Day 22: Network programming (RMI)

1 Network programming

For the better part of the history of computing, computers have worked in isolation. Calculations that took place in one computer were independent from those taking place in another one. The only way that computers could communicate was by passing programs or data indirectly using external data storage units like perforated tapes, magnetic disks, or CDs (and using a human being to move them from one computer to the next). In the 1960s-70s computers were physically huge machines to which users connected through so-called dumb terminals. In the 1980s-90s personal computers became the norm, and therefore there were many more computers in the world, but they were still mostly isolated from each other.

The development of network technologies and the Internet made it feasible to communicate computers directly and in real time, faster and faster every year. This has opened the door to network programming, where programs run in more than one computer at the same time. In the early stages, most of the computation was done in one machine (where the data was) and a little part of it (visualising the results) was done in another one (where the human user was). Nowadays, grids and clouds are able to run programs on several computers at the same time, each of them working only on a small part of the problem, giving the general impression of having a bigger, broader, more powerful computer running the whole program.

In this section we will learn how to create programs that run in more than one computer; we will have different objects of the same program communicating with each other through the network, even if they live on different machines. Although the topic of network programming is very broad and exceeds the scope of this module, the main concepts that we will learn are common to most network technologies.

1.1 The basics

Most network applications follow the client–server paradigm[1]. This means that the program has two parts: *server* and *client*. The former is runnning in a well-known computer, waiting for clients to request its services. *Clients* run on different computers and use the services provided by the server. Attentive readers may have noticed that this looks similar to classes calling methods of other classes, and this is actually the way in which it is modelled in many object-oriented programming languages as we will see later.

Depending on the application, the server and the client can be perceived as the same program or as two different programs. For example, web servers (like Apache) and web browsers (like Mozilla Firefox or Google Chrome) are seen as different programs, and actually they are developed by

[1] In recent years, the P2P (peer to peer) model has become quite popular. In P2P applications, all computers act as client *and* server at the same time.

different teams of people. On the other hand, the clients and the servers of applications like Dropbox or of games like World of Warcraft are considered to be part of the same program. This is only a matter of perception: at a technical level, client and server are always two different processes with two different and unrelated memory spaces (stack and heap), usually on two different machines[2].

By metonymy, the words “client” and “server” are commonly used to refer to the machines where the server or the clients run. For example, depending on the context, the term “web server” may refer to the program that serves web pages (like Apache) or a physical computer where this program is running (like ’www.dcs.bbk.ac.uk’).

1.2 Remote Method Invocation (RMI)

There are many technologies to create network applications. Some popular ones include RPC, CORBA, RMI, REST, GWT, servlets, and web services. Most of them have several things in common. We are going to learn to use RMI (the acronym is for Remote Method Invocation).

RMI applications are based on the client–server paradigm. Servers are implemented by objects that expose (i.e. implement) some interface. Clients are commonly implemented as objects too. Servers and clients are usually living on different machines, so clients cannot execute the methods on the server objects directly; they would need a pointer to the object, and pointers can only exist inside their own memory space.

In order to be able to call methods on objects that do no exist on their own machine, clients ask the *registry* for a *stub* to the server. The registry is a special kind of program —a sort of white pages— where servers can register themselves, and then clients can ask for a reference to a server by providing a name. On request, the registry will provide a reference to a stub for those servers that have been registered.

