Client-server model

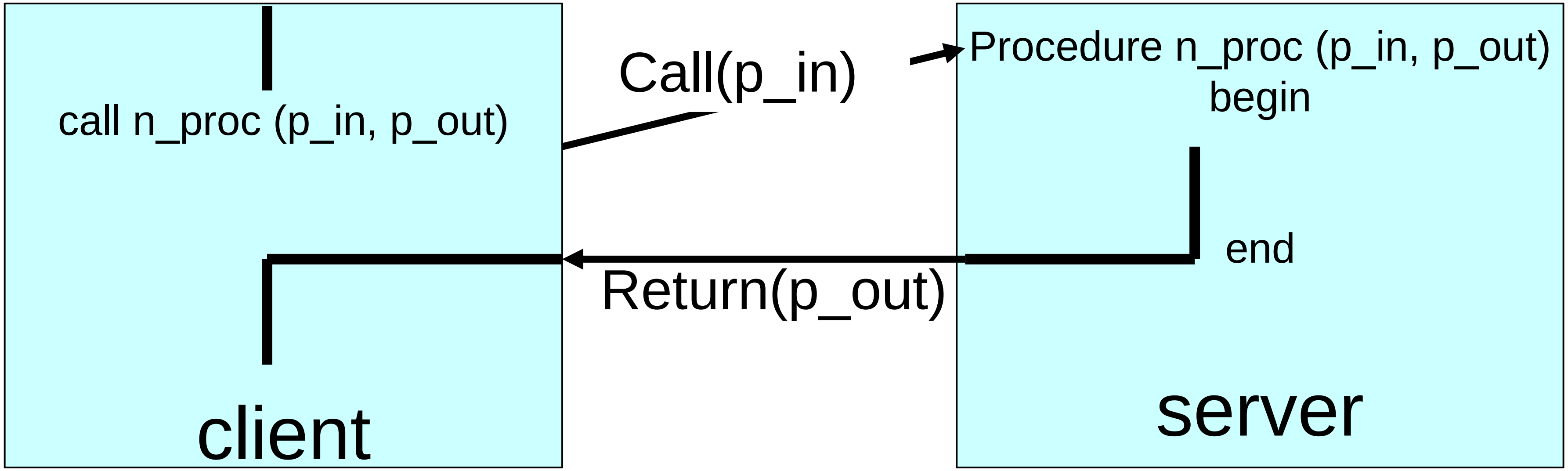
Daniel Hagimont

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

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

Client-server model based on message passing

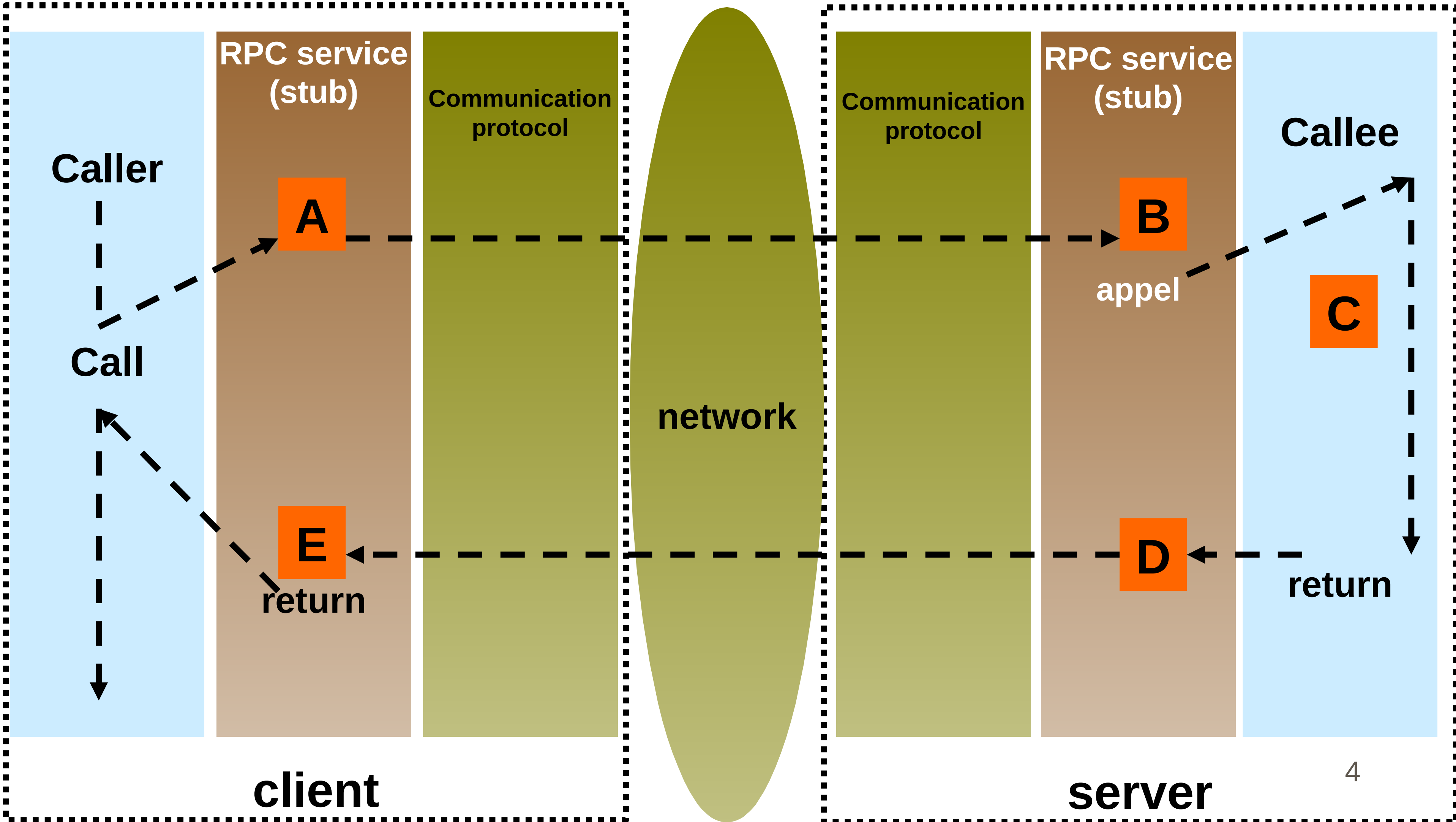
- Two exchanged messages (at least)
  - The first message corresponds to the request. It includes the parameters of the request.
  - The second message corresponds to the response. It includes the result parameters from the response.



