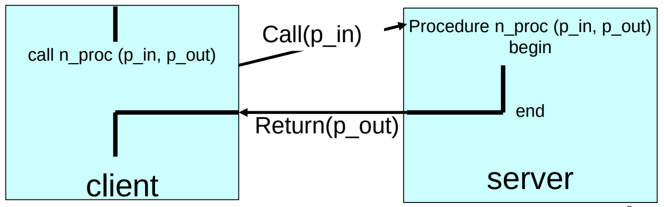
Client-server model based on message passing

- 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).

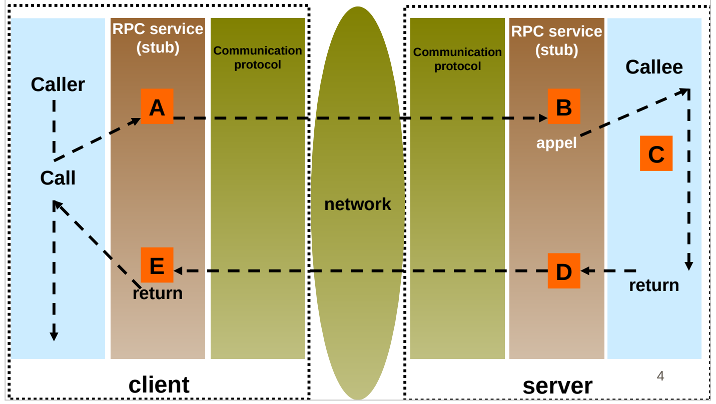
You then have at least 2 messages exchanged for such an interaction.

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.

RPC [Birrel & Nelson 84] Implementation principle



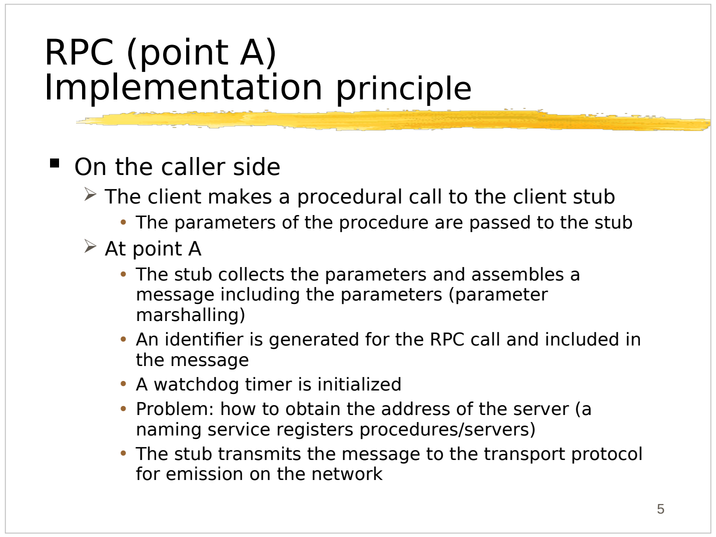
This is the general principle of RPC tools.

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

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.

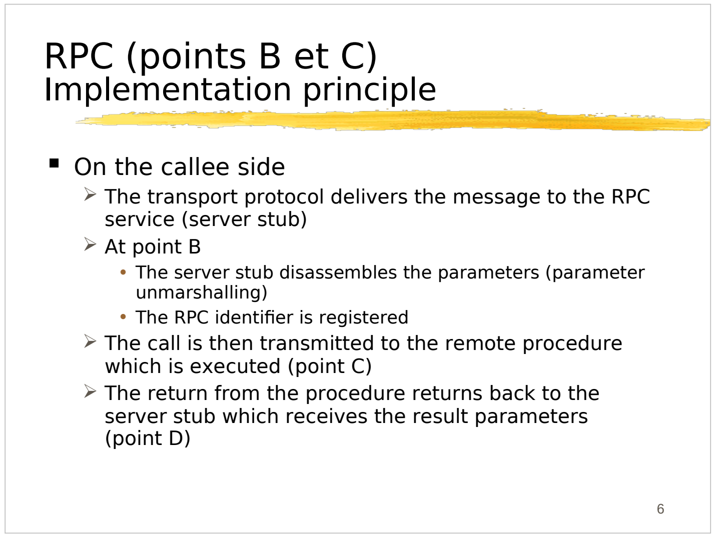
At point A, the client stub is called and receives the parameters from the procedure call. It assembles a request message which includes these parameters (this step is called parameter marshalling). An identifier for this RPC call is generated and included in the request message. This identifier allows 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.

