Distributed objects technology

Case of JAVA :
Java RMI

1) Intro to RMI
2) More in-depth study
3) Advanced Features

1) Intro. Java Remote Method Invocation

From Sockets to Java-RMI

- Goal: **client/server applications**
  - Distributed, and
  - multi-threaded at server side to serve each client efficiently
- Solution: From (manual!) programming by sockets
  - Stream oriented, with launch of 1 thread per client
  - …to server-side functions invocation, automatically launched
- In a Java context, and along a S.O.A philosophy:
  - Portability
  - Polymorphism
  - Dynamic Generation of code
  - Dynamic code/class loading –during execution

Java-RMI

- RMI means Remote Method Invocation
- Introduced from JDK 1.1
- Integrated part of the Java core (API + runtime support)
  - The public part of RMI is in package(s) `java.rmi`
- RMI = RPC in Java + dynamic code loading
- Same concepts of stubs and skeletons of RPC
- Needs to rely upon the Serialization API (generally used for persistence)
- Possible to have RMI interacts with CORBA and DCOM

Basic concepts

- RMI imposes a clear differentiation between
  - Local methods
  - Methods accessible through the network
  - Distinctions along
    - Declaration
    - Usage (light distinction from syntactic viewpoint; less light from semantic viewpoint, see a later lesson)
- An object whose methods are accessible remotely is named a remote object
- The stub (=proxy) is (type-) compatible with the called object
  - It looks the same
- The skeleton is generic

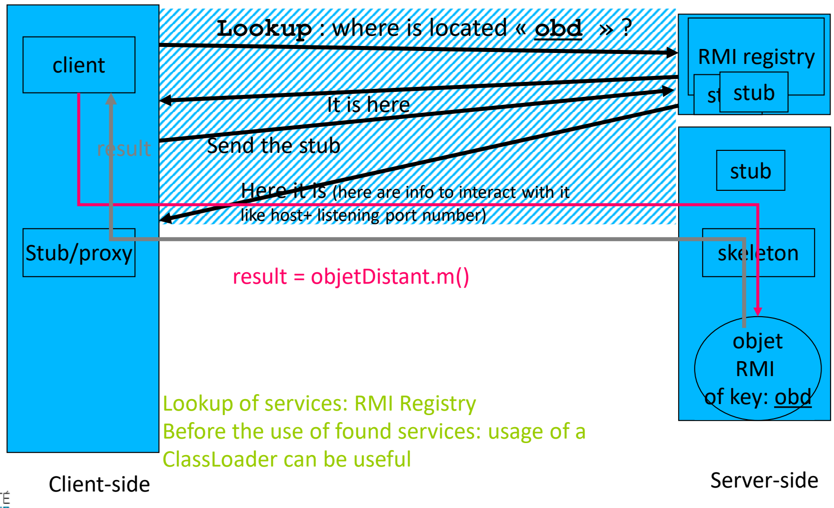
UNIVERSITÉ CÔTE D'AZUR

Some vocabulary and key concepts

- Stubs and skeletons concepts (idem as in RPC)
  - ***Proxy***: *(person with) authority or power to act for another ("mandataire")*
  - ***Stub***: *the short part of something which is left after the main part has been used or left ("morceau restant", "talon/souche")*
    - In distributed programming, the stub in most cases is an interface which is seen by the calling object as the "front-end" of the "remote proxy mechanism", i.e. "acts as a gateway for client-side objects and all outgoing requests to server-side objects that are routed through it".
    - The proxy object implements (in the Java sense) the Stub
  - ***Skeleton*** *(ossature, charpente)*
    - In distributed programming, skeleton acts as gateway for server-side objects and all incoming client requests are routed through it
    - The skeleton understands how to communicate with the stub across the RMI link (>=JDK 1.2 –generic-- skeleton is part of the remote object implementation extending Server class, thanks to reflection mechanisms)

UNIVERSITÉ CÔTE D'AZUR

Interaction Object Client <-> Object RemoteServer



Remote object implementation

- The only remotely accessible methods are those listed in the Remote interface(s)
  - Write a specific interface for the object, that extends `java.rmi.Remote`
- Each remote method must announce possibility to raise the exception `java.rmi.RemoteException`
  - Indicates that there could arise some problems due to the interaction happening between two distributed hosts
- The remote object must provide the code of the methods listed in the interface it claims to implement

Remote object implementation

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

public interface MonInterfaceDistante extends Remote {
      public void echo() throws RemoteException;
}
```

This interface indicates that each object that will implement it has a method entitled echo() that can be invoked remotely

Remote object implementation

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

public class MonObjetDistant extends UnicastRemoteObject
implements MonInterfaceDistante {
       public MonObjetDistant() throws RemoteException {}
       public void echo() throws RemoteException{
             System.out.println(« Echo »);
        }
}
```

- The class of the remote object must implement methods of the interface
- Have at least one public constructor (with or without parameters) that raises RemoteException
- Inherit from `java.rmi.server.UnicastRemoteObject`
- Or, other solution, object `obj` is not an RMI one, but exported/exposed on the network using: `MonInterfaceDistante od= UnicastRemoteObject.exportObject(obj,`**`port`**`)`
  - each instance will be associated to a TCP port (entry point to the remore object), and the wished port number can be provided by user if **port** different of 0

