Client-server model

Daniel Hagimont

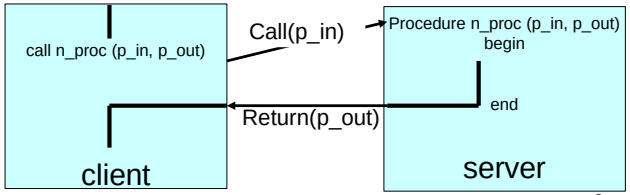
IRIT/ENSEEIHT
2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7

Daniel.Hagimont@enseeiht.fr
http://hagimont.perso.enseeiht.fr

This lecture is about the client server model. It reviews the concepts and illustrates them with its instanciation in the Java environment, with Remote Method Invocation (RMI).

Client-server model based on message passing

- Two exchanged messages (at least)
  - The first message corresponds to the request. It includes the parameters of the request.
  - The second message corresponds to the response. It includes the result parameters from the response.



Client-server interactions can be implemented with message passing (using sockets).

You then have at least 2 messages exchanged for such an interaction.

The first message corresponds to the request, including parameters, and the second message corresponds to the response, including result parameters.

The client's execution is suspended after sending of the request, until reception of the response.

We can observe that such an interaction looks like a procedure call, except that the caller (client) and the callee (server) are located on different machines.

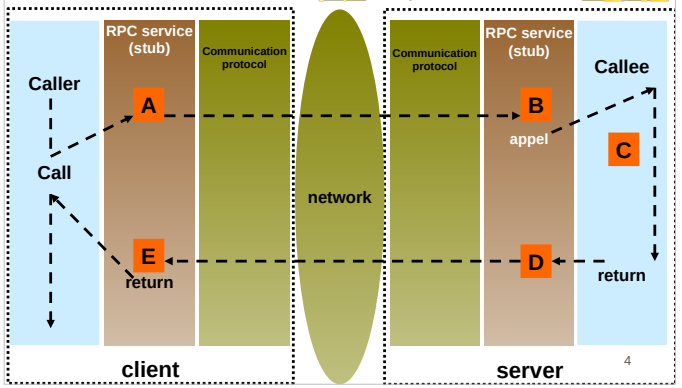
Remote Procedure Call (RPC) Principles

- Generating most of the code
  - Emission and reception of messages
  - Detection and re-emission of lost messages
- Objectives: the developer should be able to program the application without the burden to deal with messages

We call RPC (Remote Procedure Call) a tool which simplifies the development of applications relying on such client-server interactions, by generating the code which implements message exchanges (requests and responses). The idea is that all this code can be generated from a description of the interface of the procedure (which can be called on the server from a remote client).

The objective is to allow the developer to program and test his application as if it was centralized (executed on one machine) without the burden to deal with message exchanges. The code enabling the application to be distributed can be generated and the code of the application is kept simple.

RPC [Birrel & Nelson 84] Implementation principle



This is the general principle of RPC tools.

In blue, you have 2 code segments, the caller in the client which invokes (call) a service, and the callee in the server which provides the service.

In a centralized environment, the call would be a simple procedure call between the caller and the callee.

The principle of a RPC tool is to generate 2 code segments (in brown) called the client stub (left) and the server stub (right).

The client stub represents the service on the client machine and gives the illusion that the service is local (can be invoked locally with a simple procedure call). The client stub implements the same procedure as the server in order to give this illusion. A call to the procedure in the client stub creates and sends a request message to the server. The server stub receives and transforms request messages into local procedure calls on the server machine.

RPC (point A)
Implementation principle

- On the caller side
  - The client makes a procedural call to the client stub
    - The parameters of the procedure are passed to the stub
  - At point A
    - The stub collects the parameters and assembles a message including the parameters (parameter marshalling)
    - An identifier is generated for the RPC call and included in the message
    - A watchdog timer is initialized
    - Problem: how to obtain the address of the server (a naming service registers procedures/servers)
    - The stub transmits the message to the transport protocol for emission on the network

5

On the caller side, the client performs a procedure call (invoking the service) as if the service was local to the client machine. Notice that the client stub implements the same procedure as the server, but the implementation of that procedure is different.