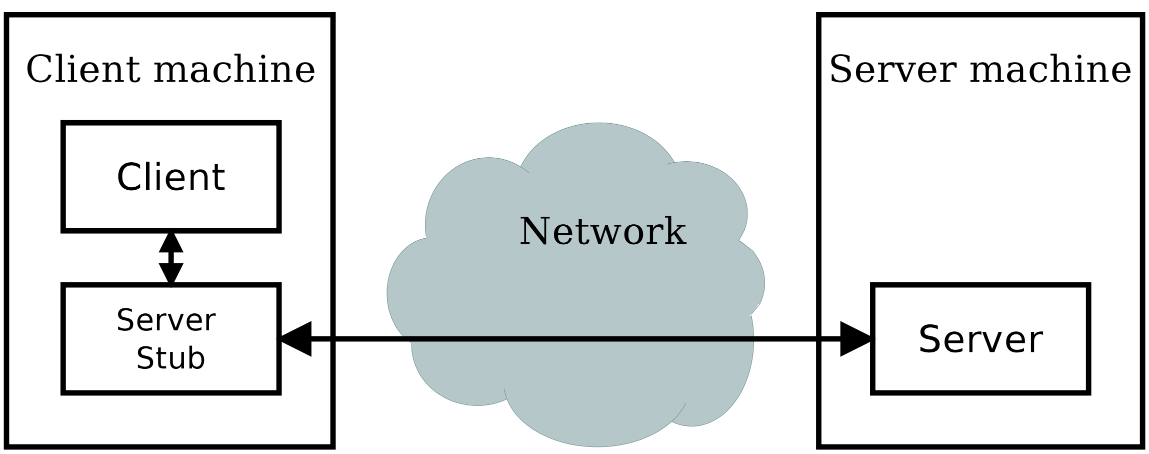

Figure 1: Flow of information (method calls, return values) in RMI

The stub implements the same interface (i.e. the same methods) as the server, so the client can call them (Figure 1) directly. However, the stub does not do any real computation: it “just”

[2]Some applications, including databases and email programs, have different client and server processes running on the same machine.

packs up the parameters provided by the client and sends them to the server through the network. The server receives those parameters, runs the appropriate code, and returns a result through the network to the stub. The stub can then return this result to the client (Figures 2, 3, 4). The good thing about RMI is that almost all the complexity of the network communication (opening up sockets, serializing objets as parameters, recovering from network errors, etc) is hidden from the programmer. From the point of view of the programmer, it looks (almost) like a normal method call.

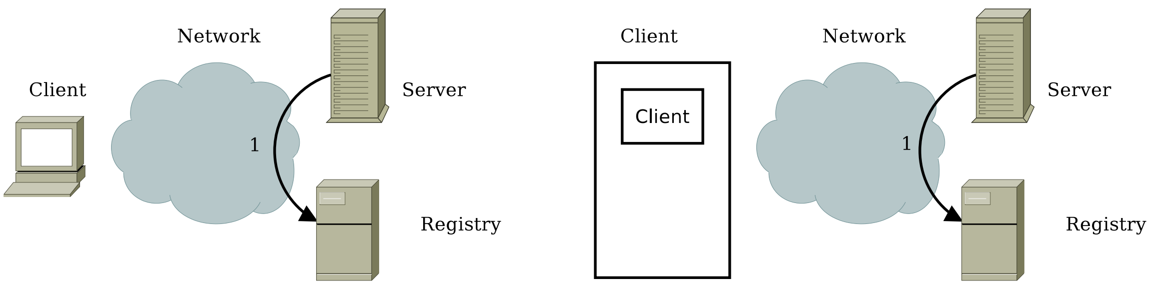

Figure 2: RMI step by step: (1) the server registers (binds) itself on the registry. The server and the registry are very often on the same physical machine.

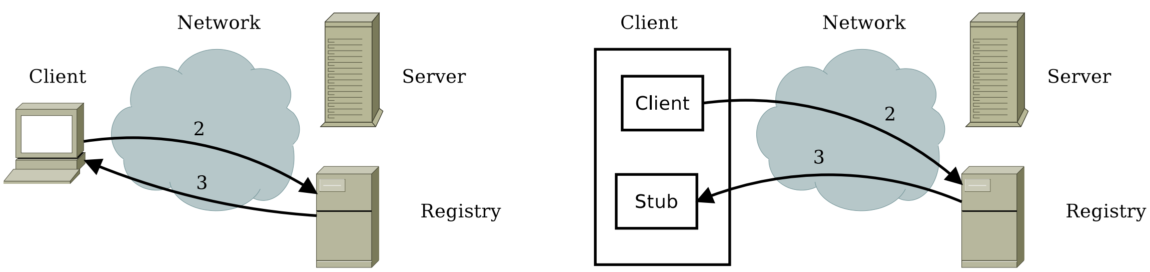

Figure 3: RMI step by step: (2) the client requests a stub for a server (as identified by a name) from the registry, and (3) the registry provides a reference to a stub of a server object if it has registered with the right name already.