Client of a remote object

- To interact with a remote object, one must
  - Know in advance the interface: the set of methods that can be remotely invoked! And all the classes of declared parameters
  - Find it: obtain a stub/proxy that implements the interface and indicates the entry point socket to host+port where object runs
  - Use it (=invoke the offered methods)
- RMI provides a naming service that allows to locate an object through its name: the RMI registry (runs on same host than the remote object)
  - The object registers itself using a name, "well known" by all potential clients
  - Clients ask for a reference (a stub) to this object through name

Use a remote object

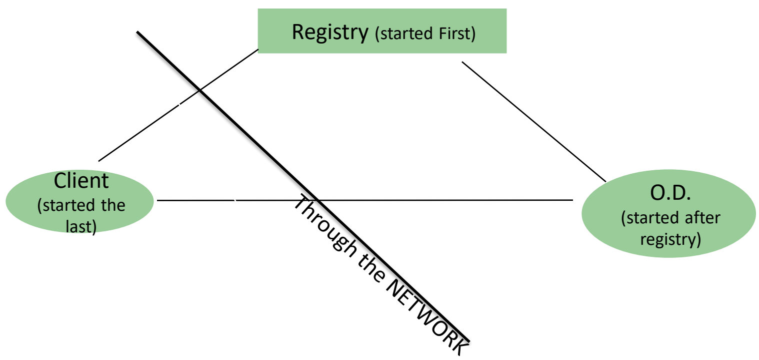
```
import java.rmi.RemoteException;

public class Client {
       public static void main(String[] args) {
           MonInterfaceDistante mod = … // some code to
           //locate the object, i.e. mod is a stub/proxy to object
            try {
                mod.echo();
            } catch (RemoteException e) {
              e.printStackTrace();
           }
      }
}
```

Locate a remote object

- The remote object must first self-register in the registry
  - A program launched on the same host where remote object runs: rmiregistry (executable command in the bin of any JDK installation)
  - Use port 1099 by default, otherwise, add wished port number as param
  - Possible to start it within the main code (class LocateRegistry)
    - Creation/localisation of registry via API java.rmi.registry:
    Registry r = LocateRegistry.createRegistry / getRegistry (numPort) ; //r.rebind(nom)
- Registry acts as a "simplified" directory service:
  - it is indeed simply a naming service (just association name->object)
- Name provided depends of used API: partial or complete URLs
  - protocol://host:port/name (rmi://localhost:2001/**HelloWorld**)
  - Protocol, host and port are optional
    - Ex: Object named *toto* running on the local machine: ///toto
- Usage of the registry via (among others) API java.rmi.Naming
  - Remote object calls Naming.bind or Naming.rebind, or r.rebind
  - Client calls Naming.lookup or r.lookup then cast to Remote interface

Starting an RMI application

- The remote object (O.D.) self-registers into the RMI registry
- The client asks the registry for a **reference** to this O.D.
- The reference is then used to invoke methods on the O.D., exposed through the interface that extends *Remote*



Bytecode generation of stubs and skeletons

- Once the remote object has been written and compiled, it is possible to generate the stub class (and skeleton, but not needed since JDK 1.2!)
- Tool from the JDK installation: *rmic*
  - Takes as only input the full (includes package name) class name of the remote object (in the same directory)
  - Outputs 1 (or 2) file(s) whose name is same as class, plus _Stub (or _Skel)
- Stub only contains methods from the remote interface
  - not all the methods that are declared in the remote object class
- You can see the original source code of the stub, with option –keep
  - rmic –keep className
- Since JDK 1.5, not needed to explicitly generate the stub class!
  - Automatic generation of the stub bytecode, and dynamic class loading
  - as soon as a code manipulates an object that is a stub
    - The stub is an object, so, any JVM manipulating it must know its type/its class

15

RMI: a summary of what to do in practice

- Write the remote interface(s)
- Write the code of the remote object (a single class or one per following items)
  - Implement the interface (and extends UnicastRemoteObject)
  - Add code about registry (in general in the main or in the constructor)
- Compile
- Generate stub et skeleton (optional)
- Write the code of client
  - Obtain a (RMI) reference to the remote object
  - Use the remote methods
- Compile
- Execution:
  - Start rmiregistry THEN start the server
  - Start the client(s)
  - Debug :)

16

Summary - RMI

- A remotely accessible object (remote object) must
  - Have an interface that extends `Remote` and whose methods raise a `RemoteException`
  - Subclass `UnicastRemoteObject` and have a constructor without parameters, that raises `RemoteException`
- To look for a reference to a remote object, one (can) rely upon a naming service, RMIregistry
  - If no use of RMI registry, it is because the RMI reference to the remote object was obtained as a return value of a remote method call towards another remote object
  - There is thus at least one remote object in the whole distributed application that must be registered in the RMIregistry; each client must find this reference using the name of that remote object=> it is an entry point of the whole RMI application

2) More in depth study of RMI

Parameters passing

- The goal of RMI is somehow to mask the distribution
- Ideally, we would like to get same parameter passing semantics in standard Java and in RMI…
- …That is, passing parameters by copy, as in Java:
  - Copy of the value for primitive data types
  - Copy of the reference (address) for objects (e.g. in C, it would mean pass a pointer, that is, the memory address of the memory where object is allocated)
    - Common term used "passing by reference", as the reference is copied
- In Java, objects are never manipulated !
  - The value of any variable is either the one of a primitive data type (eg, an int), or the reference (memory address) to an object