At point A, the client stub is called and receives the parameters from the procedure call. It assembles a request message which includes these parameters (this step is called parameter marshalling). An identifier for this RPC call is generated and included in the request message. This identifier allows to detect on the server side the reception of 2 requests for the same call (if the message is supposed to be lost and re-emitted).

A watchdog timer is initialized. It wakes up after a given time. If we don't receive a response before the wakeup, we consider that the request was lost and the request is re-emitted.

One problem here is to get the address (IP/port) of the server process for sending requests. Generally a naming service allows to register available procedures and their addresses.

The stub can then send the request message with the communication protocol (generally UDP as the data to be transmitted is not large).

The client is then suspended, waiting for the response message.

RPC (points B et C)
Implementation principle

- On the callee side
  - The transport protocol delivers the message to the RPC service (server stub)
  - At point B
    - The server stub disassembles the parameters (parameter unmarshalling)
    - The RPC identifier is registered
  - The call is then transmitted to the remote procedure which is executed (point C)
  - The return from the procedure returns back to the server stub which receives the result parameters (point D)

6

On the callee side, the communication protocol delivers the request message to the server stub. At point B, the server stub disassembles the parameters of the call (this step is called parameter unmarshalling). The RPC identifier is registered to detect redundant requests for the same call.

The call is then reproduced, i.e. the procedure to be called in the callee is actually called (point C). This is a normal procedure call. On return, the procedure returns back (point D) to the server stub (with some result parameters).

RPC (point D)
Implementation principle

- On the callee side
  - At point D
    - The result parameters are assembled in a message
    - Another watchdog timer is initialized
    - The server stub transmits the message to the transport protocol for emission on the network

At point D, the server stub assembles the result parameters in a response message.

Another watchdog timer is initialized. It wakes up after a given time. If we don't receive an acknowledgment from the client (that the response was received) before the wakeup, we consider that the response was lost and the response is re-emitted.

The server stub can then send the response with the communication protocol.

RPC (point E)
Implementation principle

- On the caller side
  - The transport protocol delivers the response message to the RPC service (client stub)
  - At point E
    - The client stub disassembles the result parameters (parameter unmarshalling)
    - The watchdog timer created at point A is disabled
    - An acknowledgment message with the RPC identifier is sent to the server stub (the watchdog timer created at point D can be disabled)
    - The result parameters are transmitted to the caller with a procedure return

On the caller side, the communication protocol delivers the response message to the client stub.

At point E, the client stub disassembles the result parameters (parameter unmarshalling).

The watchdog timer created at point A can be disabled.

An acknowledgment message with the RPC identifier is sent to the server stub (the watchdog timer created at point D can be disabled).

The result parameters are transmitted to the caller with a procedure return.

RPC
Role of stubs

Client stub

- It is the procedure which interfaces with the client
  - Receives the call locally
  - Transforms it into a remote call with a sent message
  - Receives results in a message
  - Returns results with a normal procedure return

Server stub

- It is the procedure on the server node
  - Receives the call as a message
  - Performs the procedure call on the server node
  - Receives the results of the call locally
  - Transmits the results remotely as a message

We summarize here the role of the client stub and the server stub.

RPC
Message loss

- On the client side
  - If the watchdog expires
    - Re-emission of the message (with the same RPC identifier)
    - Abandon after N attempts
- On the server side
  - If the watchdog expires
  - Or if we receive a message with a known RPC identifier
    - Re-emission of the response message
    - Abandon after N attempts
- On the client side
  - If we receive a message with a known RPC identifier
    - Re-emission of the acknowledgment message

We provide here a global view of the handling of message loss.