1.3 Offering a remote service through RMI

The first step to offer some service using RMI is to define the interface of the service. This is a normal Java interface, with two additional details: it must extend `java.rmi.Remote`, and its methods must throw `java.rmi.RemoteException` (in case anything goes wrong with network connectivity).

If we had a very simple service called `echo` that just returned the same string that was provided as a parameter[3], its interface may look like this:

[3]Such a service actually exists, and it used to be employed to verify network connectivity. A modern parody / homage can be accessed at www.ismycomputeron.com.

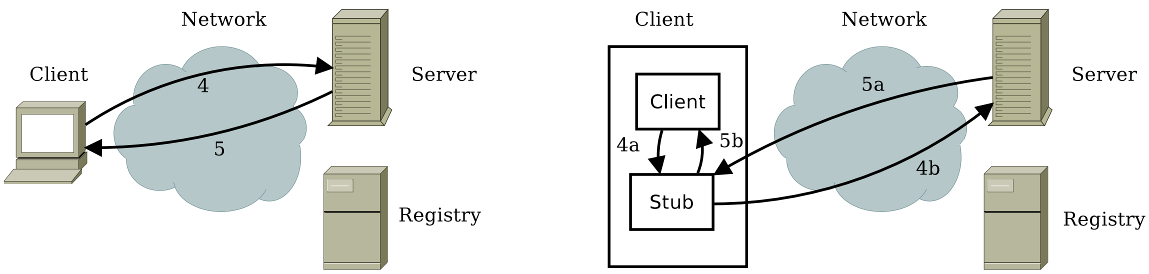

Figure 4: RMI step by step: (4) the client —by means of the stub— calls a method of the server, and (5) gets a return value.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * An implementation of the echo service.
 */
public interface EchoService extends Remote {
    /**
     * Returns the same string passed as parameter
     * @param s a string
     * @return the same string passed as parameter
     */
    public String echo(String s) throws RemoteException;
}
```

All objects used as parameters or return values in a RMI interface must implement the interface `java.io.Serializable`. This is a tagging interface —an interface with no methods— that indicates that Java should be able to "flat down" the object into bytes (this is called *serializing*) and then reconstruct the object again at the other end (this is called *marshalling*). Usual data types from Java (e.g. Integer, String, etc) implement `Serializable`.

1.4 Implementing the service

The server that provides the service, as we have mentioned, is implemented like a normal Java object. . . with two additional details. First, the server object must extend `UnicastRemoteObject` (this means that it cannot extend another class!). Second, an explicit constructor must be provided —even if empty— that throws `RemoteException`

The implementation of the interface above could look like (comments ommitted for brevity):

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

public class EchoServer extends UnicastRemoteObject implements EchoService {

    public EchoServer() throws RemoteException {
        // nothing to initialise for this server, but it is important
        // to declare that the constructor throws RemoteException
    }

    @Override
    public String echo(String s) {
        // This println() is not necessary, but helps verifying whether
        // the server has received the call or not on the remote machine
        System.out.println("Replied to some client saying '" + s + "'");
        return s;
    }
}
```

We can now compile the server and create the stub to be sent to clients. Compiling the server requires a normal Java compilation:

```
    javac EchoServer.java
```

This will produce an `EchoServer.class` file, as we know.

From Java 8 on, we do not need to do anything special to create the stubs that implement the communication between server and client. These are generated on demand by Java[4].