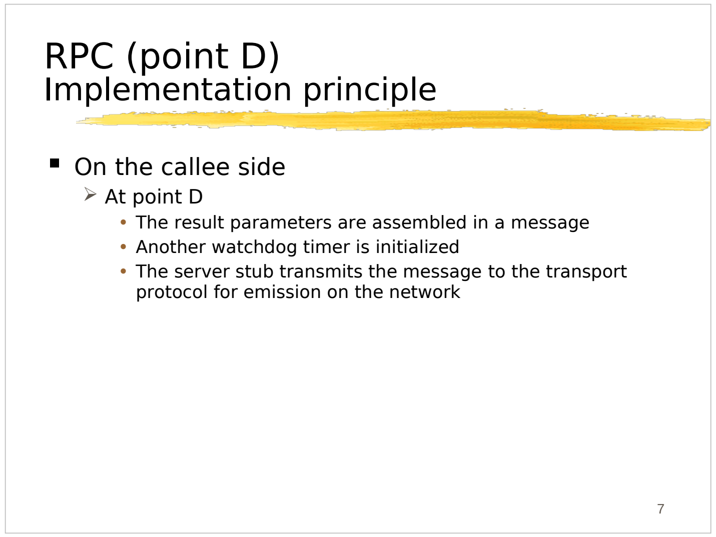


RPC (points B et C)
Implementation principle

- 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)

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

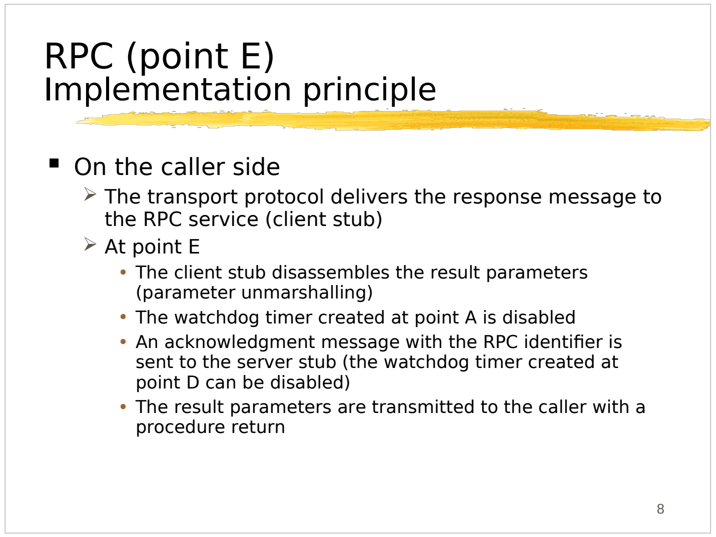
- On the callee side
  - At point D
    - The result parameters are assembled in a message
    - Another watchdog timer is initialized
    - The server stub transmits the message to the transport protocol for emission on the network

7

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

8

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

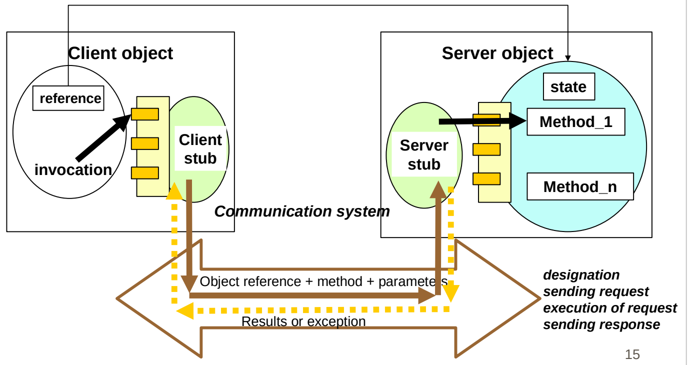
At point E, the client stub disassembles the result parameters (parameter unmarshalling).