Parameters passing

```
public void foo(int a) {
   a=a+1;
}
```

```
public class MonInt {
  public int i;
  .....
}
public void foo(MonInt a) {
   a.i=a.i+1;
}
```

```
int x = 10;
foo(x);
// what is x value here? ... still 10...
```

```
MonInt x = new MonInt(10);
foo(x);
// what x holds ? Address of object
// what x.i holds ? 11
```

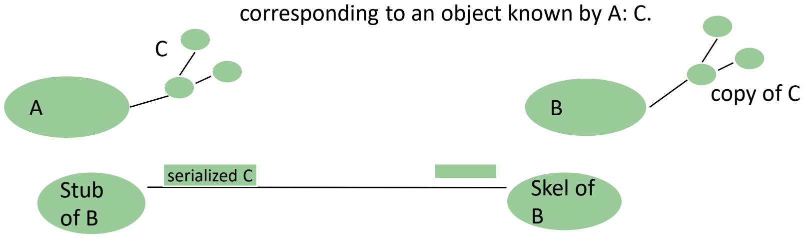
Parameters passing

- Can we do the same in RMI, i.e. a by-copy semantics ?
- Easy for values of primitive types:
  - Send values onto the network, values are automatically copied
  - It is the stub role to do such copies
    - when invoking a method, primitive type values of parameters are copied on the socket
- Little bit more complex for objects to copy them
  - Send the value of the object (only if object is serializable)
- Passing reference to an object may also be required
  - RMI allows it, but only for references to remote objects
  - What is a reference to a remote object ? Its stub!
  - So, simply copy the stub object (which is instance of a serializable class ☺)

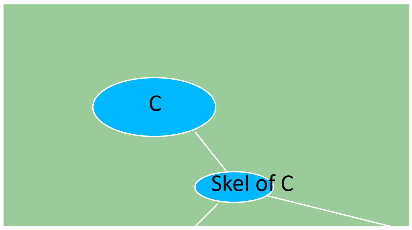
Parameters passing

- Copying a reference to a standard, non RMI object = copying the object itself by a deep-copy (serialization) mechanism

Ex: A calls a method on B, with a parameter corresponding to an object known by A: C.

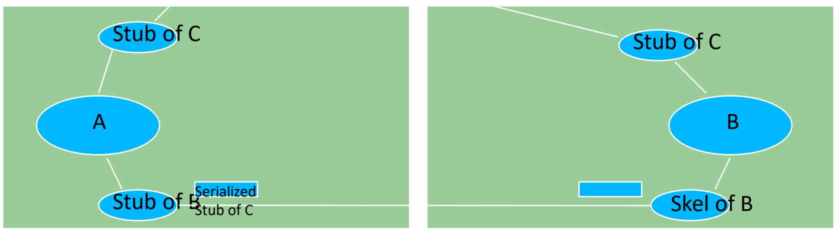


Parameters passing



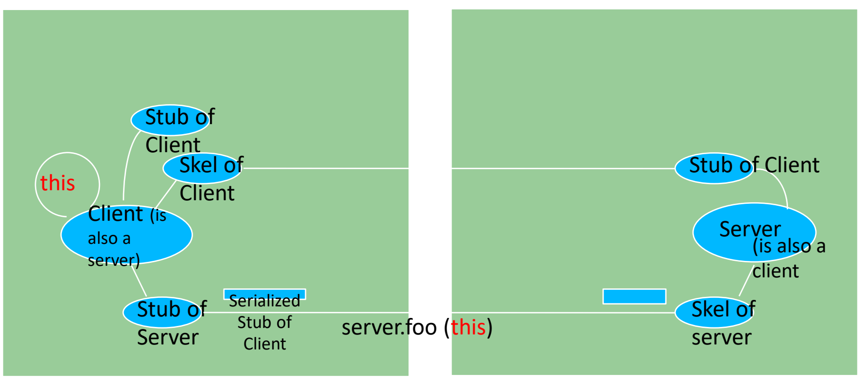
References to remote (RMI) objects are copied= the stub is copied

Ex: A calls a method on B, with a parameter corresponding to a RMI reference to an object C (which is a remote object)



Parameters passing

A local reference (this) of an object that is indeed a remote object is automatically converted to an RMI reference (i.e. to a stub).
When passing *this* as parameter, the stub is copied



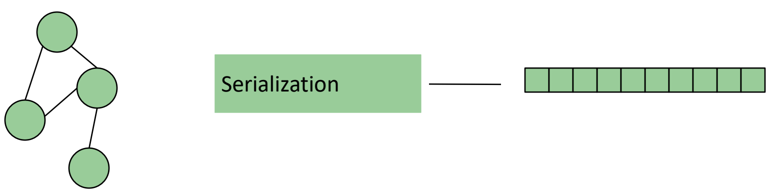
Summary

- Each variable declared as a primitive Java type is passed by copy
- Each standard object is passed by (deep) copy
- Each remote object is passed by reference: its RMI reference is copied

- But, how to copy an object?
  - Not only copy the object
  - Also copy all the objects it references
- Very tedious if done by hand
- Hopefully, an API does exist: the Serialization API

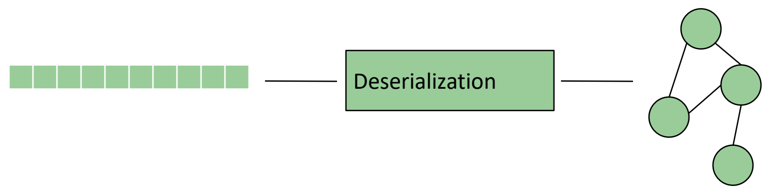
Serialization

- A generic mechanism that transforms an object graph into a stream of bytes
  - The object passed as parameter is converted to an array
  - Also, all the objects it may reference
  - This is a recursive process (=> so the name “deep” copy)
- Basic behaviour
  - Encode the class name
  - Encode attributes values