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 as if it was centralized

RPC [Birrel & Nelson 84] Implementation principle



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

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)

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

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

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

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

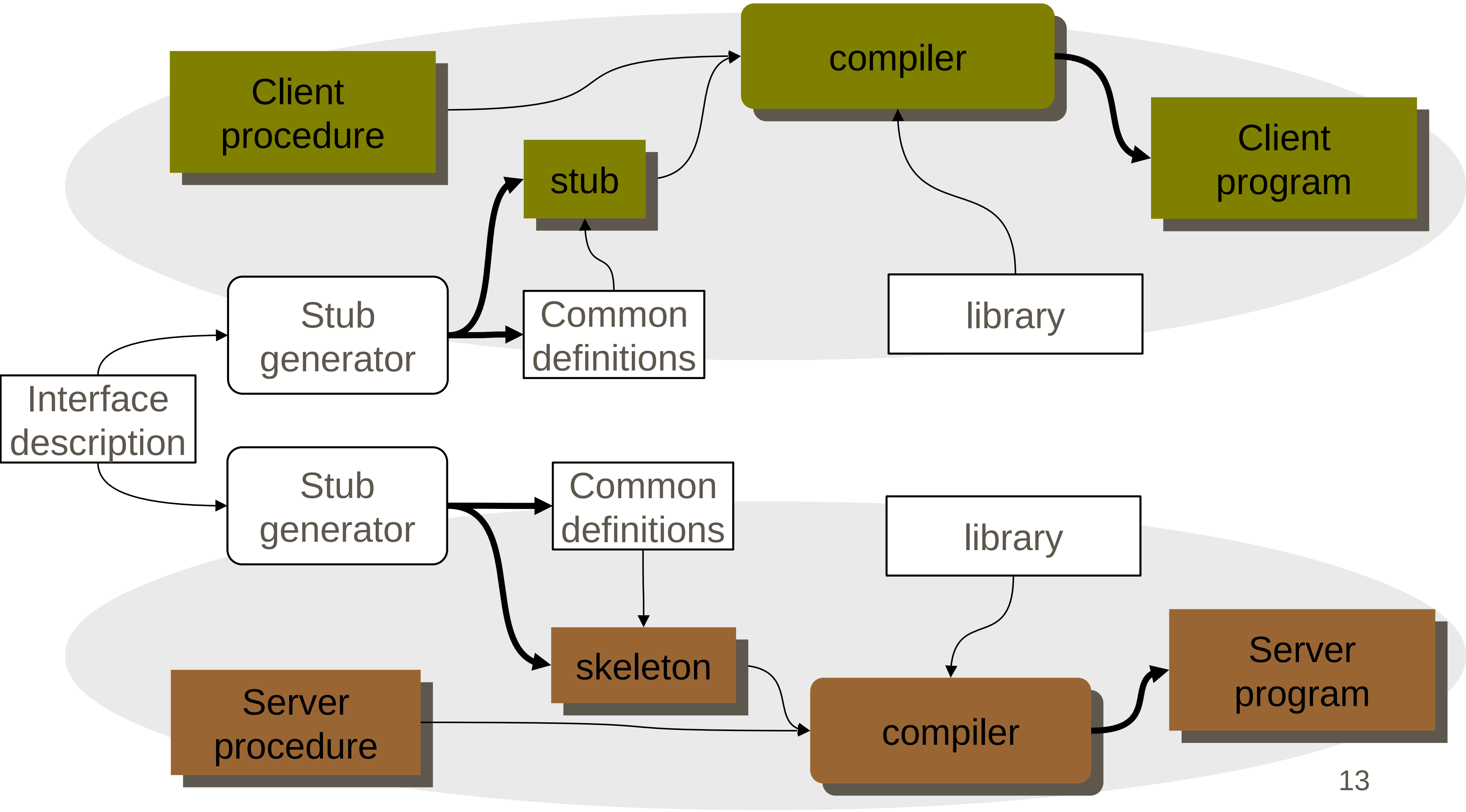
RPC Problems

- 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)

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)

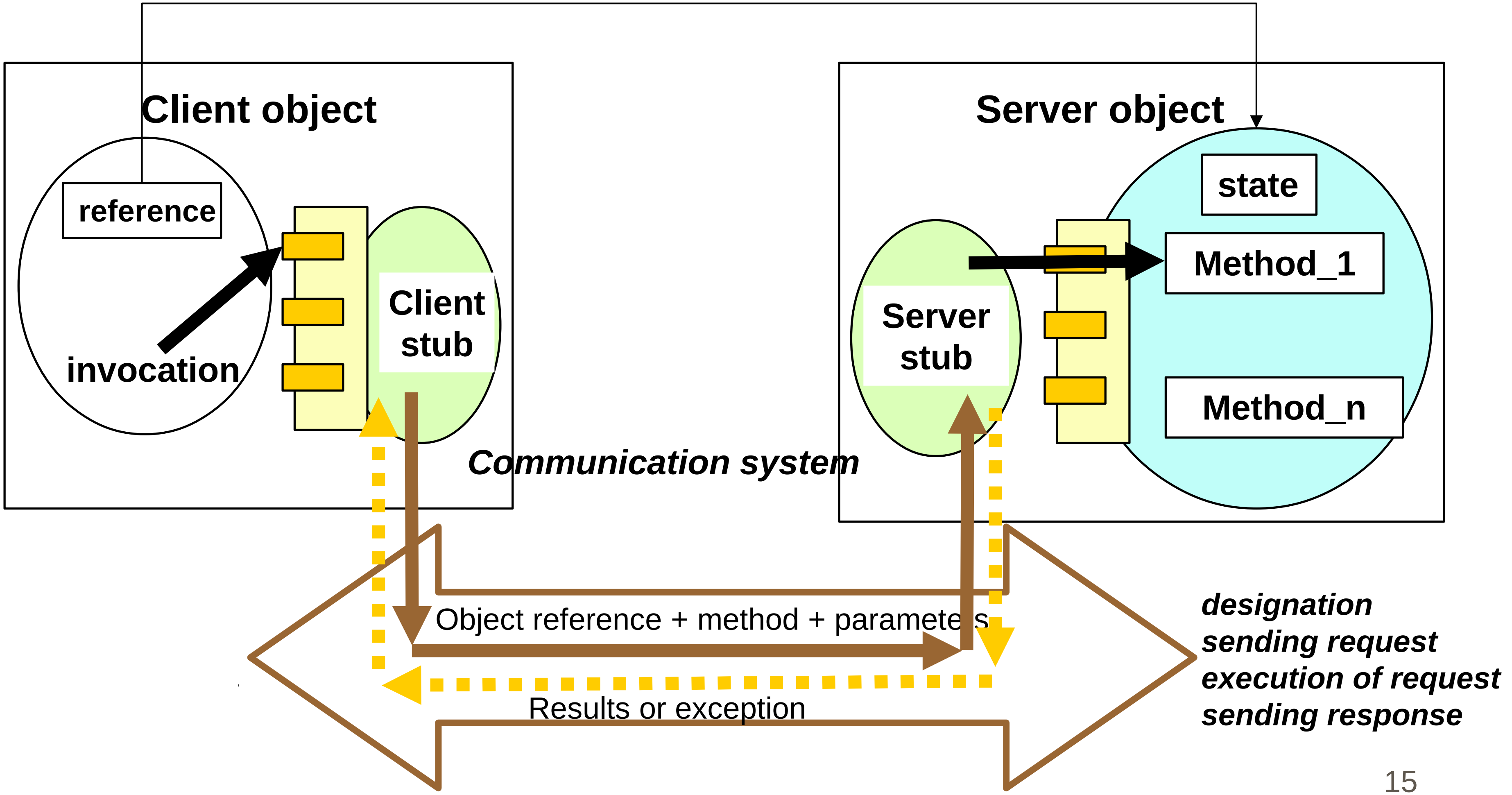
RPC
Functional mode (rpcgen)