1.5 Launching the server

Launching a server consists of a series of steps that must be followed in sequence. They are shown in the following code (this may be in a different class, maybe in a file called `EchoServerLauncher.java`):

```
    private void launch() {
        try {
            // 1. Create the registry if there is not one
            LocateRegistry.createRegistry(1099);
            // 2. Create the server object
```

[4]If you work in the future with Java 7 (or earlier) codebases that use RMI you will learn to use a program called 'rmic'. It is not difficult with your Java current knowledge, only a bit cumbersome.

```
        EchoServer server = new EchoServer();
        // 3. Register (bind) the server object on the registy.
        //    The registry may be on a different machine
        String registryAddress = "localhost";
        String registryUrl = "//" + registryAddress + "/";
        String serviceName = "echo";
        Naming.rebind(registryUrl + serviceName, server);
    } catch (MalformedURLException ex) {
        ex.printStackTrace();
    } catch (RemoteException ex) {
        ex.printStackTrace();
    }
}
```

The registry will listen on port 1099 by default, but this can be changed (e.g. to pass through a firewall that forbids use of port 1099). Then an instance of the server class is created and registered (i.e. bound) with a given name. This name must be known by clients that want to use the services offered by the server.

1.6 Using the services from the client

We are almost there. The only piece missing is the client!

The first thing that the client needs to do is to find a reference to the remote server object. It can do so by asking the registry:

```
String registryAdress = "127.0.0.1";
Remote service = Naming.lookup("//" + registryAddress + ":1099/echo");
EchoService echoService = (EchoService) service;
```

Assuming the name is right —if not, a `MalformedURLException` will be thrown— and a server has registered with the right name (in this case, “echo”) —otherwise, the registry will not find it and will throw a `NotBoundException`—, the registry will return the stub, which is an object of type `Remote`. In order to use the methods in interface `EchoService`, the client must downcast it explicitly to the right type. Once this is done, using the service is as easy as a normal method call:

```
String receivedEcho = echoService.echo("Hello!");
```

This simple line will fire steps 4 and 5 on Figure 1: the stub will take string “Hello!”, convert it into bytes, and send them to the server. The server will reconstruct the string, execute the method, and send the return value back to the stub. The stub will give the return value to the client, who will store it in variable `receivedEcho`. If anything goes wrong at any point, a `RemoteException` will be thrown.

The client must have access (on its CLASSPATH) to the classes needed to serialize and marshall the objects sent and received to the server. This includes those classes used for parameters and return values[5].

[5] If the client does not have access to all the classes required for the service, and need to download new classes from the server, this can constitute a security hazard. In these cases, a security policy must be set up. This goes beyond the scope of these notes.

1.7 Summary

The steps to be followed on the server side are:

1. Define the interface of the service.
2. Implement the service.
3. Compile the server.
4. Write a launcher that binds the server to the registry, and execute it.

The client needs to look up the server at the registry (by providing a name). Once it has a reference to it (through the stub), it can call the methods in the interface like it was a local object.

2 Conclusion

In the modern world, programs are no longer confined to a single machine, and many programs actually run on several machines at the same time. Some popular examples include web servers and web browsers, bitTorrent programs, distributed storage programs, and version control programs like Git.

Most networking programs are based on the client–server paradigm, including P2P programs in which every computer can act as a server or a client at the same time. Servers expose or offer services and clients make requests for those services, e.g. asking for a webpage or pushing the latests commits.

We have seen how RMI works. RMI is a specific Java technology that we have chosen because it is a core Java component (i.e. it does not need external libraries) and it suffices to illustrate most of the main networking concepts. It is also important because many other Java network technologies are built on top of RMI.

The concepts we have learned are common to any network technology you will use in the future: client, server, registry, and remote method invocation. Understanding how to implement these concepts in RMI will make it easy to learn and use other technologies in the future, such as REST and web services.