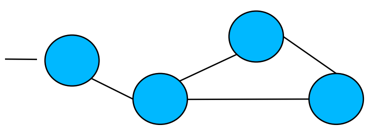
Deserialization

- A process that is symmetrical to the serialization
  - Takes as input a stream of bytes
  - Create the corresponding object graph
- Basic behaviour
  - Read the class name
  - Build an object instance of that class
    - This assumes that the .class is known …
  - Read all attributes values from the stream, to update their value in the new instance



Cycle management

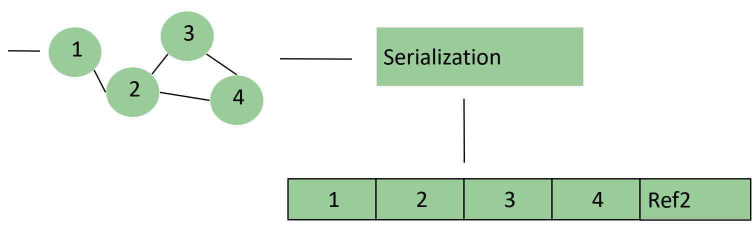
- Serialization is a recursive process
- What happens when there exists a cycle in the object graph?



A naive algorithm would loop for ever
Solution:
Detect cycles
The serialization process remembers which objects have already been serialized in the byte stream
Instead of serializing a given object one more time, put instead its index in the byte stream

Cycle management

- The byte stream contains 3 types of information
  - The class name
  - The values of attributes
  - Some references to other parts in the stream



Usage of the serialization

- By default, an object is <u>not</u> serializable
  - Security protection, because the serialization ignores access rights (the attributes, even if declared private, are serialized …)
  - Raise a *NotSerializableException*
- It is needed to explicitly indicate that an object is serializable
- Put a marker at class level
  - All instances of that class are serializable
  - All instances of a sub class are also serializable
- Usage of the *java.io.Serializable* interface
  - Just a **marker** Interface: no method at all to implement!

Usage of the serialization

- RMI uses the serialization
  - In a totally transparent way
- But a programmer can also use it manually
  - Can be quite practical to copy objects
- Steps
  - Verify the object is instance of a serialized class
  - Create input et output streams
  - Use these streams to create object streams: object input and object output streams
  - Pass the object to serialize to the output stream object
  - Read it from the input stream object

Example: How to use serialization

```
static public Object deepCopy(Object oldObj) throws Exception
{
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;
    try {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
         oos = new ObjectOutputStream(bos);
         // serialize and pass the object
        oos.writeObject(oldObj);
        oos.flush();

         ByteArrayInputStream bin = new ByteArrayInputStream(bos.toByteArray());
          ois = new ObjectInputStream(bin);
        // return the new object
        return ois.readObject();
    }
    catch(Exception e)  {
        System.out.println("Exception in ObjectCloner = " + e);
        throw(e);
    }
    finally {
        oos.close();
        ois.close();
    }
}
```

Fine tuning of the serialization process

- Mark a class with the Serializable interface means all its attributes will be serialized
- This by default behaviour may not be acceptable because of
  - Security
  - Efficiency (why copying something that may be more rapidly re-computed?)
- Possible to apply a more finer grain control
  - Mark an attribute as being non serializable: *transient* keyword
  - Or, give to an object ability to be serialized in an ad-hoc way

Control of the serialization

- To modify the by-default serialization, two methods in the class of the object must be implemented/overridden
  - `writeObject()` : serialization
  - `readObject()` : deserialization
- Their signature is
  - `private void writeObject(ObjectOutputStream s) throws IOException`
  - `private void readObject(ObjectInputStream o) throws ClassNotFoundException,IOException`
- They will be automatically invoked by the serialization process, hiding the by-default behaviour
- In the methods, put some code that is specific to the class the object is an instance of

Control of the serialization

- In the readObject/writeObject methods, it is possible to do anything
  - No limit in theory
  - Manipulation & modification of the attributes of the object is possible
- Based on the streams (from java.io API)
  - FIFO Implementation
  - So, the reading is to be done in the same order as the writing was done
- Symmetric
  - Normally, read all that has been written

Write – Read steps

- Utilisation of methods from ObjectOutputStream and ObjectInputStream
  - Primitive types
    - {write|read}Double, {write|read}Int…
  - Objects
    - {write|read}Object
    - Automatically provokes a new serialization
- Possible to use the "old" implementation
  - The defaultWriteObject() and defaultReadObject() methods on streams
  - Very easy to add a new functionality

Example 1: reproduce the by-default behaviour!

```
public class Defaut implements Serializable {
  public Defaut() { }

  private void writeObject(ObjectOutputStream s) throws
                                            IOException {
        s.defaultWriteObject();
    }

  private void readObject(ObjectInputStream s) throws
                       IOException, ClassNotFoundException {
     s.defaultReadObject();
  }
}
```


