INF0408 – DISTRIBUTED APPLICATIONS DEVELOPMENT

RMI REMOTE METHOD INVOCATION



OUTLINE

- Part I:
  - Introduction.
  - IP Addresses Handling in Java.
  - Inputs / Outputs in Java (Streams).
  - Sockets in TCP mode.
  - Threads in Java.
  - Multi-threaded applications.
  - Sockets in UDP mode.
  - Multicast sockets
- Part II:
  - Introduction to Middleware.
  - RMI (Remote Method Invocation).
  - Architecture implementation
  - Asynchronous call

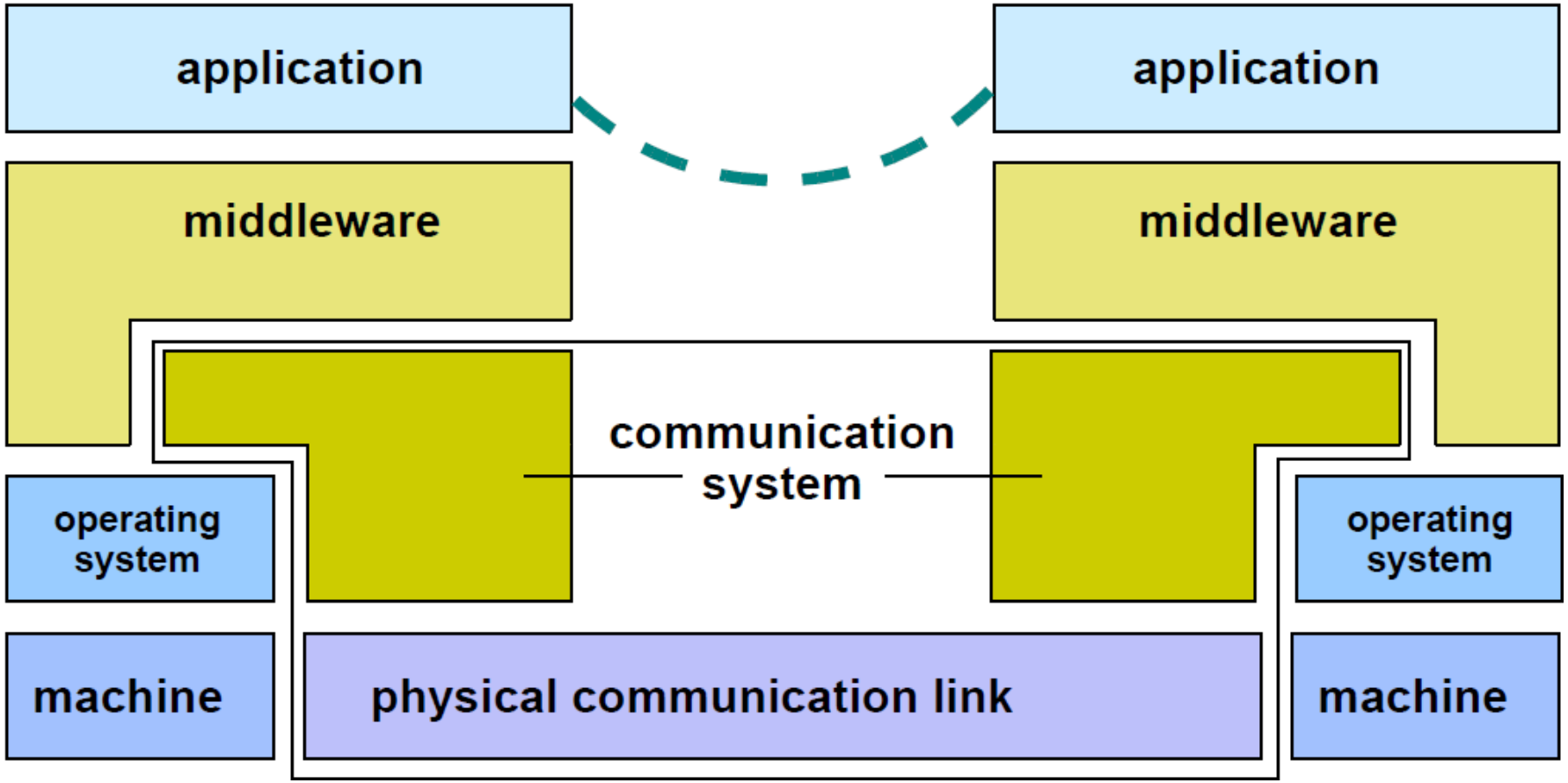
MOTIVATIONS

- Sockets are a simple and flexible technology for data communication in distributed systems
- Sockets are restricted to the transmission of data Sockets leave the semantics of this data unconsidered
- Protocols which provide the semantic interpretation of the data must be developed on the application level
- The development of such protocols is often time-consuming and error-prone

What is a middleware

Middleware is software which lies between an operating system and the applications running on it.

Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications.

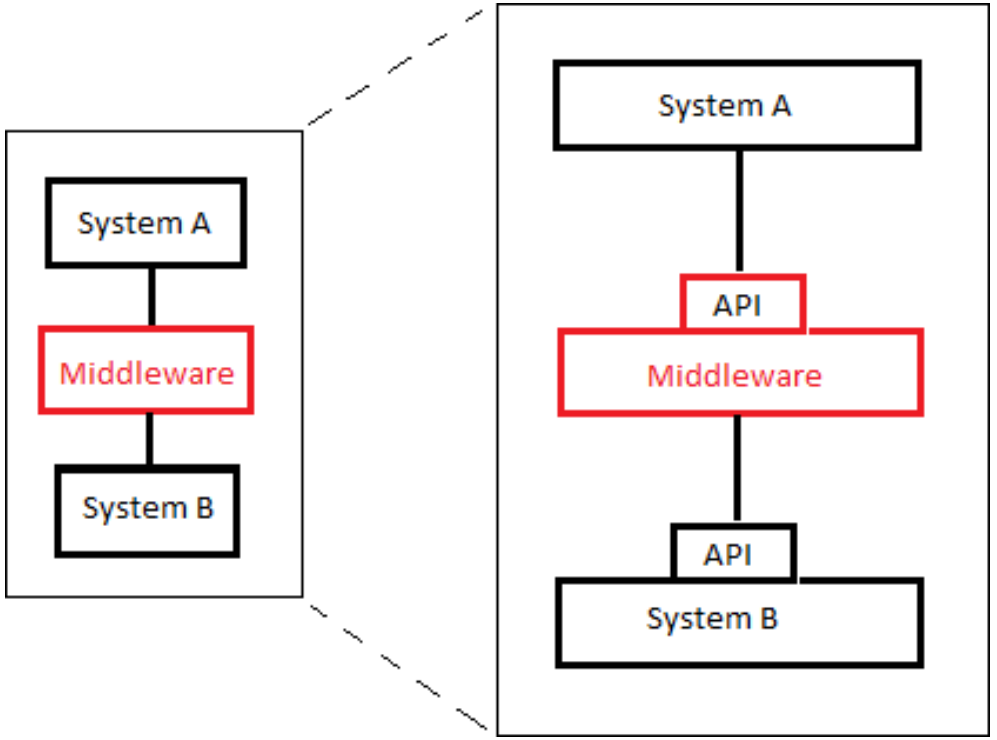


MIDDLEWARE

- Middleware is an application which connects two or more applications, How? through some interface that is called API.
- Middleware is a general term, API comes in technical details - how systems (will) communicate via or without middleware.

You can compare middleware to a translator helping people who speak different languages understand each other.

Middleware has also been referred to as "software glue", since it helps "glue" together different software so they can work together.



FUNCTIONS OF A MIDDLEWARE

- A middleware has mainly four functions :
  1. Make distribution as invisible (transparent) as possible
  2. Provide a homogeneous view of underlying heterogeneous hardware and software systems
  3. Provide services of common use for distributed systems
  4. Provide a high-level interface or API (Applications Programming Interface) for programming distributed applications

MIDDLEWARE FOR DISTRIBUTED SYSTEMS

- Middleware aims at simplifying programming distributed Systems
  - Implementation, evolution and reuse of applications code
  - Inter-platform portability of applications
  - Interoperability between heterogeneous applications

TYPES

- Remote Procedure Call (RPC)
- Object-Oriented Middleware (OOM)
  - Java RMI
  - CORBA
- Message-Oriented Middleware (MOM)
  - Java Message Service
  - IBM MQSeries
  - Web Services

WHAT TO USE?