On the client side, we created a watchdog before sending the request. If this watchdog expires, we can suppose that the request was lost and we re-send the request with the same RPC identifier (we re-initialize the watchdog before sending). We abandon after N attempts, assuming that the network is down.

On the server side, we create a watchdog before sending the response. As previously, we re-send the response if the watchdog expires. Another case on the server side is when we receive a request with a known RPC identifier (requests are logged). This means that we already received this request and the procedure was called and the response sent, but the response was lost. Then we re-send the response. As previously, we abandon after N attempts.

Finally, on the client side, if we receive a response with a known RPC identifier (response are logged), i.e. a response that we already received, it means that the acknowledgment sent to the server was lost and we re-send it.

RPC
Problems

- Failure handling
  - Network or server congestion
    - The response arrives too late (critical systems)
  - The client crashes during the request handling on the server
  - The server crashes during the handling of the request
  - Failure of the communication system
  - What guarantees ?
- Security problems
  - Client authentication
  - Server authentication
  - Privacy of exchanges
- Performance
- Designation
- Practical aspects
  - Adaptation to heterogeneity conditions (protocols, languages, hardware)

Many other problems can be handled by RPC systems.

The handling of failures covers many types of failure :

- dealing with network or server congestion. Messages may be re-emitted, but redundant messages must be managed. In a real-time system, the execution time of a procedure is specified and the procedure should return an error if the deadline is not respected.

- dealing with the crash of the client or the server during the handling of the request, or the failure of the communication system. The system should provide guarantees (e.g. transactional behavior).

A RPC tool may also integrate security features, like authentication and encryption of exchanges.

Many other aspects were also considered :

- performance of RPC, especially the optimization when the client and server processes are on the same machine, or on the same LAN.

- designation : different designation scheme can be provided, for identifying the target (process) of call.

- heterogeneity : a lot of work was done to enable heterogeneity (of languages, OS …) between the caller and callee (see CORBA).

RPC
IDL : interface specification

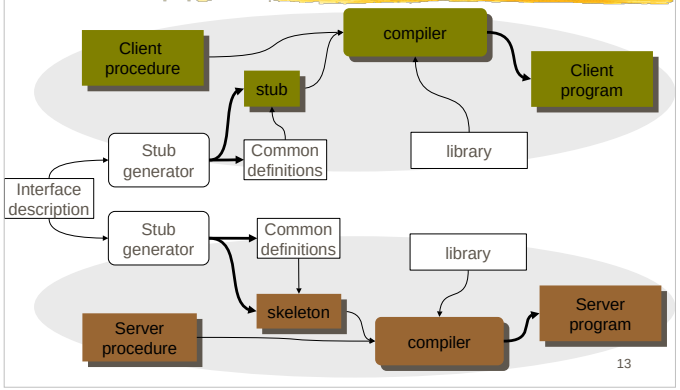
- Use of an interface description language (IDL)
  - Specification which is common to the client and the server
  - Definition of parameter types et natures (IN, OUT, IN-OUT)
- Use of the IDL description to generate:
  - The client stub (also called proxy or stub)
  - The server stub (also called skeleton)

Generally, a RPC tool generates stubs from the specification of the interface of the procedure which can be called remotely.

An IDL (Interface Description Language) is a simple language for describing the interface of a procedure which can be called through a RPC system. It simply allows describing the signature of the procedure, including the type of the parameters (data structures).

Such a specification allows to generate the client stub (sometimes called proxy or simply stub) and the server stub (often called skeleton).

RPC
Functional mode (rpcgen)



rpcgen is one of the first RPC tools which was available in a Unix/C environment.

From the interface description (expressed with the IDL), a stub generator generates both the sub and skeleton.

On the client side, the client procedure (caller) is compiled with the stub in order to obtain an executable binary (client program).

On the server side, the server procedure (callee) is compiled with the skeleton in order to obtain a executable binary (server program).