Example 2: save a single attribute (not all)

```
public class Defaut implements Serializable {
   private int valeur;  // THE ONLY ATTRIBUTE TO SAVE
   private double valeur2
   public Defaut() { }

   private void writeObject(ObjectOutputStream s) throws
                                            IOException {
        s.writeInt(valeur); // WRITE the ATTRIBUTE
    }

   private void readObject(ObjectInputStream s) throws
                       IOException, ClassNotFoundException {
     valeur = s.readInt();  // READ ATTRIBUTE from the STREAM
  }
}
```

Serialization and inheritance

- The sub-classes of a serializable class are serializable
- But, a class can be serializable whereas its ancestor may not
  - The sub-class is responsible of the saving/restoring of the inherited attributes
  - During the deserialization, the by-default constructor is invoked, to initialize the non-serializable attributes
    - Thus, the constructor of the parent class will be invoked too, etc
- Source of hard-to-identify bugs

3) Advanced features of RMI

Distribution of classes

- RMI makes distinction between two object types
  - Those that can be accessed remotely
  - The others
- They are most of the time on different hosts (client hosts, server host)
- How are the **classes** (initially) distributed (+- mapped on the network)
  - Classes at Client side:
    - Implementation of the client
    - Remote Interface with all necessary classes used as types in methods signature
    - Stub class if not dynamically generated (the serialized stub object is obtained by registry lookup or by a method call return)
  - Classes at Server side:
    - Remote Interface with all necessary classes used as types in methods signature
    - Implementation of server object(s) using possibly some subclasses of the declared classes (sub-classes for remote methods parameters)

On-demand Class downloading

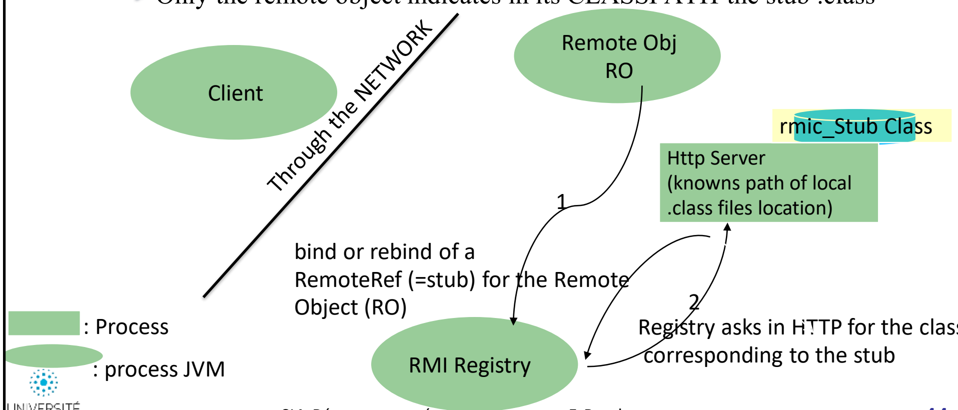
- In a perfect world
  - all classes are already distribued on the various machines
  - Nothing changes and everything is known in advance
- But, in practice
  - Classes are *more or less well* distributed, in locations where we guess they will be needed
    - Some hosts may not possess some useful classes
  - Ex: Call of a remote method passing as parameter an object instance of a sub class of the declared class in the method
    - The server needs to download the .class file, present at client host side in order to know how to deserialize this parameter
- Solution: capability to download any missing class

On-demand Class downloading

- A mechanism part of the RMI technology
- Use of HTTP
  - Allows to cross firewalls
  - But requires an HTTP server, running on any location on the network ☺
- Principle:
  - Serialisation streams are annotated with a *codebase*
  - This indicates where a necessary class may be downloaded if missing
  - During the deserialization, if a class misses, the HTTP server whose URL is indicated as codebase is contacted
  - If the class is available and can be downloaded from the HTTP server, the program continues execution, otherwise, it raises a ClassNotFoundException

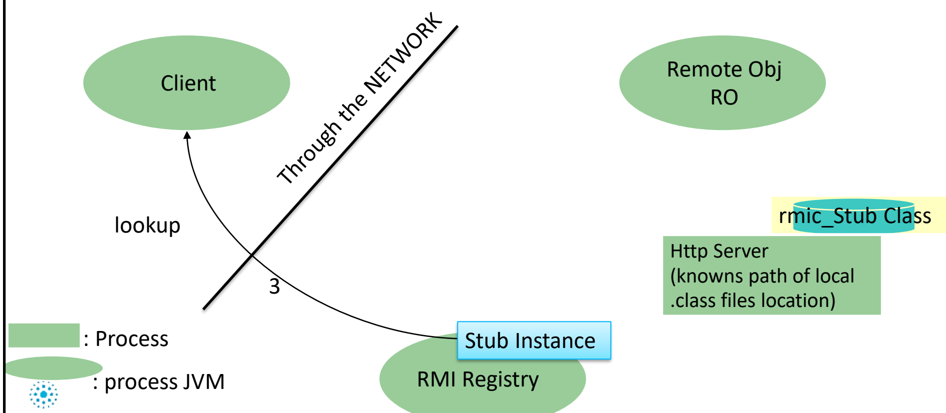
Simple Example: principle

- Minimal deployment & under hypothesis that the stub class has been created using rmic, and not in an automatic way
  - The stub class is then needed in any JVM that manipulates a stub
  - The stub class is not available at registry nor at client sides
    - Only the remote object indicates in its CLASSPATH the stub .class