- Sockets API ≡ send & recv calls ≡ I/O
- Remote Procedure Calls (RPC)
  - Goal: to provide a procedural interface for distributed (i.e., remote) services
  - To make distributed nature of service transparent to the programmer
- Remote Method Invocation (RMI)
  - RPC + Object Orientation
  - Allows objects living in one process to invoke methods of an object living in another process

REMOTE PROCEDURE CALL (RPC)

- RPC is a technology developed in the 80s to call procedures on remotes computers (with the procedural programming paradigm)
- RPC allows the call of procedures located in another process space on a remote computer (or on the same machine)
- Technical issues for implementing RPC
  - Different address spaces
  - Heterogeneous machines

RPC IMPLEMENTATION

- Different address spaces
  - In the local case:
    - Data used in a procedure call is simply passed as a reference (a pointer)
    - This reference refers to a physical memory address
    - Such a reference has no correct meaning in a different address space
  - In the distributed case:
    - The referenced data used in a remote call needs to be passed as a copy

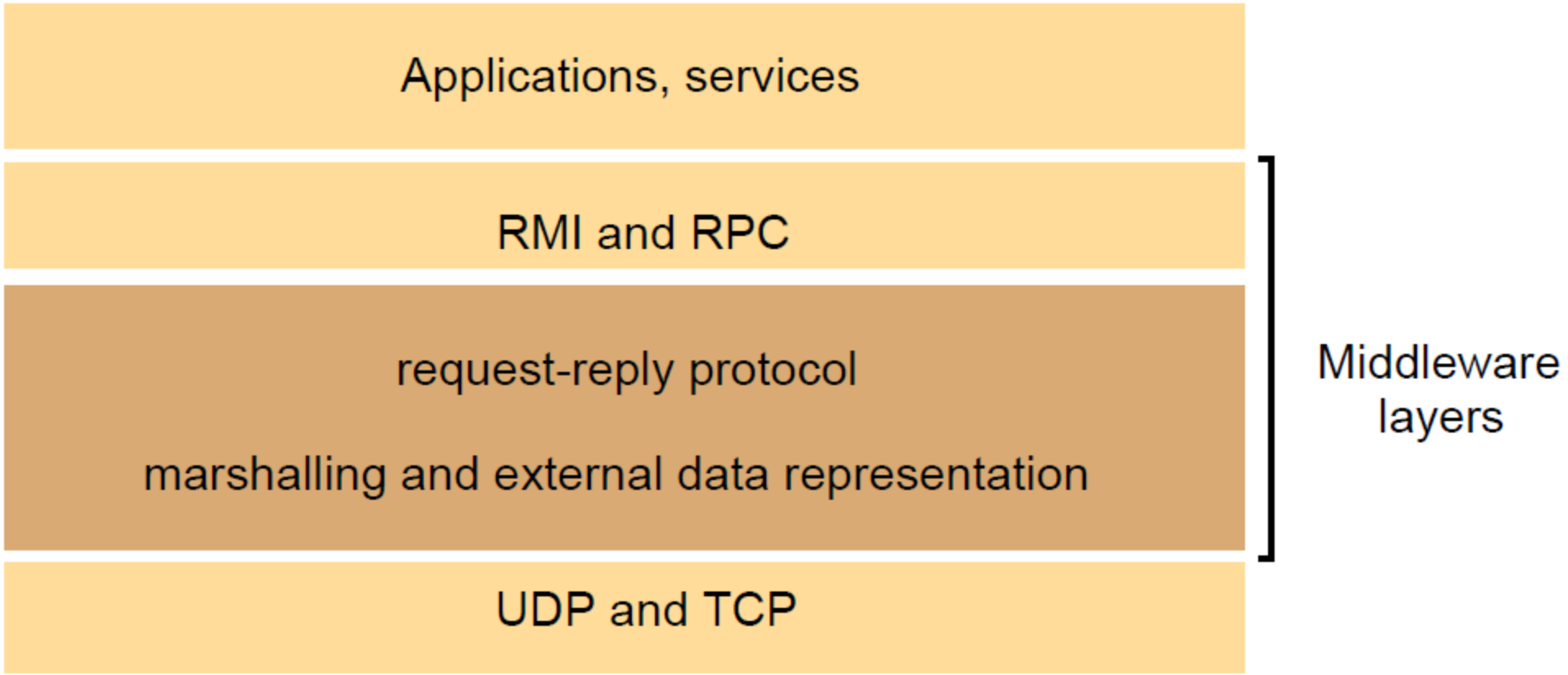
RPC IMPLEMENTATION (2)

- Heterogeneous machines
  - In communication between heterogeneous computer architectures, the internal representation of data on another computer may not be the same as on the original computer
  - Data sent in remote procedure calls must be converted into a platform-independent data format (e.g. XDR – eXtensible Data Representation)
  - Data received in remote procedure calls must be converted back into an internal representation of the receiver's side

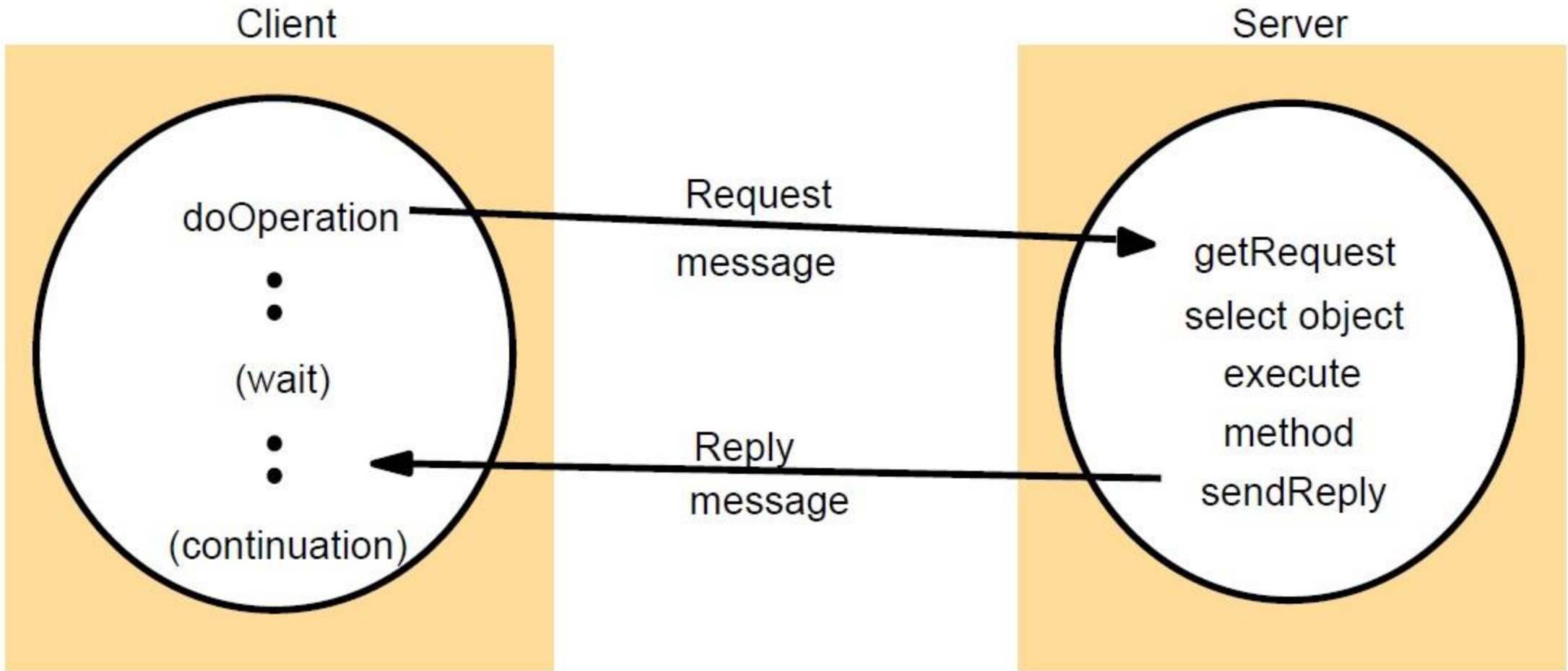
RPC IMPLEMENTATION

- As an example,
  - a process on machine *A* can call a procedure on machine *B*.
  - The process on *A* is suspended and execution continues on *B* .
  - When *B* returns, the return value is passed to *A* and *A* continues execution.

Middleware layers



Request-reply communication

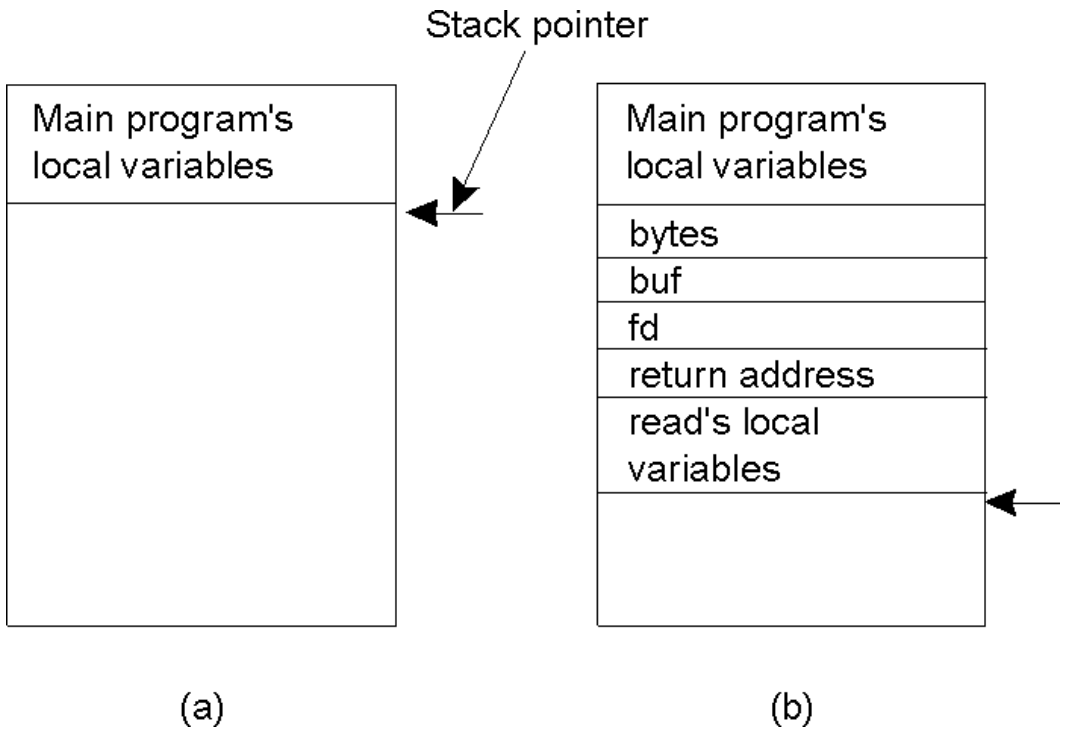


CONVENTIONAL PROCEDURE CALL

a) Parameter passing in a local procedure call: the stack before the call to read