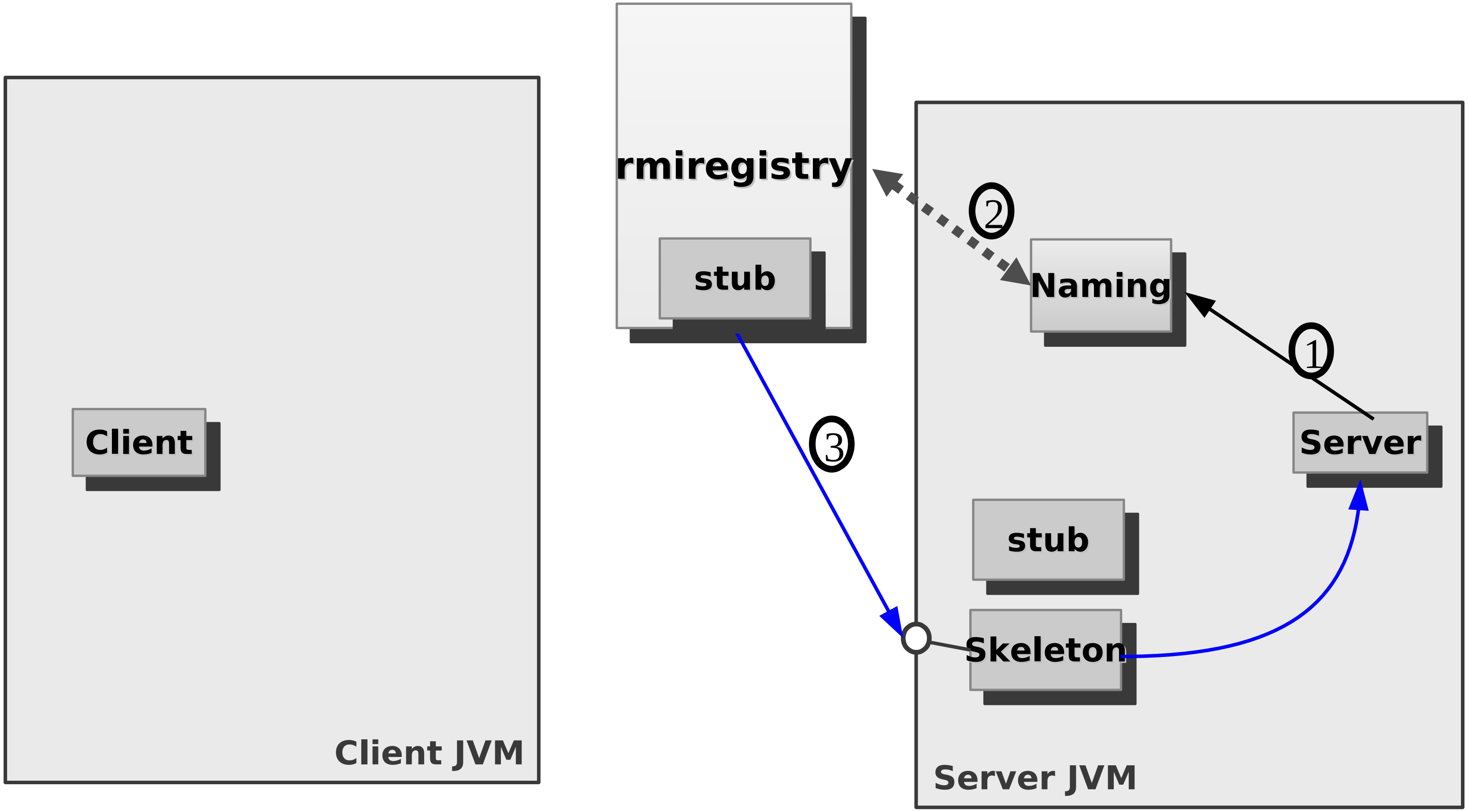
Java Remote Method Invocation RMI

- 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
Principle



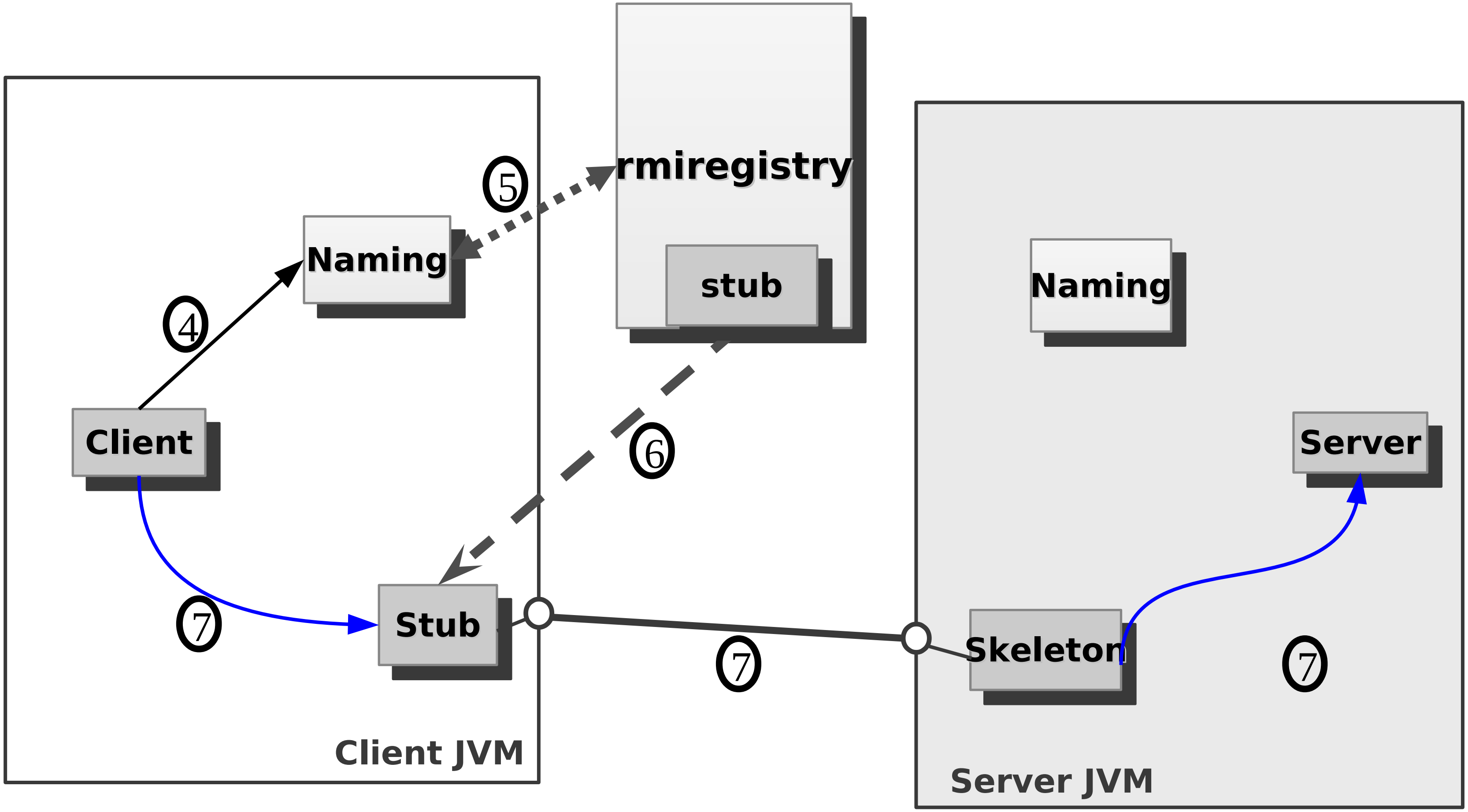
Java RMI
Server side



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 (*rebind* method)
- 2 – The naming service (*rmiregistry*) registers the *stub*
- 3 – The naming service is ready to give the *stub* to clients

Java RMI
Client side



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*

Java RMI Utilization

- Coding
  - Wrting 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

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

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

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

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();
                // 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