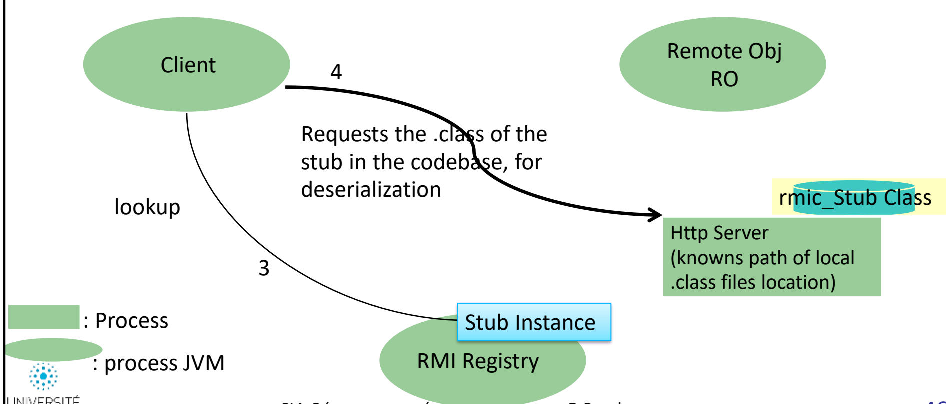


Example simple: principle

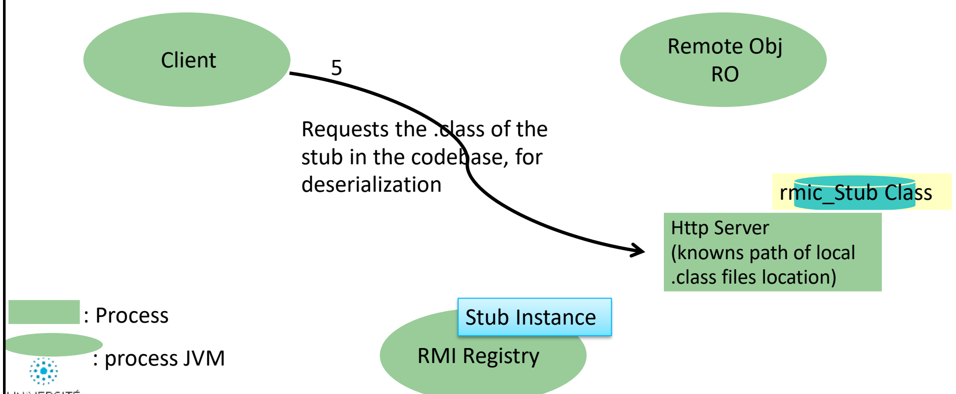
- When the client lookups for the stub in the RMI registry, it then needs the .class of the stub for deserialization
- Where is this class located?