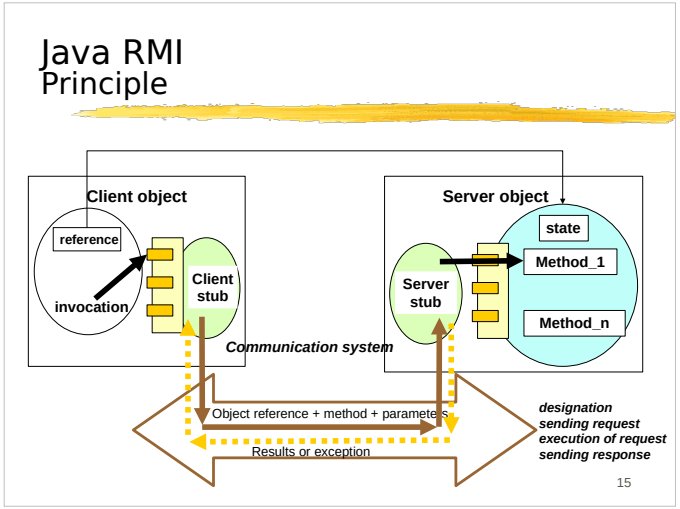
These 2 binaries can be installed on different machines and executed.

Java Remote Method Invocation
RMI

- An object based RPC integrated within Java
- Interaction between objects located in different address spaces (*Java Virtual Machines* - JVM) on remote machines
- Easy to use: a remote object is invoked as if it was local

Java RMI (Remote Method Invocation) is an example of implementation of a RPC tool integrated in a language environment (here Java).

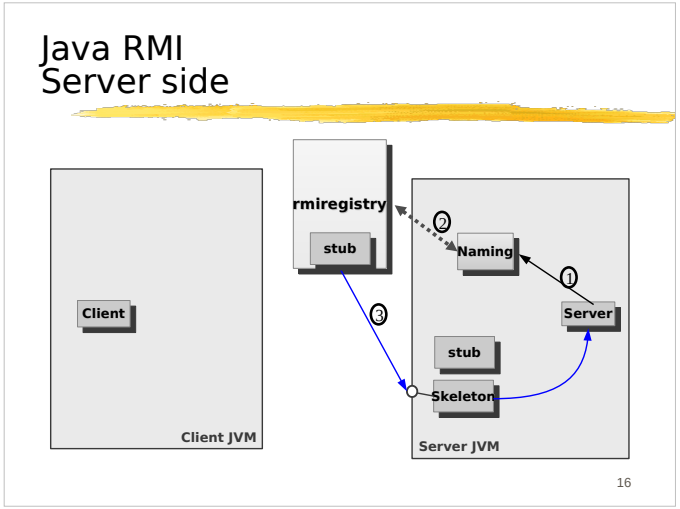
It allows the invocation of methods on instances located on remote machines (in a remote JVM). Such a remote method invocation is programmed as if the target object was local to the current JVM.



The general principle of Java RMI is illustrated in this figure.

A client object in one JVM (left) includes a reference to a server object (remote) in another JVM (right).

This reference is actually a reference to a local stub object (client stub). This stub transforms a method call into a request message (which includes an object reference to identify the object in the server JVM, an method identifier and the parameters of the call). This request message is received by a skeleton object (server stub) which performs the actual method call on the server object.



We describe the general functioning the RMI before describing its programming model.

We assume that a Server class has been programmed following the RMI programming model.

On the server side, when the Server class is instantiated, stub and skeleton objects are instantiated The skeleton object is associated with a local port of the machine for receiving requests.

In order to make the Server object accessible from clients, it must be registered in a naming service called rmiregistry (the rmiregistry runs in another JVM). This registration is possible thanks to the Naming class which provides a bind method (which registers the association between a name ("foo") and the Server object).