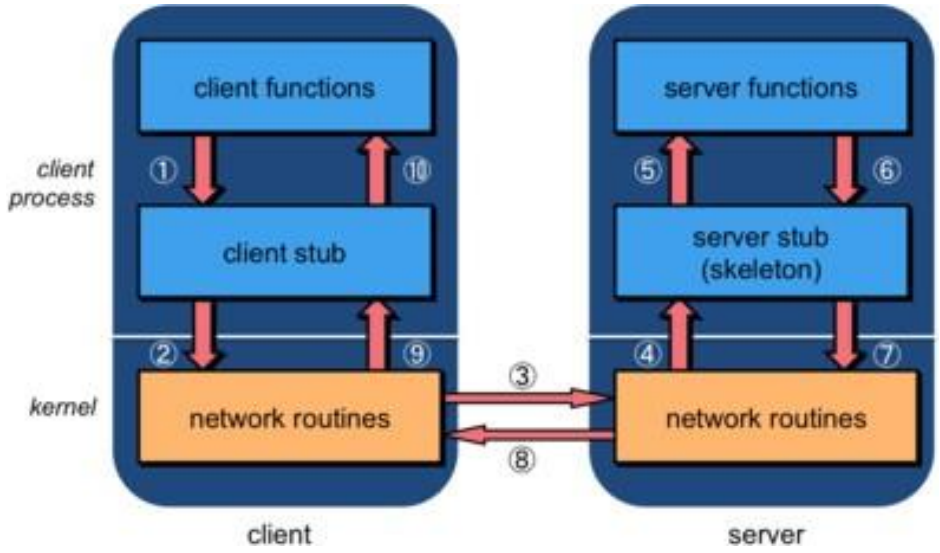
```
Count = read (fd, buf, nbytes);
```

b) The stack while the called procedure is active



- The sequence of operations that takes place in making a remote procedure call

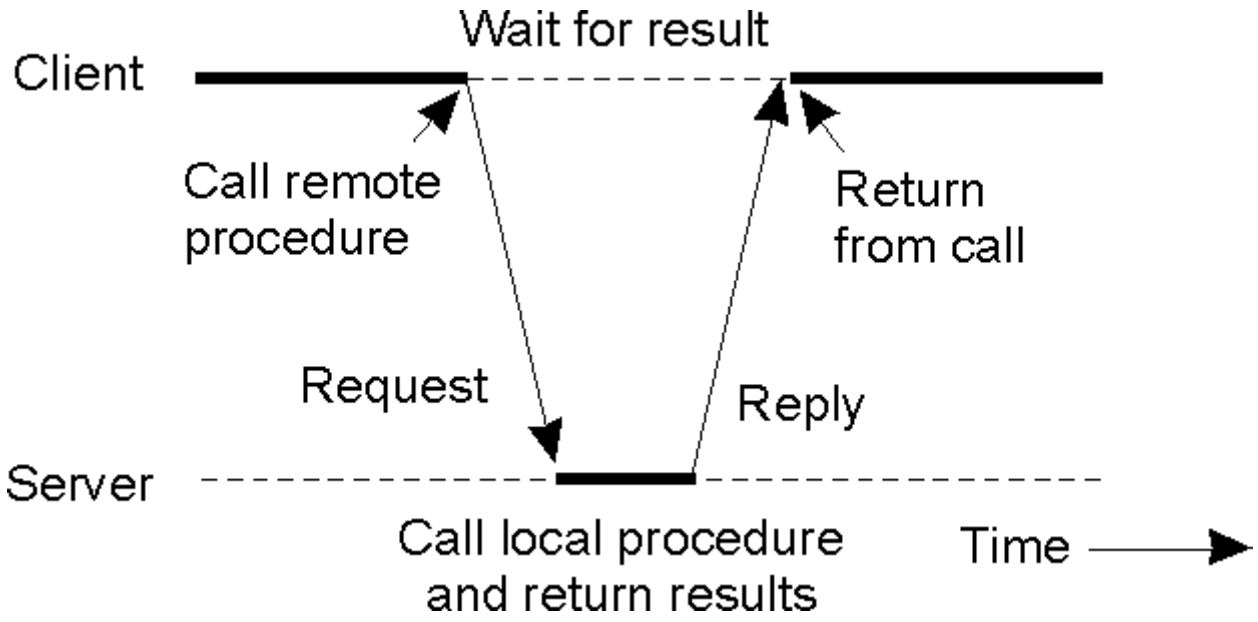
1. The client calls a local procedure, called the client stub. To the client process, it appears that this is the actual remote procedure, which it can call using a local procedure call (since the stub is a local procedure). The client stub packages the arguments to the remote procedure (this may involve converting them to a standard format) and builds one or more network messages. The packaging of arguments into a network message is called marshaling.
2. Network messages are sent by the client stub to the remote system (by writing to a socket via system calls to the local kernel).
3. Network messages are transferred by the kernel to the remote system via some protocol (either connectionless or connection-oriented).
4. A server stub process, sometimes called a skeleton, on the server receives the messages. It unmarshals the arguments from the messages and, if needed, converts them from a standard format into a machine-specific format.



5. The server stub calls a local procedure call to the actual server function, passing it the arguments that it received from the client. The server function gets the illustion that it was called locally by the client since the server stub calls it locally with the same parameters that the client transmitted.
6. When the server is finished, it returns to the server stub with its return values.
7. The server stub converts the return values to a standard format (if necessary) and marshals them into one or more network messages to send to the client stub.
8. Messages get sent back across the network to the client stub.
9. The client stub reads the messages from the local kernel.
10. It then returns the results to the client function (possibly converting them first). The client feels that it just received a return value from the remote function, unaware that all the network messaging took place.

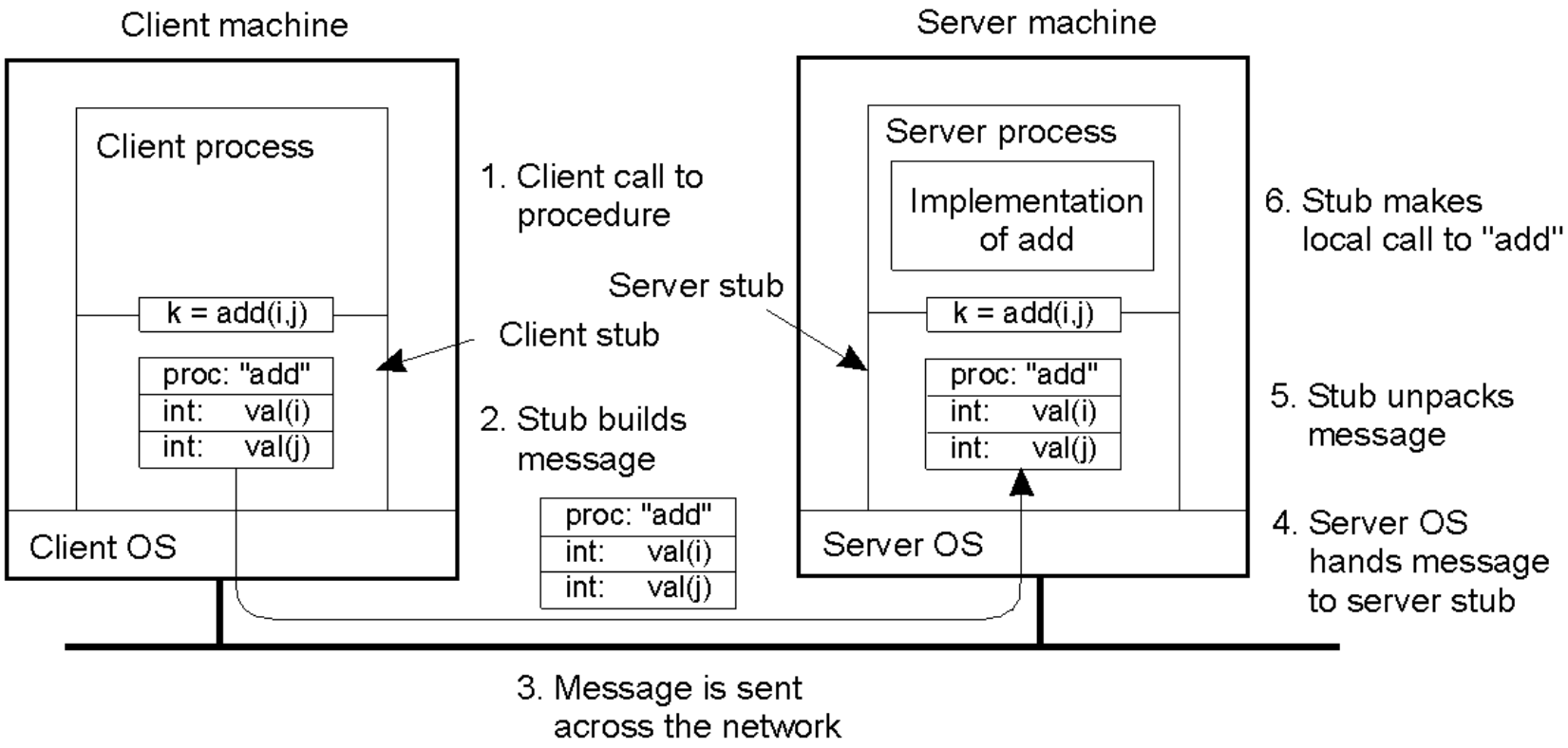
CLIENT AND SERVER STUBS

- Principle of RPC between a client and server program.
- The read stub is called on behalf of the real read procedure!