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.

Java RMI
Principle



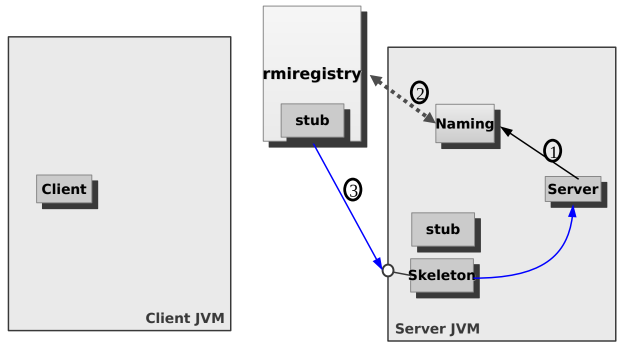
15

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.

Java RMI
Server side



16

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

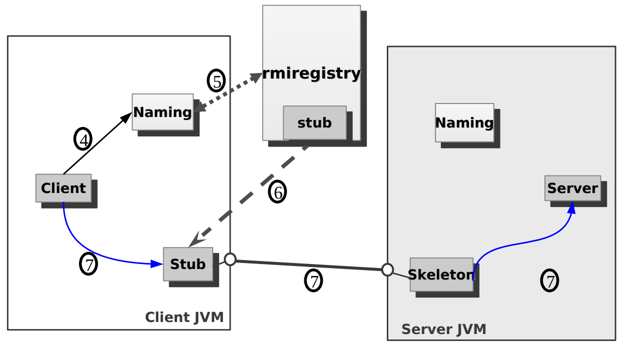
We assume that a Server class has been programmed following the RMI programming model.

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
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
Example: interface

file Hello.java

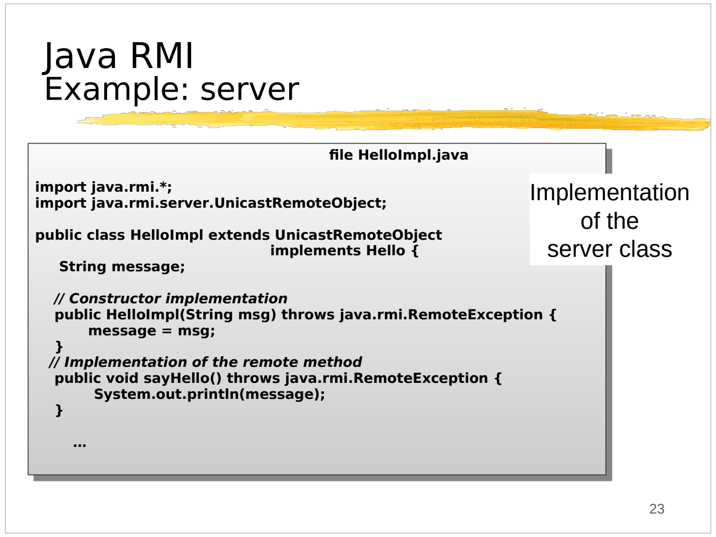
```
public interface Hello extends java.rmi.Remote {
  public void sayHello()
          throws java.rmi.RemoteException;
}
```