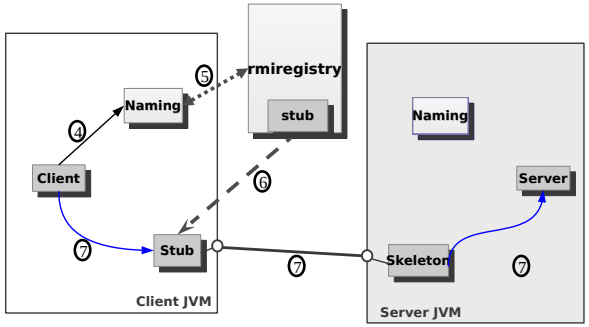
This registration in the rmiregistry makes a copy of the stub in the rmiregistry (and registers its association with "foo"). The rmiregistry is ready to deliver copies of the stub to clients.

Java RMI
Server side

- 0 – At object creation time, a *stub* and a *skeleton* (with a communication port) are created on the server
- 1 – The server registers its instance with a naming service (*rmiregistry*) using the *Naming* class (*bind* method)
- 2 – The naming service (*rmiregistry*) registers the *stub*
- 3 – The naming service is ready to give the *stub* to clients

We recall here the main step of creation of a Server object and registration in the rmiregistry.

Java RMI
Client side



On the client side, the client can fetch a reference to the Server object from the rmiregistry. This is possible thanks to the Naming class which provides a lookup method (which queries the object registered with a name ("foo")).

The query on the rmiregistry returns a copy of the stub (associated with "foo"). This stub implements the same interface as the Server object. It can be used by the client to invoke a method. The stub creates and sends a request message to the skeleton which performs the actual call on the Server object.

Java RMI
Client side

- 4 – The client makes a call to the naming service (*rmiregistry*) using the *Naming* class to obtain a copy of the stub of the server object (*lookup* method)
- 5 – The naming service delivers a copy of the *stub*
- 6 - The *stub* is installed in the client and its Java reference is returned to the client
- 7 – The client performs a remote invocation by calling a method on the *stub*

We recall here the main step of querying the rmiregistry and performing a method invocation.

Java RMI
Utilization

- Coding
  - Writing the server interface
  - Writing the server class which implements the interface
  - Writing the client which invokes the remote server object
- Compiling
  - Compiling Java sources (javac)
  - Generation of *stubs* et *skeletons* (rmic)
    - *(not required anymore, dynamic generation)*
- Execution
  - Launching the naming service (*rmiregistry*)
  - Launching the server
  - Launching the client

Here are the main steps for using RMI.

Regarding coding :

- you must define the Java interface of the Server. This interface is used both by the Server and the Client.

- the Server class implements the previous interface. The Server is instantiated and the instance is registered in the rmiregistry.

- the Client can declare a variable whose type is the previous interface. The Client obtains a copy of the stub from the rmiregistry. The stub implements the interface. The Client can call a method on this stub.

Regarding compiling :

- the application is compiled with javac as usually

- the stub and skeleton classes can be generated with rmic (a stub generator). This is not necessary anymore on recent versions of Java, the stubs being generated dynamically when needed.

Regarding execution :

- you have to launch the rmiregistry

- then you can launch the server and then the client

Java RMI Programming

- Programming a remote interface
  - public interface
  - interface: extends java.rmi.Remote
  - methods: throws java.rmi.RemoteException
  - serializable parameters: implements Serializable
  - references parameters: implements Remote
- Programming a remote class
  - implements the previous interface
  - extends java.rmi.server.UnicastRemoteObject
  - same rules for methods

Programming RMI applications comes with programming constraints.

For the interface of the Server :

- the interface must be public

- the interface must implement the Remote interface

- all the methods must throw RemoteException

- parameters of remote methods can be of built-in type (int, char), or a Java reference. In this last case, their type must be an interface which is either Serializable or Remote (this is detailed later).