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) { ...}
}
```

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) { … }
  }
}
```

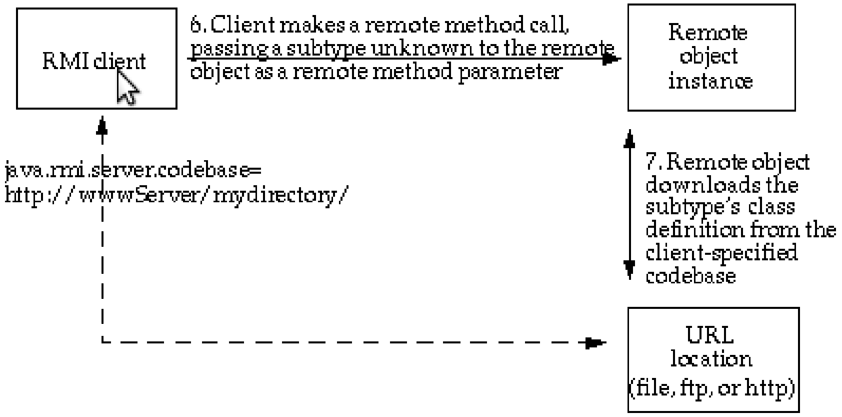
Implementation of the client class

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

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



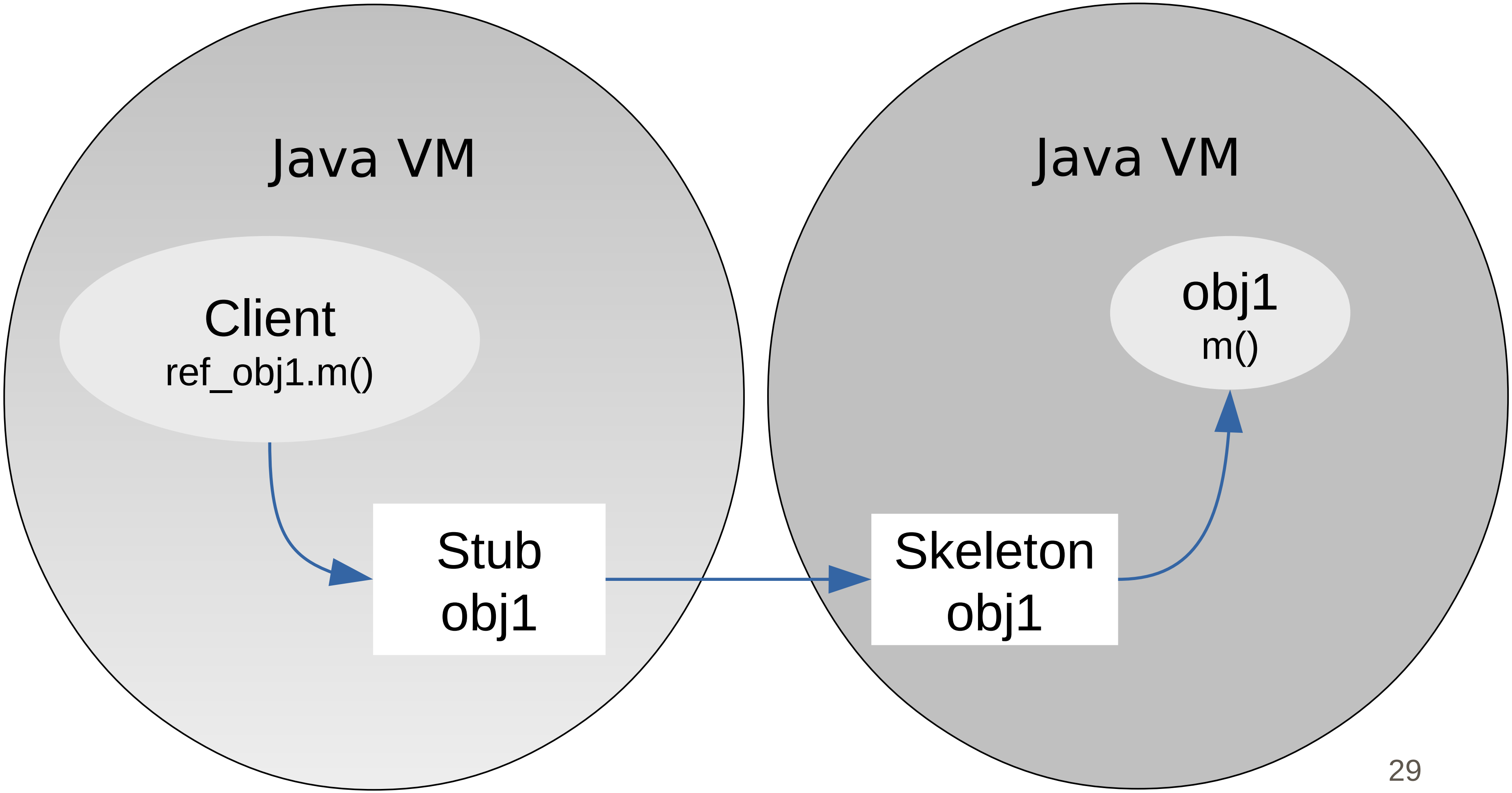
6. Client makes a remote method call, passing a subtype unknown to the remote object as a remote method parameter