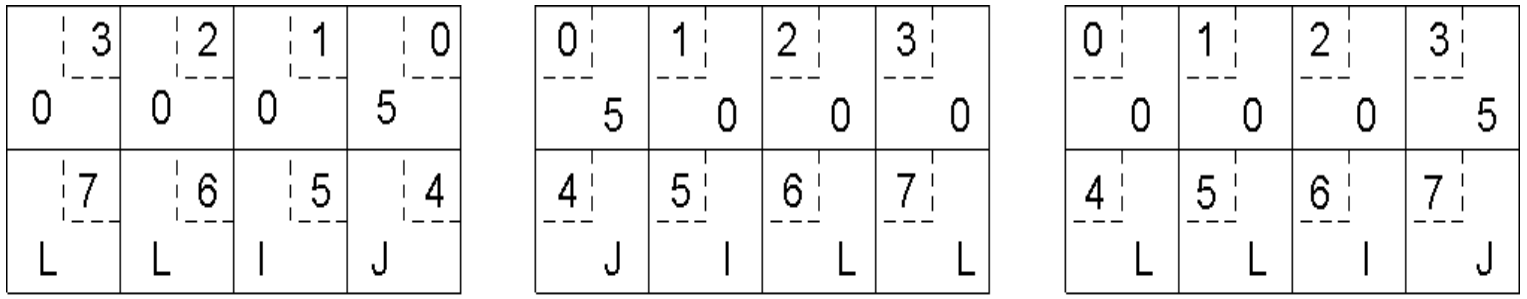
PASSING VALUE PARAMETERS (1)

- Steps involved in doing remote computation through RPC



PASSING VALUE PARAMETERS (2)

- Different character set standards (ASCII vs EBCDIC)
- Little-Endian vs Big-Endian Architecture.



(a) (b) (c)

a) Original message on the Pentium (L. E.)

b) The message after receipt on the SPARC (B. E.)

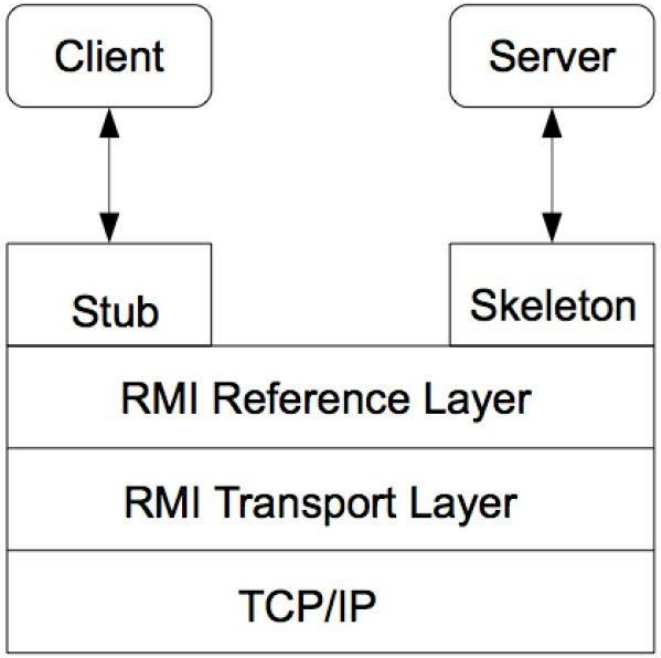
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

THE JAVA REMOTE METHOD INVOCATION (RMI)

- The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM.
- Java RMI provides applications with transparent and lightweight access to remote objects. RMI defines a high-level protocol and API.
- Programming distributed applications in Java RMI is simple.
  - It is a single-language system.
  - The programmer of a remote object must consider its behaviour in a concurrent environment.

RMI ARCHITECTURE

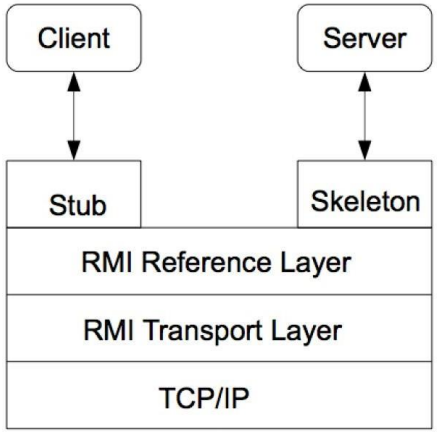
- RMI Architecture
  - Stub: lives on the client; pretends to be the remote object
  - Skeleton: lives on the server; talks to the true remote object
  - Reference Layer: determines if referenced object is local or remote
  - Transport Layer: packages remote invocations; dispatches messages between stub and skeleton



RMI ARCHITECTURE

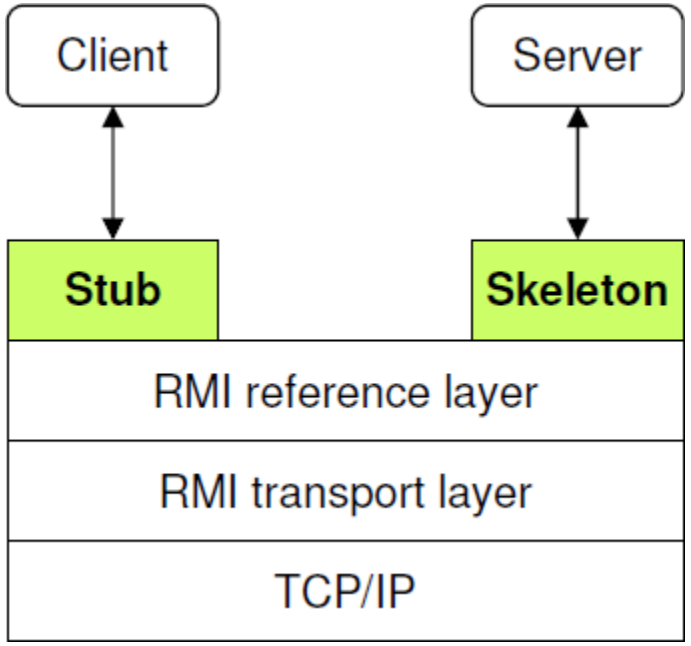
- Transparency regarding distribution
  - RMI offers full transparency regarding distribution
  - After a first initialization, a call can be used in exactly the same way as in the local case

- RMI client and server
  - Both client and server are normal objects implemented in Java
  - The server must document the interface it provides for remote access
  - From this description, additional classes are automatically created by a special compiler
  - These classes internally take care of communication handling between client and server
  - These classes are known as *Stub* (client-side) and *Skeleton* (server-side)



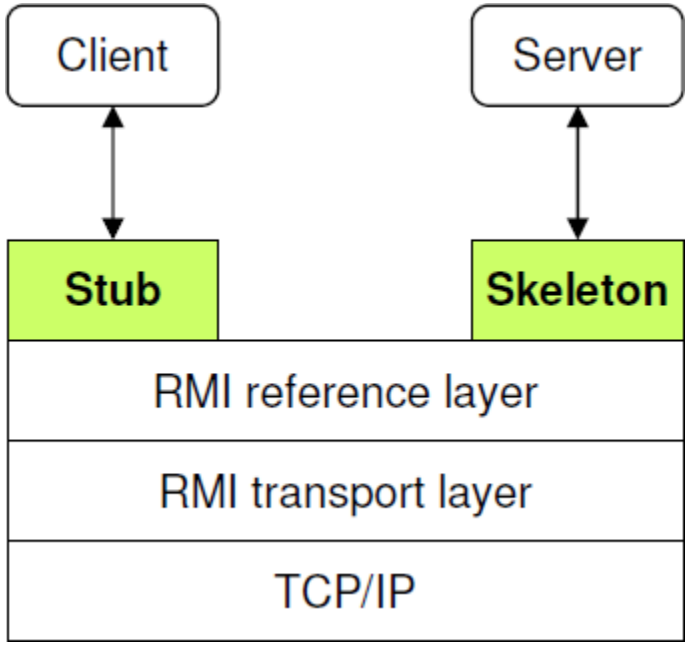
STUB/SKELETON

- Stub
  - lives on client
  - pretends to be remote object - a proxy for the remote object
- Skeleton
  - lives on server
  - receives requests from stub, talks to the remote object and delivers response to stub
- Stubs and skeletons are not written by the programmer!
- They are generated by a special compiler “rmic”



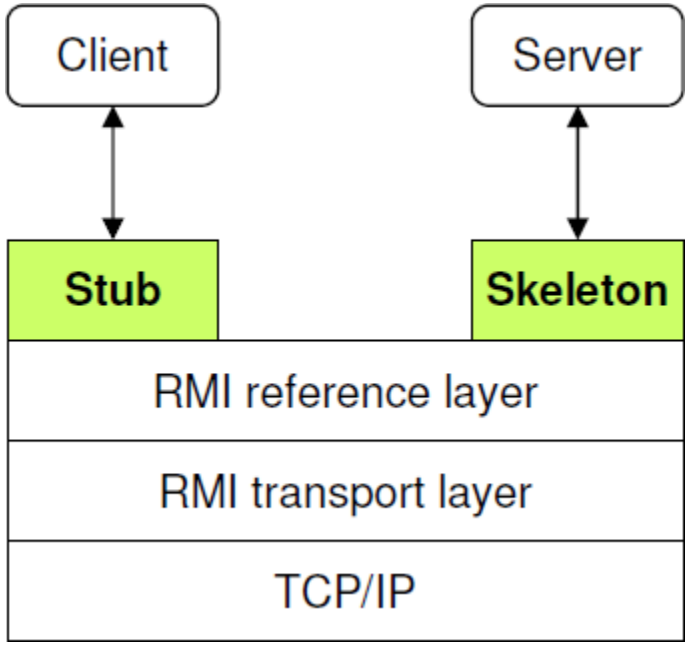
STUB/SKELETON

- Stub – responsibilities
  - Initiate remote calls
  - Marshals arguments to be sent
  - Informs the remote reference layer that a call should be invoked on the server
  - Unmarshals a return value (or exception)
  - Informs the remote reference layer that the call is complete