For the Server class :

- it must implement the previous interface

- it must extend the UnicastRemoteObject class

- same rules for methods (as in the previous interface)

Java RMI Example: interface

file Hello.java

```
public interface Hello extends java.rmi.Remote {
  public void sayHello()
        throws java.rmi.RemoteException;
}
```

Description of the interface

We review a very simple example.

Here is the definition of the interface.

Interface Hello implements Remote and throws RemoteException.

Java RMI
Example: server

file HelloImpl.java

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

public class HelloImpl extends UnicastRemoteObject
                                implements Hello {
    String message;

   // Constructor implementation
   public HelloImpl(String msg) throws java.rmi.RemoteException {
         message = msg;
   }
  // Implementation of the remote method
   public void sayHello() throws java.rmi.RemoteException {
          System.out.println(message);
   }

      ...
```

Implementation of the server class

Here is the code of the server class.

Class HelloImpl extends UnicastRemoteObject and implements interface Hello.

Your constructors must throw RemoteException.

The remote method sayHello() throws RemoteException.

Java RMI
Example: server

file HelloImpl.java

```
    ...

  public static void main(String args[]) {
        try {
                // Create an instance of the server object
                Hello obj = new HelloImpl("hello");
                // Register the object with the naming service
                Naming.rebind("//my_machine/my_server", obj);
                System.out.println("HelloImpl " + " bound in registry");
        } catch (Exception exc) {... }
    }
}
```

Implementation of the server class

NOTICE : in this example, the naming service (rmiregistry) must have been launched before execution of the server

The rest of the code of the server.

The main method creates an instance of the server class (HelloImpl) and registers it in the rmiregistry, thanks to the Naming class.

The URL passed in the rebind() method is //<machine-name>:<port>/<name>

- machine-name is the name of the machine which runs the rmiregistry

- port is the port used by the rmiregistry (the default port is 1099)

- name is the name identifying the registered object in the rmiregistry

In its implementation in Java, the rmiregistry has to be colocated (on the same machine) with the JVM which runs the server object. A work around is to implement another rmiregistry (allowing remote registrations).

Notice that after the registration, this is the end of the main method and the JVM would exit. This is not the case, since when we instantiated the server object, a skeleton was instantiated with creation of a communication socket and of a thread waiting for incoming requests. Because of that thread, the JVM does not exit.

Here, we assume that the rmiregistry was launched (rmiregistry is an executable) on the same machine as the server object, with the command :

rmiregistry <port> (default is 1099)

Java RMI
running the rmiregistry within the server JVM

file HelloImpl.java

```
public static void main(String args[]) {
  int port;    String URL;

  try {
    Integer I = new Integer(args[0]); port = I.intValue();
  } catch (Exception ex) {
    System.out.println(" Please enter: java HelloImpl <port>"); return;
  }

  try {
    // Launching the naming service - rmiregistry - within the JVM
    Registry registry = LocateRegistry.createRegistry(port);

    // Create an instance of the server object
    Hello obj = new HelloImpl();

    // compute the URL of the server
    URL = "//"+InetAddress.getLocalHost().getHostName()+":"+
                    port+"/my_server";
    Naming.rebind(URL, obj);
  } catch (Exception exc) { ...}
}
```


In this other version, we launch a rmiregistry in the same JVM as the one hosting the server object.

The createRegistry method launches a rmiregistry within the local JVM on the specified port.

The interest of doing so is that when you start the application, a rmiregistry is automatically launched and when you kill the JVM, the rmiregistry is also killed. This is very convenient when debugging.

Java RMI
Example: client

file HelloClient.java

Implementation of the client class