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

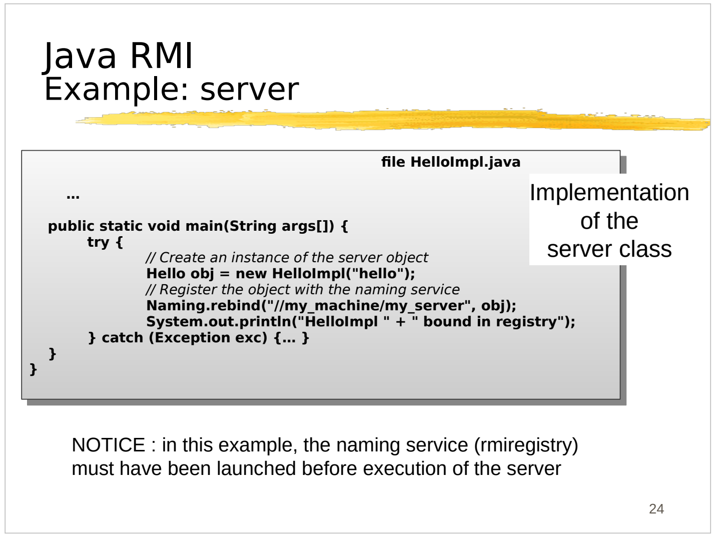
23

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

24

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)

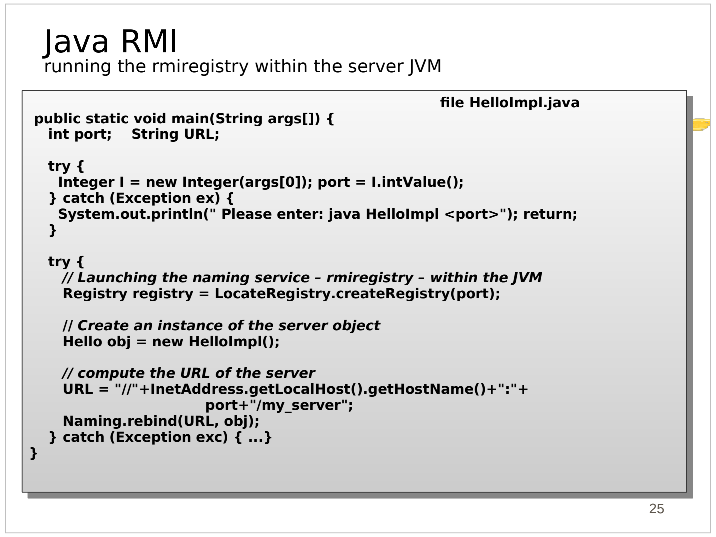
- name is the name identifying the registered object in the rmiregistry

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

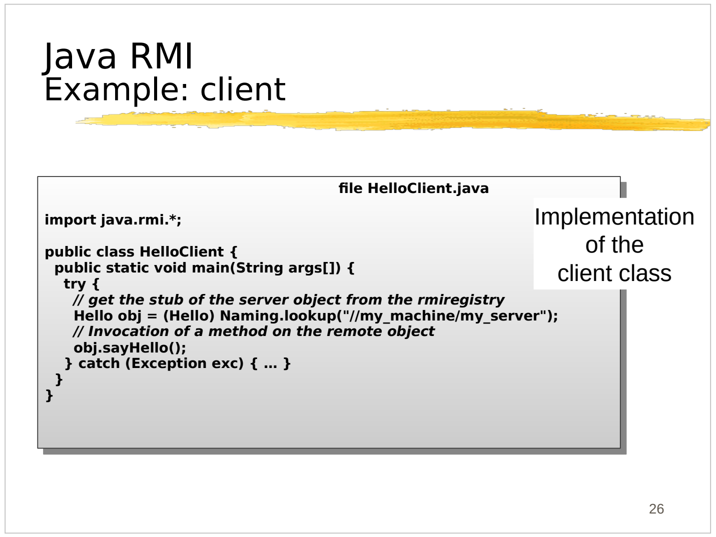
file HelloImpl.java

```
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

```
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) { … }
  }
}
```

Implementation
of the
client class

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.