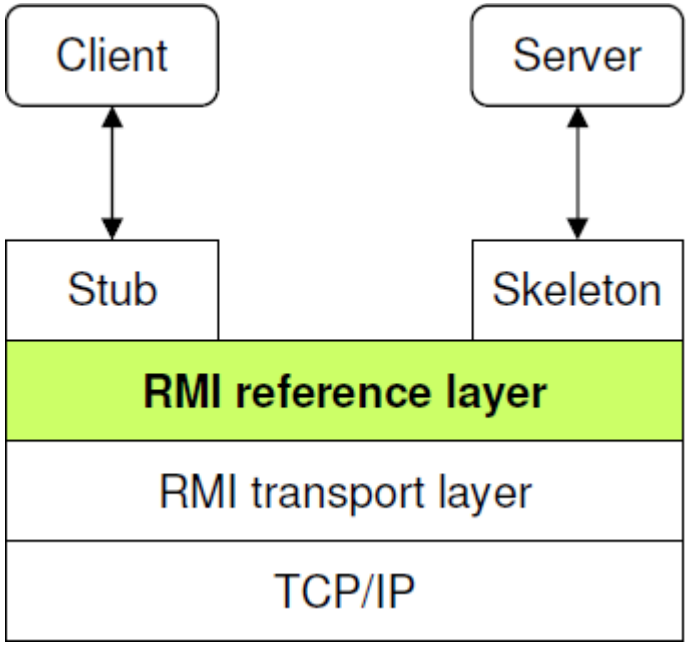
STUB/SKELETON

- Skeleton – responsibilities
  - Unmarshals incoming arguments
  - Calls the actual remote object implementation
  - Marshals return values for transport to the client
- Marshaling – definition
  - The process of converting native programming language data types to a format suitable for transmission across a network



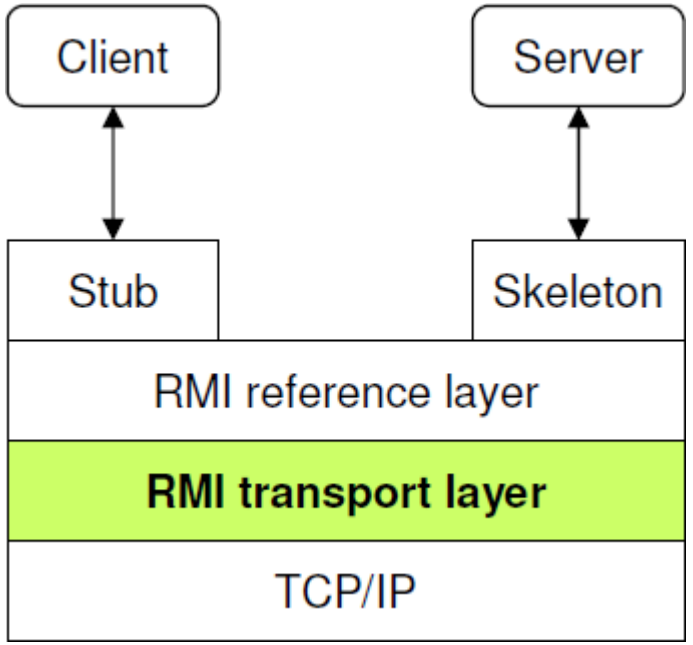
RMI REFERENCE LAYER

- The reference layer in RMI
  - It finds the respective communication partners
  - It includes the name service, the *registry*



RMI TRANSPORT LAYER

- The transport layer in RMI
  - It manages communication connections
  - It handles communication
  - It must not be confused with the network transport layer (e.g. TCP/IP)

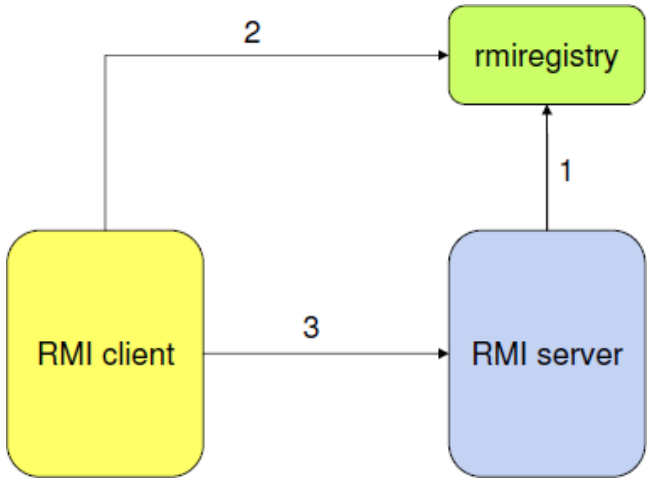


OVERVIEW OF RMI-BASED DISTRIBUTED APPLICATIONS

- RMI applications comprise two separate programs, a server and a client
- The server program:
  - creates some remote objects,
  - makes references to these objects accessible,
  - and waits for clients to invoke methods on these objects.
- The client program:
  - obtains a remote reference to one or more remote objects on a server,
  - and then invokes methods on them.
- RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

OVERVIEW OF RMI-BASED DISTRIBUTED APPLICATIONS

- Distributed object applications follow these steps:
  - Client locates remote objects on server
    - Various mechanisms to obtain references to remote objects
    - An application server can register its remote objects with RMI registry (RMI's simple naming facility)
    - An application server can return remote object references as part of other remote invocations
  - Client communicates with remote objects on server
    - To the client programmer, remote communication looks similar to regular Java method invocations
    - Details of remote communication are handled by RMI and transparent to client and server programs
  - Load class definitions for objects that are passed around
    - Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definition



A SIMPLE EXAMPLE

REMOTE OBJECTS

- Objects with methods that can be invoked across Java virtual machines are called *remote objects*
- An object becomes remote by implementing a *remote interface*, which has the following characteristics:
  - A remote interface extends the interface *java.rmi.Remote*
  - Each method of the interface declares *java.rmi.RemoteException* in its throws clause, in addition to any application-specific exceptions

A simple example

Remote object – Interface definition

```
import java.rmi.*;


public interface Hello
    extends Remote {

    // A method provided by the
    // remore object
    public String sayHello()
          throws RemoteException;

}
```

A simple example (2)

Remote object – Interface definition

```
import java.rmi.*;


public interface Hello
    extends Remote {

    // A method provided by the
    // remore object
    public String sayHello()
          throws RemoteException;

}
```

Remote object – Class implementation

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

public  class HelloImp
    implements Hello {

    private String message;

    public Hello(String s) {
          message = s ;
    }

    public String sayHello ()
          throws RemoteException {
          return message ;
    }

}
```

A simple example (3)

Server

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

public class HelloServer {

  public static void  main(String [] args){
     ...

     try {
        // Create a Hello remote object
        HelloImp h =
             new HelloImp ("Hello world !");
        Hello h_stub = (Hello)
        UnicastRemoteObject.exportObject(h, 0);

        // Register the remote object in RMI
        // registry with a given identifier
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("Hello1", h_stub);

        System.out.println ("Server ready");

     } catch (Exception e)  {
             System.err.println("Error on server :"
             + e) ;
             e.printStackTrace();
             return;
     }
  }
}
```

A simple example (4)

Server

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

public class HelloServer {

  public static void  main(String [] args){
     ...

     try {
        // Create a Hello remote object
        HelloImp h =
              new HelloImp ("Hello world !");
        Hello h_stub = (Hello)
        UnicastRemoteObject.exportObject(h, 0);

        // Register the remote object in RMI
        // registry with a given identifier
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("Hello1", h_stub);

        System.out.println ("Server ready");

     } catch (Exception e)  {
              System.err.println("Error on server :"
              + e) ;
              e.printStackTrace();
              return;
     }
  }
}
```

Client

```
import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient {
  public static void main(String [] args) {

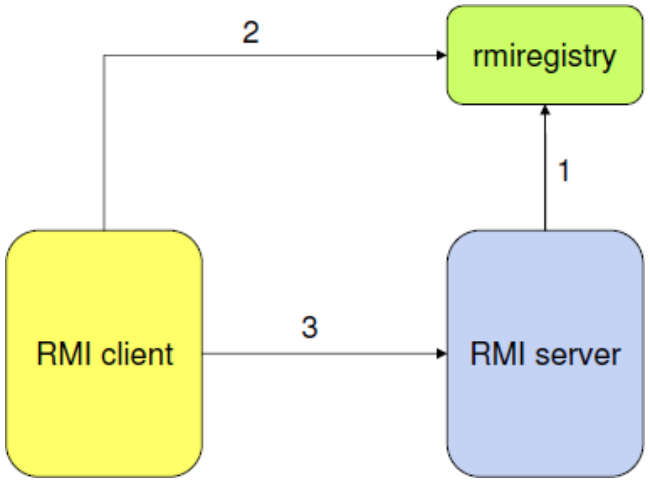
     try {
     if (args.lenght < 1) {
        System.out.println("Usage:
              java HelloClient <server host>");
        return;
     }
     String host = arg[0];