```
import java.rmi.*;

public class HelloClient {
  public static void main(String args[]) {
    try {
      // get the stub of the server object from the rmiregistry
      Hello obj = (Hello) Naming.lookup("//my_machine/my_server");
      // Invocation of a method on the remote object
      obj.sayHello();
    } catch (Exception exc) { ... }
  }
}
```

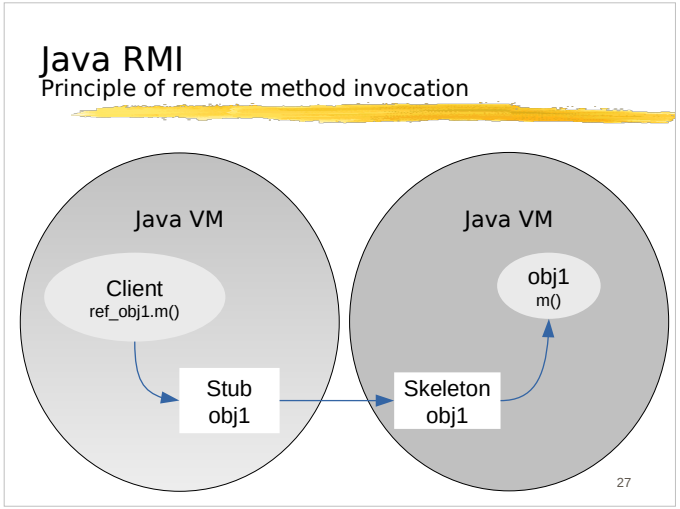

Here is the code on the client side.

It first requests a reference to the target object from the rmiregistry, using the lookup method from the Naming class (the used URL is the same as before.

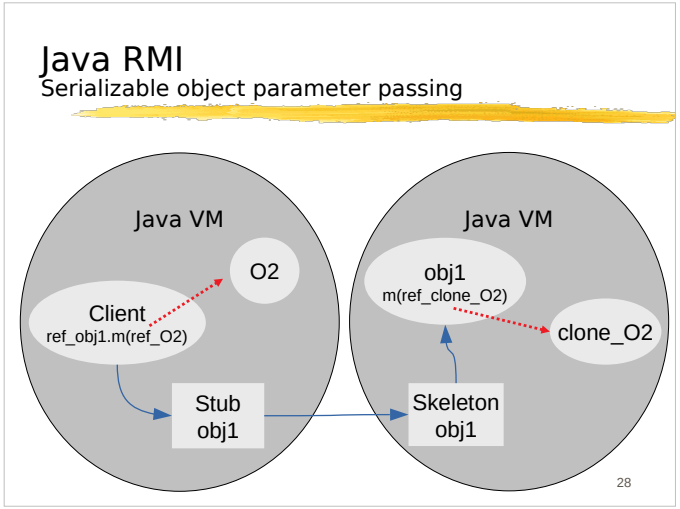
Notice that here the client can be executing on a different machine.

The rmiregistry returns a stub instance. This stub instance implements the same interface as the server object (here Hello). So we can cast the obtained reference with the Hello interface.

Then, invoking a method on the remote object is programmed as if the object was local.



To summarize the functioning of Java RMI, a client which obtained (from the rmiregistry) a remote reference (ref_obj1) to a remote object (obj1) has actually a reference to a local stub object (Stub obj1). The client can invoke a method m() on the remote object. It will invoke this method on the stub, which will send the request message. This message is received by the skeleton (Skeleton obj1) which performs the actual invocation on the server object.



Parameters passed in a remote method can be of built-in types (int char …). Then the parameters are simply copied (transferred) in the remote server.

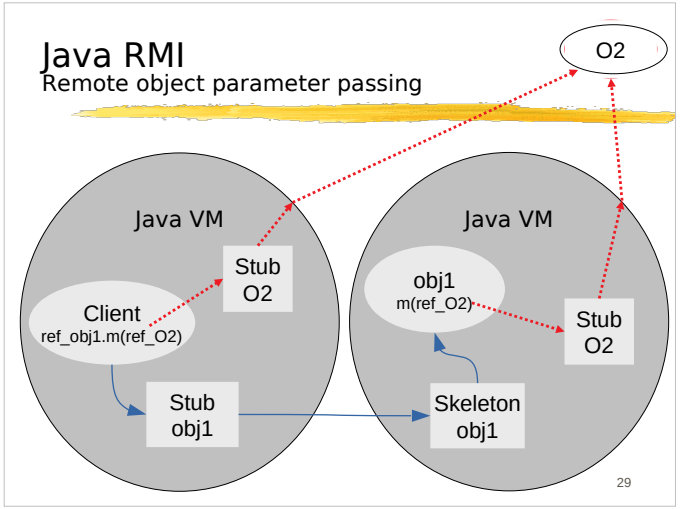
If a parameter is a Java reference to an object, the type of the parameter in the method signature must be an interface. Then, there are 2 possibilities:

- Serializable. If the interface is serializable (inherits from Serializable), then the passed object is copied to the server (the object is cloned).

- Remote. If the interface is Remote (inherits from Remote), then the remote reference (i.e. the stub) is passed to the server, meaning that the stub is copied in the server. Therefore, the passed object becomes accessible remotely in the server.

- if the interface is neither Serializable nor Remote, this is an error (it should not compile).

This figure illustrates the Serializable case. The client passes as parameter a reference to object O2 which is local in the client. Then, O2 is copied to the server and the invoked method (m) receives a reference to a clone of object O2 in the server.



This figure illustrates the Remote case. The client passes as parameter a reference to object O2 which is remote (in another JVM). It means that the reference to O2 in the client is a local reference to a sub of O2. Then, the stub of O2 is copied to the invoked server and the invoked method (m) receives a reference to a copy of stub of O2 in the server. Therefore, m() receives a remote reference to O2.