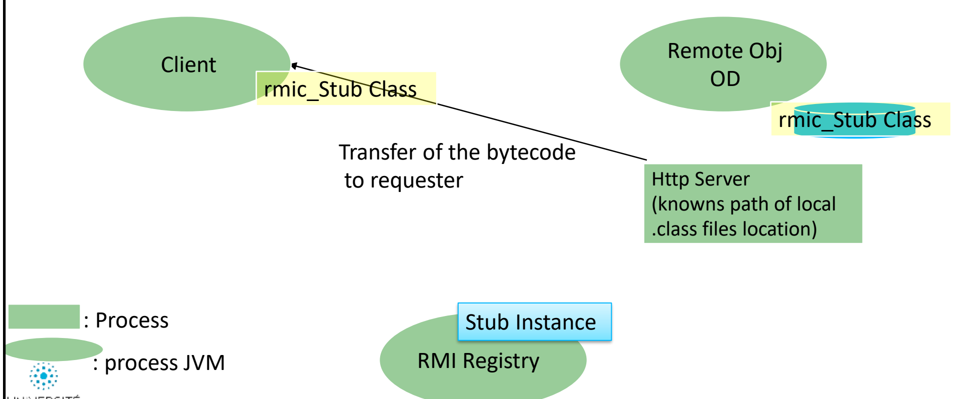
Example simple: principle



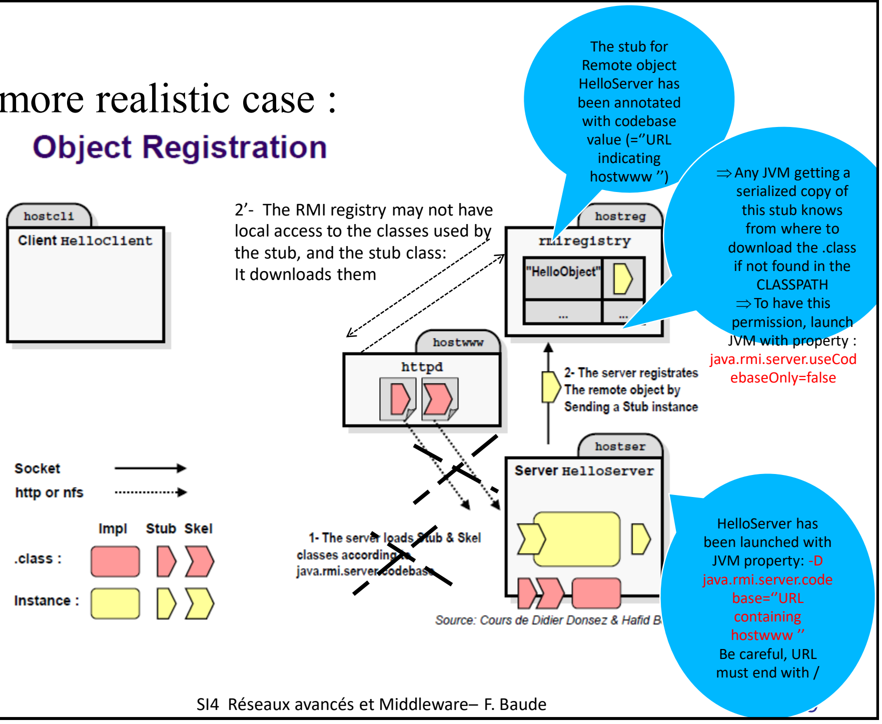
Example simple: principle



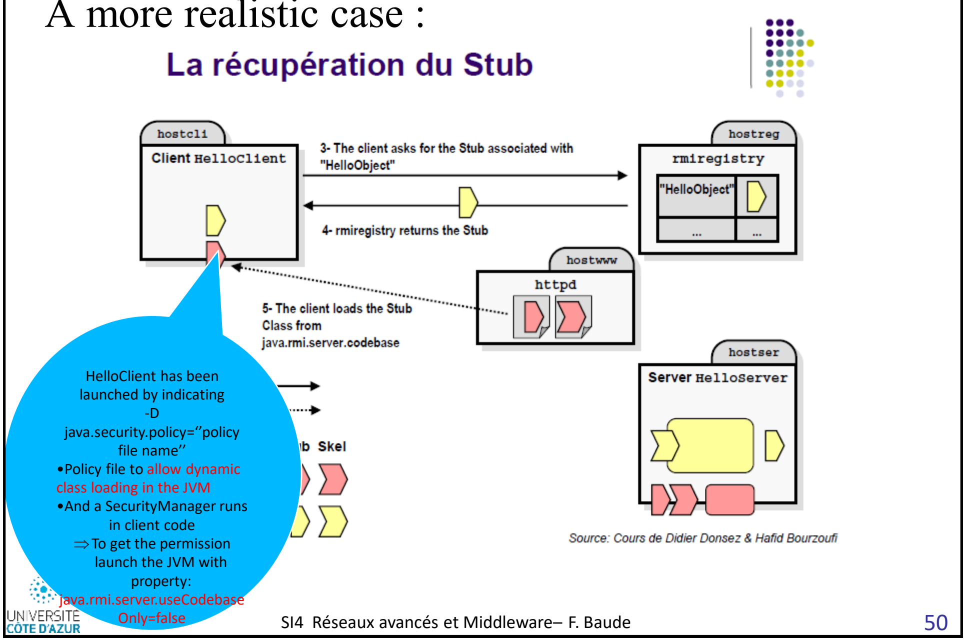
Example simple: principle



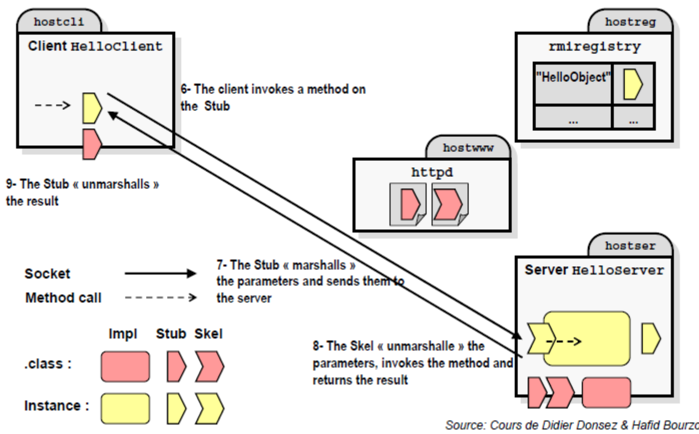
A more realistic case :
Object Registration



A more realistic case :
La récupération du Stub

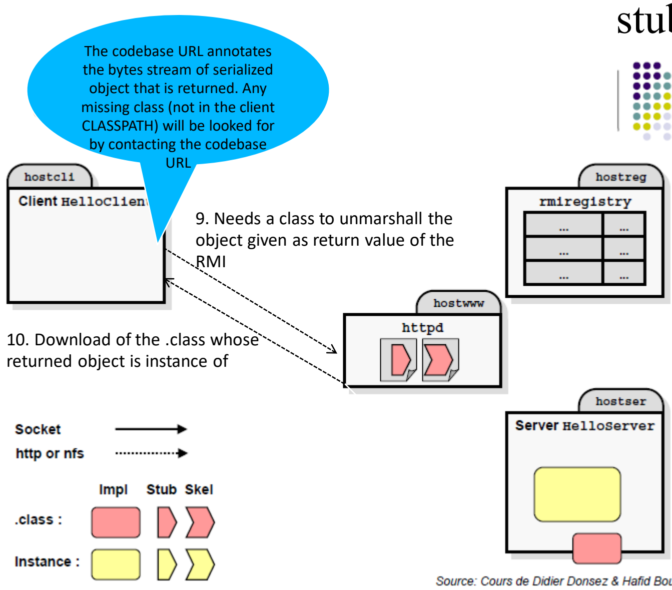


A more realistic case :
Remote Method Invocation



Source: Cours de Didier Donsez & Hafid Bourzoufi

…RMI requiring to download other classes than stub classes



Source: Cours de Didier Donsez & Hafid Bourzoufi

Summary of dynamic class loading from web serverS

- Launch JVM with JVM property java.rmi.server.codebase=”http://hostWWW:portWWW/”
  - Each serialized stream from that JVM is annotated with http://hostWWW:portWWW/
- Each received stream by a JVM that has no knowledge of .class to deserialize the stream:
  - Starts an HTTP communication with http://hostWWW:portWWW/
  - to download missing .class file(s), from another code base than the folders listed in this JVM CLASSPATH
    - Permission to do so: the JVM must have been launched with
      - Property and value: **java.rmi.server.useCodebaseOnly=false**
      - A security manager object, controled by a policy file giving permission to open an HTTP connection towards hostWWW:portWWW
- The RmiRegistry JVM can be launched without any CLASSPATH
  - Any serialized stub coming from any JVM server is deserialized using the needed .class corresponding file(s), downloaded from the indicated web server

