hiding distribution in OO programming. Java RMI does provide:

- location transparency
- naming transparency
- programming support for complex data exchange
- programming support for application-level protocols

Java RMI does not provide failure transparency.

2. Official documentation

[1] https://docs.oracle.com/javase/tutorial/rmi/

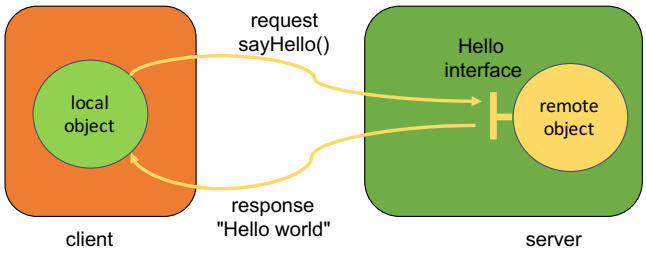
[2] http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html

[3] http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136424.html

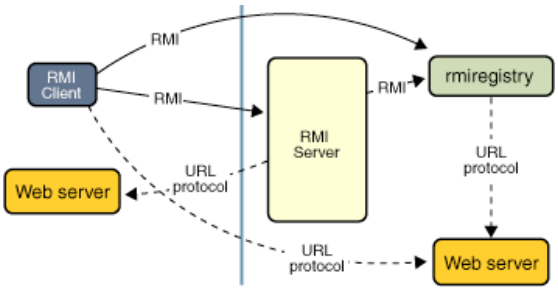
3. The RMI Model

RMI follows the client-server model. The server provides access to remote objects which implement well-defined interfaces. A client may obtain the reference to a remote object and call it using the interface.

In the picture below, the server provides a service (remote object) implementing the Hello interface. The client may call the sayHello() method if it gets the reference to the remote object. The object will return its response.



To obtain the reference of a remote object, a client may use the RMI naming service (rmiregistry) or **receive it as a result of a method call to another remote object.**



4. Exercice : Run and Understand the HelloWorld Example

The goal of this exercice is to make you discover the basic principles of Java RMI and make you succeed in running a Java RMI application.

3.1 The Remote Interface

In your favorite IDE, define the remote interface Hello.

Hello.java

```
1. public interface Hello extends Remote {
2.
3.     // A method provided by the
4.     // remote object
5.     public String sayHello() throws RemoteException;
6.
7. }
```

Pay attention to the fact that the interface needs to extend the Remote interface and that each remotely-accessible method is to throw a RemoteException. If you do not follow these rules, you cannot benefit from the RMI framework and from the automatic management for the distributed communication between the client and the server.

3.2 Implementation of the remote object

Create a class that implements the remote interface. It will provide the implementation of the service that will be mad accessible by the server.

HelloImpl.java

```
1. import java.rmi.*;
2.
3. public  class HelloImpl implements Hello {
4.
5.     private String message;
6.
7.     public HelloImpl(String s) {
8.            message = s ;
9.     }
10.
11.    public String sayHello() throws RemoteException {
12.           return message ;
13.    }
14.}
```

3.3 The server

- The server creates the object that implements the service in line 11.
- In line 12, it obtains the remote reference of the object.
  This reference is needed by the clients to communicate with the object. As you see, we do not manipulate explicitly the IP address of the physical machine, nor the port. The exportObject method takes care of all the underlying details (actually, there will be a socket and a port allocated, but RMI hides this from us). The value 0 for the port parameter says that an available port will be used.
- To help clients obtain the reference of the newly created object, the server puts the reference in the Java naming service (the registry). The server obtains a reference to the registry (line 15) and registers the reference of the object under a given name (line 16).
  For this to work, the registry should already run. If LocateRegistry is used without parameters, the registry would be looked up at the localhost. Otherwise, it is possible to connect to a distant registry via getRegistry(String host, int port).

HelloServer.java

```
1. import java.rmi.*;
2. import java.rmi.server.*;
3. import java.rmi.registry.*;
4.
5. public class HelloServer {
6.
7.   public static void  main(String [] args){
8.
9.      try {
10.        // Create a Hello remote object
11.        HelloImpl h = new HelloImpl ("Hello world !");
12.        Hello h_stub = (Hello) UnicastRemoteObject.exportObject(h, 0);
13.
14.        // Register the remote object in RMI registry with a given identifier
15.        Registry registry= LocateRegistry.getRegistry();
16.        registry.bind("HelloService", h_stub);
17.
18.        System.out.println ("Server ready");
19.
20.     } catch (Exception e)  {
21.        System.err.println("Error on server :" + e) ;
22.         e.printStackTrace();
23.     }
24.
```

3.3 The client

- The client locates the registry (line 13). This should be the registry used by the server to register the hello service.

- The client searches for the hello service using the registry and using a predefined name (line 14)
- If the reference is obtained, the client may call the remote object using the standard method call syntax. In other words, for this part, the distribution is transparent for the developer.