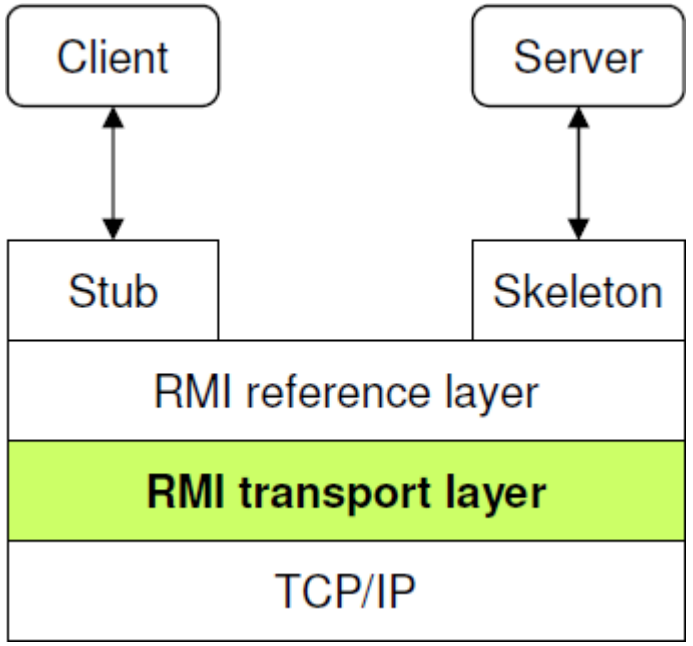
     // Get remote object reference
     Registry registry =
        LocateRegistry.getRegistry(host);
     Hello h = (Hello) registry.lookup("Hello1");

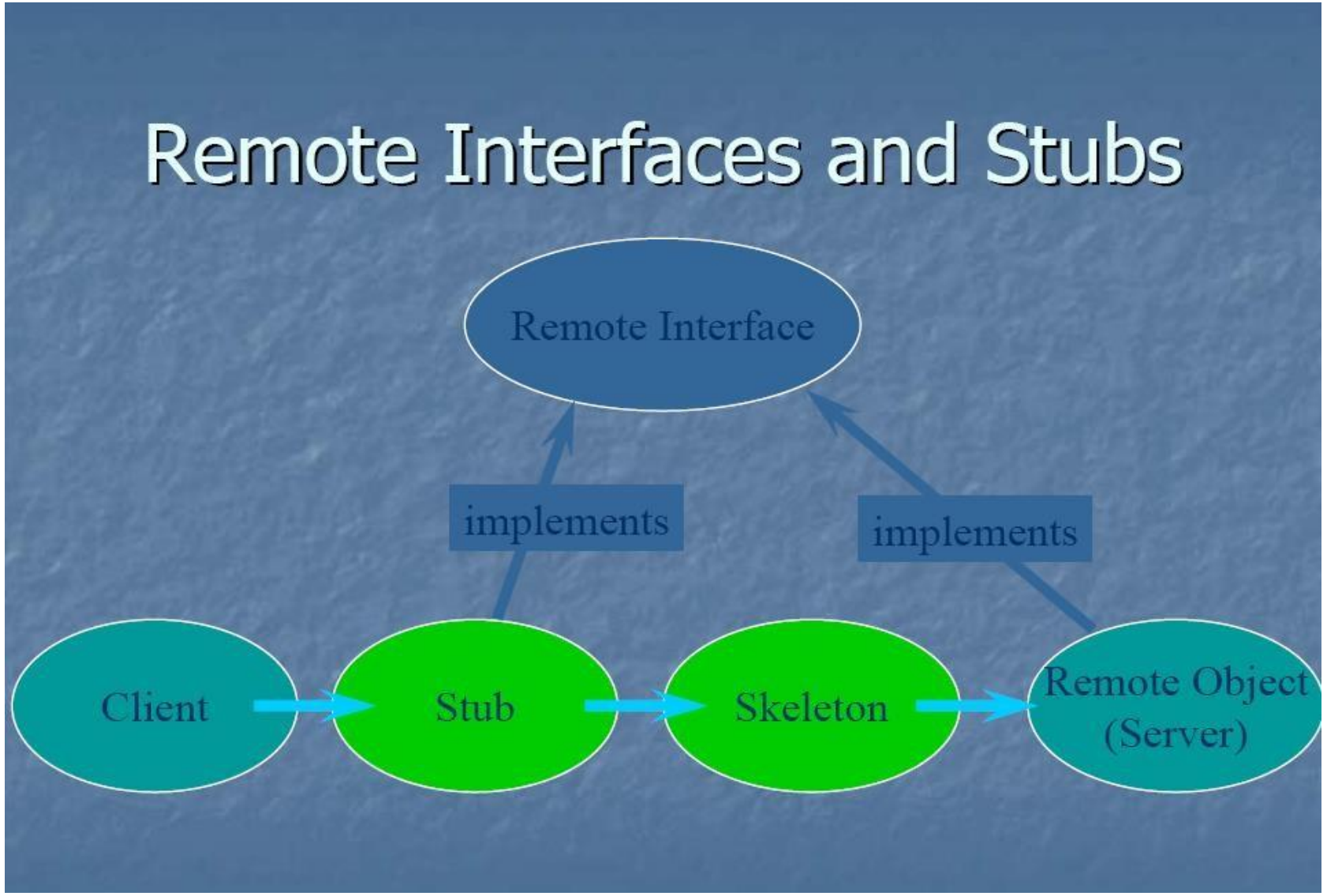
     // Remote method invocation
     String res = h.sayHello();

     System.out.println(res);