java.rmi.server.codebase=
http://wwwServer/mydirectory/

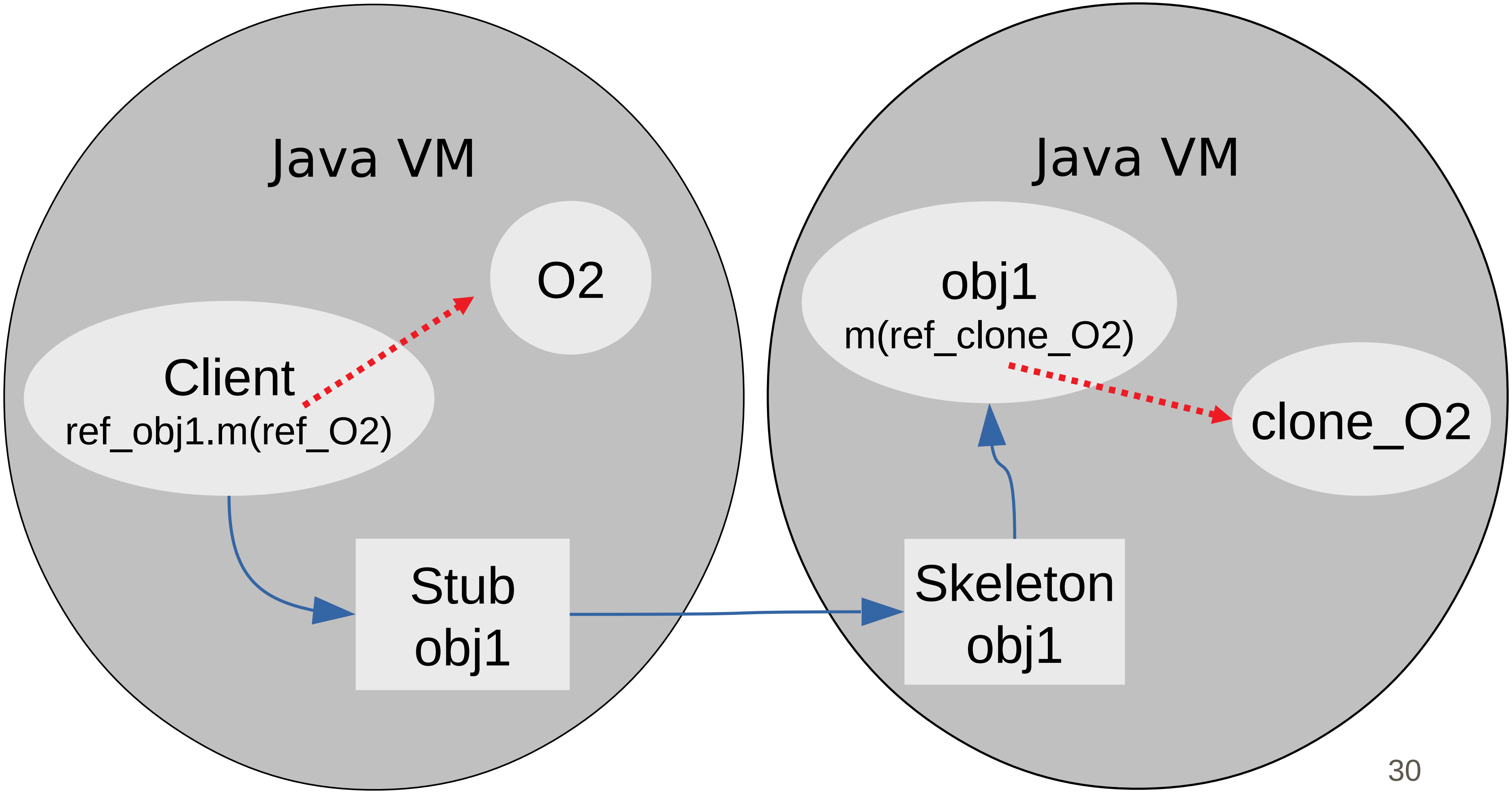
7. Remote object downloads the subtype's class definition from the client-specified codebase