Java RMI
Compiling

- Compiling the interface, the server and the client
  - javac Hello.java HelloImpl.java HelloClient.java
- Generation of stubs (*not needed anymore*)
  - rmic HelloImpl
    - *skeleton* in HelloImpl_Skel.class
    - *stub* in HelloImpl_Stub.class

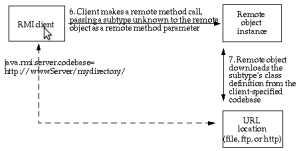
30

To execute the application, you must first compile the interface and the server and client classes.

As previously mentioned, generating stubs and skeletons is not necessary anymore, but you can still do it.

Java RMI
Deployment

- Launching the naming service
  - rmiregistry &
- launching the server
  - java HelloImpl
  - java -Djava.rmi.server.codebase=http://my_machine/…
    - URL of a web server from which the client JVM will be able to download missing classes
    - Example: serialization
- Launching the client
  - java HelloClient



Here we explicitly launch the rmiregistry in a shell.

Then, we can launch the server and then the client.

One tricky issue is the availability of classes. Assume the client invokes a method m(Data d) on the server, Data being an interface which is Serializable. Both the client and the server know the interface Data (it was necessary to use the method m and to compile the code). Then the client may invoke m passing an instance of class ClientData (which implements Data). But the server which receives a copy of the object does not have the ClientData class (and different clients may have different implementations of the Data interface).

More generally, a JVM may transfer copies of objects (with Serialization) to other JVMs. How can the first JVM make these classes available to other JVMs. The solution is to specify, when launching a JVM, a web site from which classes can be downloaded. When classes are missing for using a serialized object, the classes are downloaded and installed dynamically.

java -Djava.rmi.server.codebase =URL <a class>

When launching a JVM this way, we specify that if serialized instances are given to other JVMs, the missing classes can be found on the web site defined by URL.

Java RMI: conclusion

- Very good example of RPC
  - Easy to use
  - Well integrated within Java
  - Java reference parameter passing: serialization or remote reference
  - Deployment: dynamic loading of serializable classes
  - Designation with URL

Many tutorials about RMI programming on the Web …
Example : https://www.javatpoint.com/RMI

To conclude this lecture, Java RMI is a very example of RPC integrated in a Java.

Many tutorials about Java RMI can be found on the net.