     } catch (Exception e)  {
              System.err.println("Error on
              client: " + e) ;
     }
  }
}
```

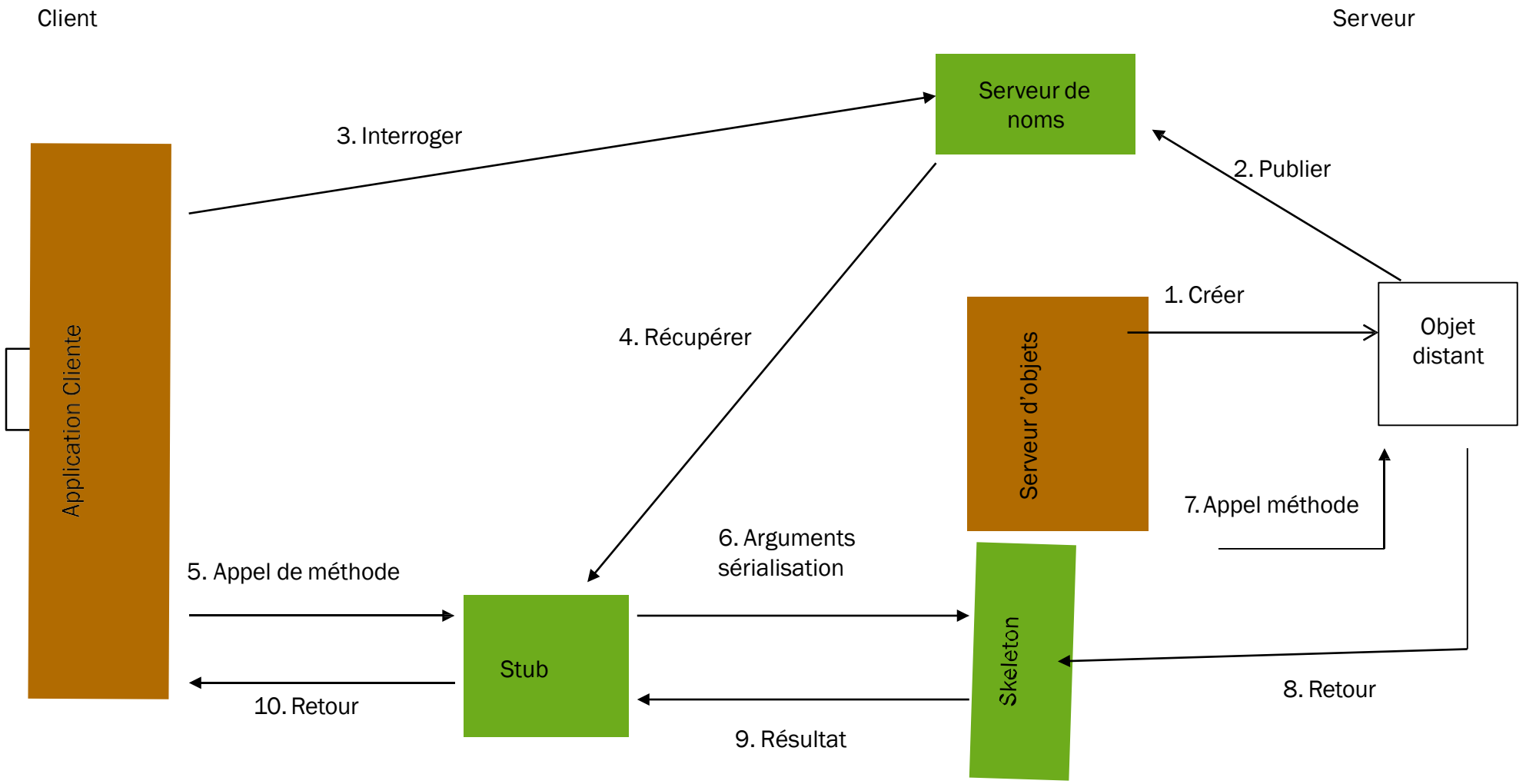

2
rmiregistry
1
RMI client
3
RMI server


Client
Server
Stub
Skeleton
RMI reference layer
RMI transport layer
TCP/IP

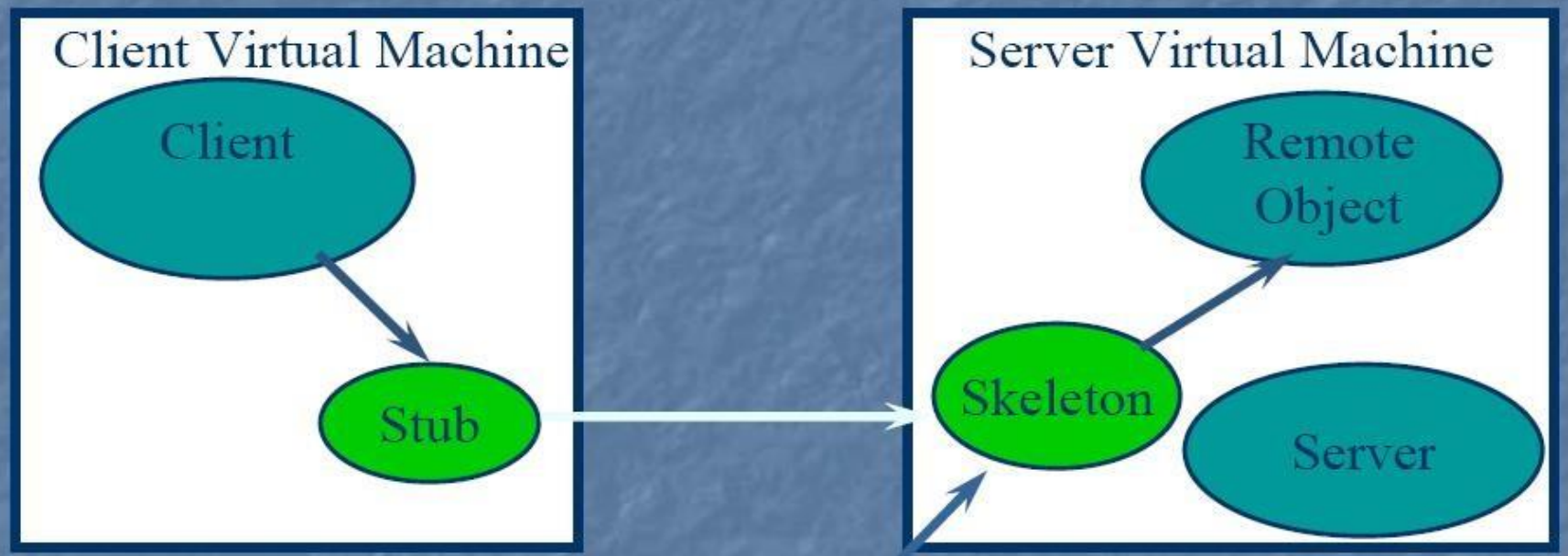


- By extending the interface java.rmi.Remote, the Interface (e.g. Hello, Compute, …) identifies itself as an interface whose methods can be invoked from another Java virtual machine.
- Any object that implements this interface can be a remote object.
- A remote method must be defined as being capable of throwing a java.rmi.RemoteException.
- This exception is thrown by the RMI system from a remote method invocation to indicate that either a communication failure or a protocol error has occurred.

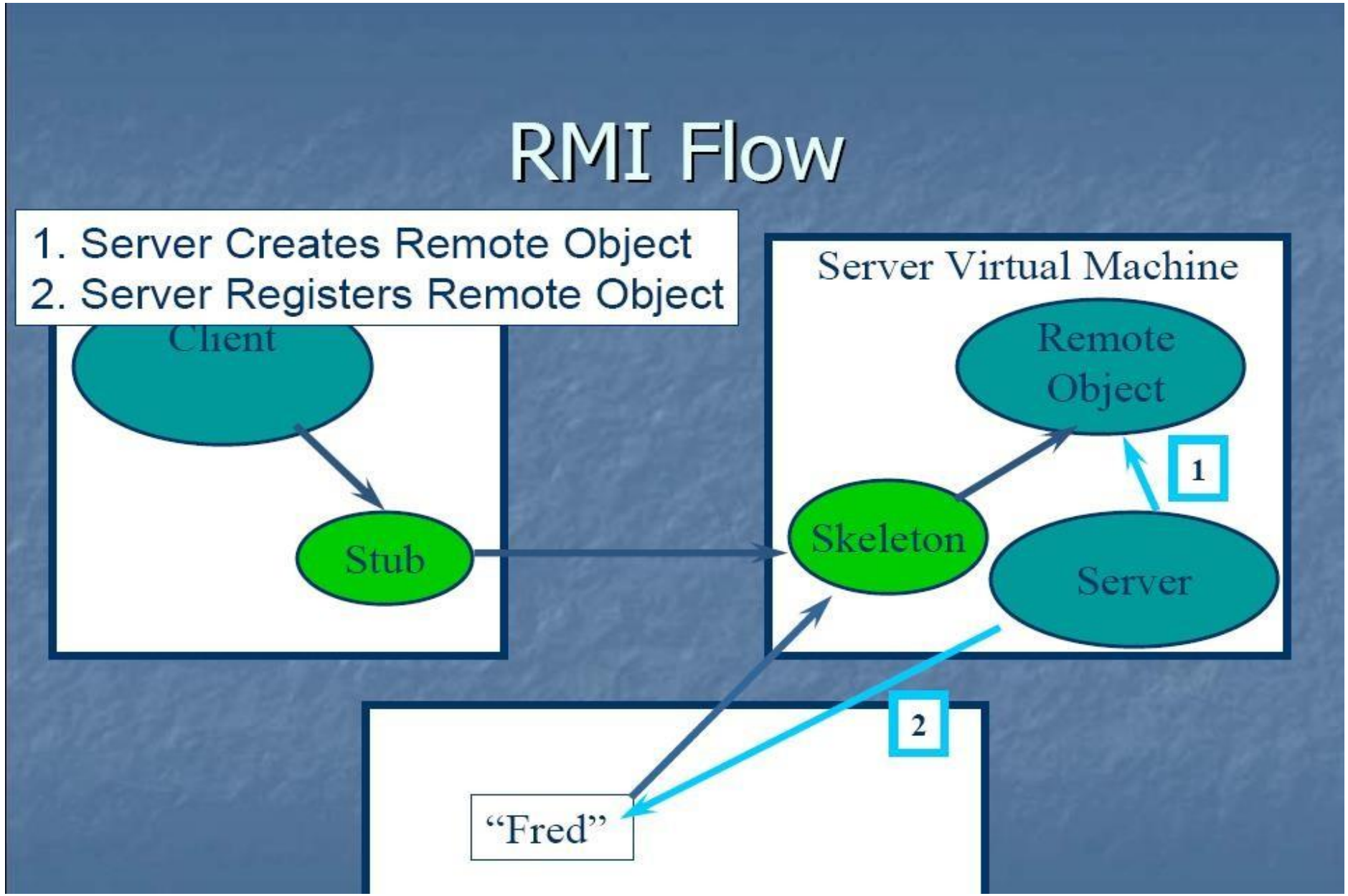
Etapes d'un appel de méthode distante

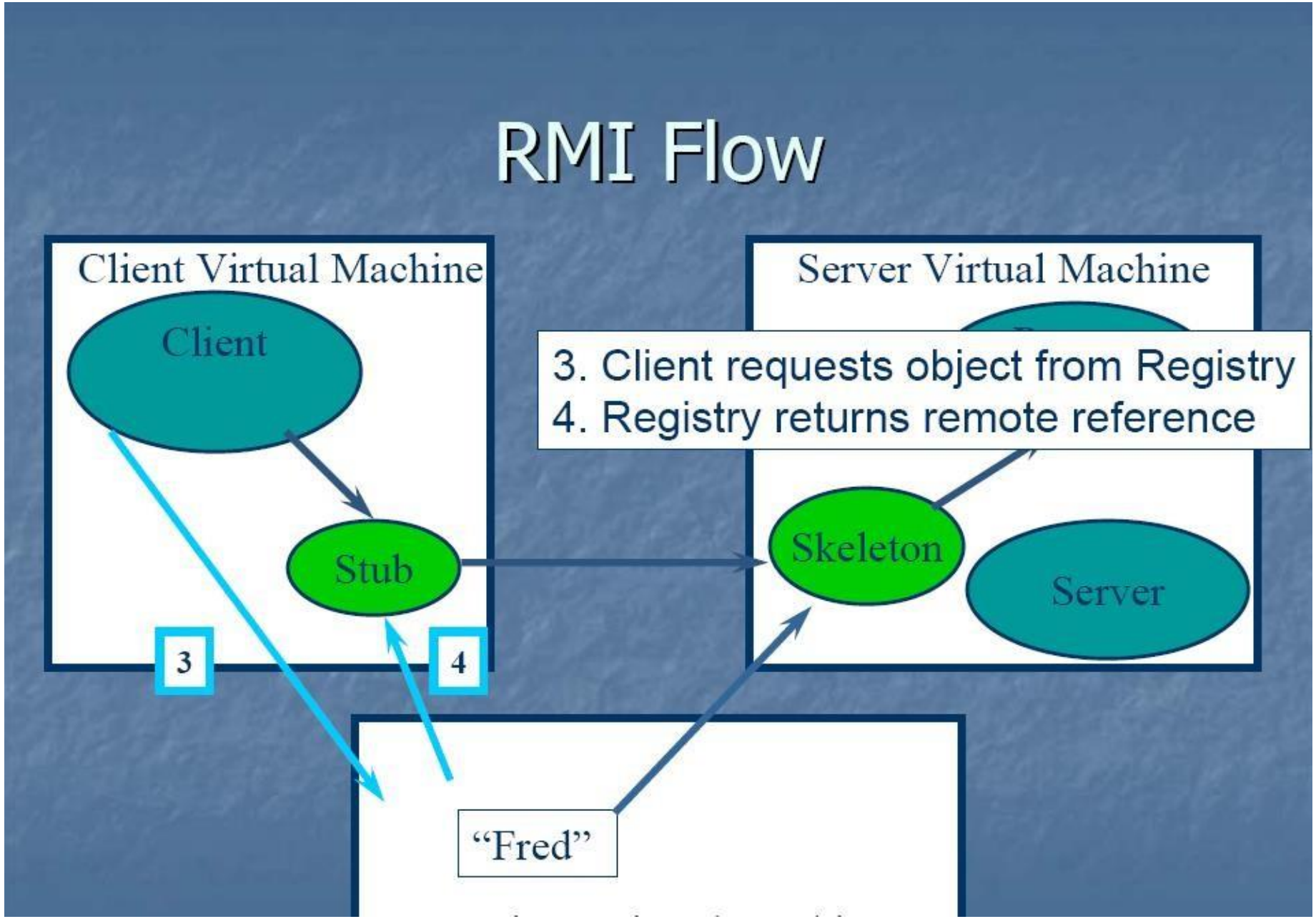


RMI System Architecture

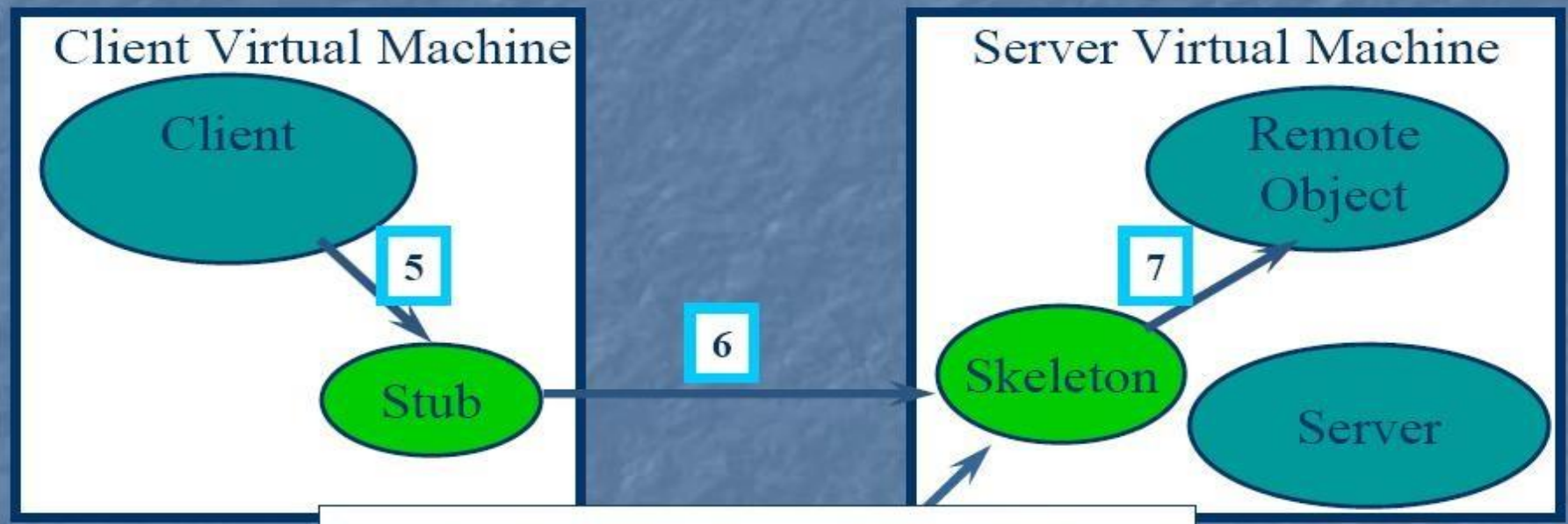


RMI Flow


1. Server Creates Remote Object
2. Server Registers Remote Object
Client
Stub
Server Virtual Machine
Remote Object
1
Skeleton
Server
2
"Fred"


RMI Flow
Client Virtual Machine
Client
Stub
3
4
Server Virtual Machine
3. Client requests object from Registry
4. Registry returns remote reference
Skeleton
Server
"Fred"

RMI Flow



5. Client invokes stub method
6. Stub talks to skeleton
7. Skeleton invokes remote object method

Steps to developp a RMI application

1. Define the Remote interface (**`Xyy.java`**) ;
2. Create the implementation of this interface (**`XyyImpl.java`**) ;
3. Compile the class (**`javac XyyImpl.java`**) ;
4. Cerate the server application (**`XyyServer.java`**) ;
5. Compile the server application;
6. Create the Stub and skelton classes using **`rmic XyyImpl_Stub.java`** and **`XyyImpl_Skel.java`** (**`Skel`** does not exist do java version >1.2) ;
7. Run the registry using **`rmiregistry`** ;
8. Run the server that cerate object and register it on the **`rmiregistry`**;
9. Create the Client class that call the remote method (**`XyyClient.java`**) ;
10. Compile and run the client class.

Inversion of a character string using a distributed object

Remote invocation of the reverseString () method of a distributed object that reverses a character string supplied by the caller.

We define :

- **`ReverseInterface.java`** : The interface that describe the Remote Object
- **`Reverse.java`** : The implementation of the Remote Object
- **`ReverseServer.java`** : The RMI server
- **`ReverseClient.java`** : The client that uses the Remote Object

Needed files

Client side

- Interface : `ReverseInterface`
- Client: `ReverseClient`

Server Side

-Interface: `ReverseInterface`
- Object: `Reverse`
- Objects server : `ReverseServer`

The interface of the Remote Object

- It is shared by the client and the server;

- It describes the characteristics of the object;

- It extends the Remote interface defined in java.rmi.

Any method on this interface can throws an exception of type RemoteException.
This exception is thrown:
- if connection refused to the remote host
- or if the object no longer exists,
- or if there is a problem during assembly or disassembly.

Interface of the Remote Class

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

public interface ReverseInterface extends Remote {
    String reverseString(String message) throws RemoteException;
}
```