Java RMI

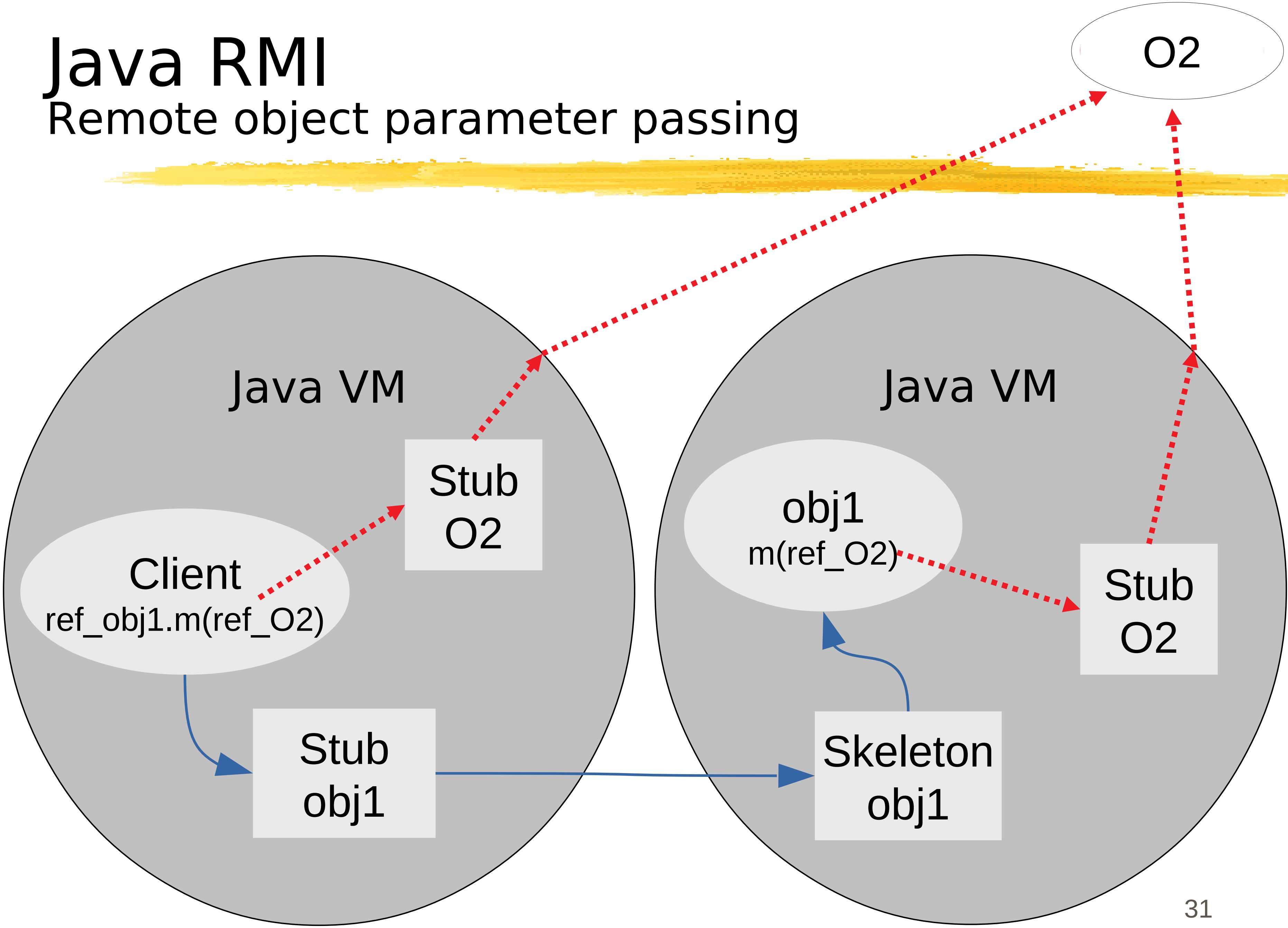
Principle of remote method invocation



Java RMI
Serializable object parameter passing



Java RMI
Remote object parameter passing



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.tutorialspoint.com/java_rmi/java_rmi_application.htm