HelloClient.java

```
1. import java.rmi.*;
2. import java.rmi.registry.*;
3.
4. public class HelloClient {
5.   public static void main(String [] args) {
6.     try {
7.       if (args.length < 1) {
8.        System.out.println("Usage: java HelloClient <rmiregistry host>");
9.        return;}
10.
11.    String host = args[0];
12.    // Get remote object reference
13.    Registry registry = LocateRegistry.getRegistry(host);
14.    Hello h = (Hello) registry.lookup("HelloService");
15.    // Remote method invocation
16.    String res = h.sayHello();
17.    System.out.println(res);
18.
19.    } catch (Exception e)  {
20.          System.err.println("Error on client: " + e) ;
21.   }…
```

3.4 Run the application

- Compile the sources
- Define the CLASSPATH variable.
  The classpath gives the location(s) of the class files to be used by java.
  The easiest is to define it in a global manner, otherwise you can use the -cp option when launching the java command.
- Run the rmiregistry

```
rmiregistry &
```

- Run the server

```
java HelloServer or java –cp …your classpath… HelloServer
```

- Run the client

```
java HelloClient localhost
```

5. Exercice : Modify the Hello Application

The goal of this exercice is to make you discover:

- *that RMI references may be passed as parameters (hence also received as results)*
- *that clients may also be servers*

Goal

Each time the sayHello() method is called on the server side, the server will print a message giving the identity of the client that called the method.

Design discussion

There are two alternatives:

1) The client provides its name (a String) each time it calls the sayHello() method. This will imply to change the Hello interface in the following way:

```
public interface Hello extends Remote {
	public String sayHello(String clientName) throws RemoteException;
}
```

2) The server contacts the client to ask it for its name. This means that the server makes a *remote* call to the client. In RMI, the only possible way of doing this is that the client provides a remote object implementing a remote interface. The server should obtain a reference to this object to be able to call it. This reference may be passed to the server by the client itself (i.e this reference does not need to be put in the Java registry).

Your turn!

1) Implement the first solution

2) Implement the second solution

- Define a remote interface Info_itf for the client

```
public interface Info_itf extends Remote {
	public String getName() throws RemoteException;
}
```

- Modify the client to implement the Info_itf.
- Modify the Hello interface to become

```
public interface Hello extends Remote {
	public String sayHello(Info_itf client) throws RemoteException;
}
```

- Change the implementation of the Hello interface to call the client and print the message about its identity.

3) Compare the solutions in terms of distributed communications

6. Exercice : Continue with the Hello Application

The goal of this exercice is to make you discover:

- *that servers may provide access to different services (different interfaces)*

Goal

We want now that the server informs the client when it has reached a certain number of calls of the sayHello() method. For example, the server may notify the client that it has reached the limit of 10 calls, then 20 calls, etc.

Design discussion

The server should be able to :

- call a client to inform it about the number of calls it has made.
- distinguish between clients (the server may have multiple clients) so as to count the number of calls per client. This requires that, at each call of the the sayHello method, the clients should identify themselves.

Your turn!

Modify the Hello application in the following way :

- Define a remote interface `Accounting_itf` for the client

```
public interface Accounting_itf extends Remote {
      public void numberOfCalls(int number) throws RemoteException;
}
```

- Change the client to implement this interface
- Define a new remote interface for the server Registry_itf

```
public interface Registry_itf extends Remote {
      public void register(Accounting_itf client) throws RemoteException;
}
```

This interface is to be used by a new client of the service Hello. The client will thus present itself to the server. The server should manage somehow the list of clients it knows.

- Provide the implementation of the interface at the server side.
- Provide a Hello2 interface in which the sayHello() method takes as a parameter a `Accounting_itf` type. The method should not be executed if the client has not registered itself.
- Provide the implementation of the Hello2 interface to count the number of calls in the sayHello method
- Make the server bind to the registry the three services : Hello, Hello2 and Registry_itf
- Test the application. Note that with this version the client can obtain the references to Hello and Hello2 and use two different implementations of the service.

7. Exercice : The Chat Application

The goal of this exercice is to make you practice the architectural design of an RMI application. Contrary to the previous examples, it will be up to you to define the entities, their roles and their interfaces.

Your task is to build a chat application. There are four stages that are proposed to you:

a) build a chat system, where participants can dynamically join, leave, exchange messages
b) add a history feature to the chat system. This means that the server « remembers » all the messages that have been exchanged since the start of the system and that a new client can consult this history.
c) make the chat system persistent (persistent history of messages). This means that if the server shuts down and restarts, it can recover the message history.
d) implement a graphical user interface for the chat system. This feature is for the ones that find RMI really easy, have plenty of time to do other stuff, are tired of the textual interface of the chat system and enjoy graphics!