Remote Object implementation

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3
4  public class Reverse extends UnicastRemoteObject implements ReverseInterface {
5      public Reverse() throws RemoteException {
6          super();
7      }
8
9      public String reverseString(String StringOrigine) throws RemoteException {
0
1          int longueur = StringOrigine.length();
2          StringBuffer temp = new StringBuffer(longueur);
3          for (int i = longueur; i > 0; i--) {
4              temp.append(StringOrigine.substring(i - 1, i));
5          }
6          return temp.toString();
7      }
8  }
```

The server

- Client listening program;

- Save the distributed object in rmiregistry
  - Naming.rebind ("rmi: //hote.cnam.fr: 1099 / Reverse", rev);

- We install a security manager if the server is required to load classes (useless if the classes are not loaded dynamically)
  - System.setSecurityManager (new RMISecurityManager ());

The server

```
1  import java.rmi.*;
3
4  public class ReverseServer {
5      public static void main(String[] args) {
6          try {
7              System.out.println("Server : Object creation");
8              Reverse rev = new Reverse();
9              System.out.println("Reference of Object Reverse sent to RMIregistry");
10             Naming.rebind("rmi://localhost:1099/MyReverse", rev);
11             System.out.println("Waiting for client calls …");
12         } catch (Exception e) {
13             System.out.println("Error binding object");
14             System.out.println(e.toString());
15         }
16     }
17 }
```

The client

- The client obtains a stub to access the object by an RMI URL

```
ReverseInterface ri = (ReverseInterface) Naming.lookup
                      ("rmi://sinus.cnam.fr:1099/MyReverse");
```

- An RMI URL begins with rmi: //, the machine name, an optional port number, and the name of the remote object.

```
rmi://hote:2110/nomObjet
```

By default, the port number is 1099 defined (or to be defined)

The Client

- Install a security manager to control dynamically loaded stubs:

```
System.setSecurityManager(new RMISecurityManager());
```

- Gets a distributed object reference:

```
ReverseInterface ri = (ReverseInterface) Naming.lookup
("rmi://sinus.cnam.fr:1099/MyReverse");
```

- Execute an object method:

```
String result = ri.reverseString ("Terre");
```

The Client

```
1  import java.rmi.*;
2
3  public class ReverseClient {
4      public static void main(String[] args) {
5          //System.setSecurityManager(new RMISecurityManager());
6          try {
7              ReverseInterface rev = (ReverseInterface) Naming.lookup("rmi://localhost:1099/MyReverse");
8              String result = rev.reverseString(args[0]);
9              System.out.println("The reverse of  " + args[0] + " is  " + result);
10         } catch (Exception e) {
11             System.out.println("Error when access the remote object.");
12             System.out.println(e.toString());
13         }
14     }
15
16 }
```

The Client

In order for the client to connect to rmiregistry, it must be provided with a security rules file **`client.policy`**.

```
$more client.policy
grant
{
permission java.security.AllPermission;
};
```

```
C:\UL\UL\Master1\Info408\Spring2020\RMI_Reverse\src>javac *.java

C:\UL\UL\Master1\Info408\Spring2020\RMI_Reverse\src>dir
 Volume in drive C is Windows
 Volume Serial Number is 7889-06D6

 Directory of C:\UL\UL\Master1\Info408\Spring2020\RMI_Reverse\src

06/04/2020  11:51 PM    <DIR>          .
06/04/2020  11:51 PM    <DIR>          ..
06/04/2020  11:49 PM                55 client.policy
06/05/2020  12:30 AM               748 Reverse.class
06/05/2020  12:24 AM               494 Reverse.java
06/04/2020  11:46 PM             1,116 ReverseClient.class
06/05/2020  12:30 AM               254 ReverseInterface.class
06/05/2020  12:25 AM               187 ReverseInterface.java
06/05/2020  12:30 AM               885 ReverseServer.class
06/04/2020  11:46 PM               528 ReverseServer.java
```

Compiling and execution

- Compile the sources (interface, implementation of the object, the server and the client):
  - **`sinus> javac *.java`**

- Launch rmi on the implementation class:

```
C:\UL\UL\Master1\Info408\Spring2020\RMI_Reverse\src>rmic Reverse
```

- Run **`rmiregistry`** :

```
C:\UL\UL\Master1\Info408\Spring2020\RMI_Reverse\src>rmiregistry -J-Djava.security.policy=
"client.policy"
```