53

53

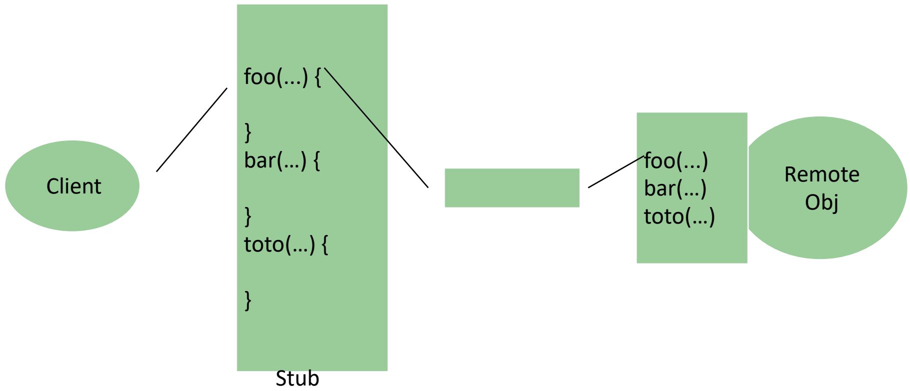
The Stub

- The stub role is to act as it would be the remote object, it is a proxy
  - It implements the Remote interface
  - Example of what it displays when invoking toString() : Proxy[HelloWorld,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:[193.51.208.206:10003](remote),objID:[75300dd9:145187ca75d:-7fff, 2812291573724520304]]]]]
- Stub has to convert any remote method invocation into a stream
  - Easy for call parameters
  - For the method name: coded on on a few bytes, according to agreement with the skeleton
- Then has to wait for the return of that method invocation
  - Reading from a socket, then deserialization
- So, a stub object is quite simple!
  - Its bytecode can be generated by the JVM that needs to deserialize the received stream
    - Necessary condition: the JVM must know the .class file of the remote interface and all parameters’ classes declared in method signatures…

4

54

The Stub



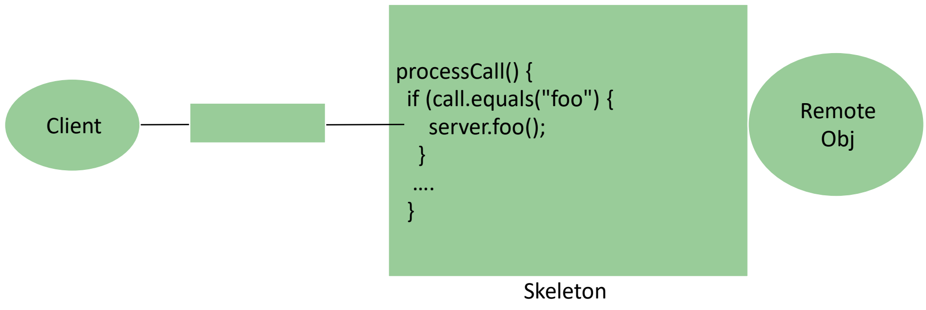
55

The Skeleton

- The skeleton role is to invoke methods on the object, locally
- And return back the result
- Is a skeleton dependant of the remote object definition?
  - Yes, only if its implementation is hard-coded (naïve case) and not generic

56

The Skeleton
(a naïve version)



57

The Skeleton

- Is there a way to consider the skeleton as being independent of the called object ?
  - Yes if there exists a way to say "I want to call the method whose name is *foo*" without having to put *foo* as method name explicitly
- Reflection
  - Capability that a code has to observe itself, or modify its own structure
  - Concretely, a reflective language allows one to manipulate some objects that do represent method invocations, attributes, . …
  - Programmer or runtime platform builds an object that represents a method invocation, and triggers its execution
  - Reflection is an integral part of Java. It is key for RMI, and for the serialization/deserialization Java processes

58

Reflection Example

```
String firstWord = "blih”;
String secondWord  = "blah";
String result = null;
Class c = String.class;
Class[] parameterTypes = new Class[] {String.class};
Method concatMethod;
Object[] arguments = new Object[] {secondWord};

concatMethod =  c.getMethod("concat",parameterTypes);
result =
(String)concatMethod.invoke(firstWord,arguments);

// what runs: result=firstWord.concat(secondWord);
```


Reflection : RMI usage

Stub side: describe the method to be invoked at server side as an object

```
    Method m = … getMethod(“sayHello”);  // server offers a
method sayHello that will be invoked by calling invoke of m on the
remote reference (named as ref)
```

It is the generic aspect of the skeleton

```
// Stub class generated by rmic, do not edit.
public final class HelloWorldImpl_Stub extends
java.rmi.server.RemoteStub implements HelloWorld, java.rmi.Remote
$method_sayHello_0 = HelloWorld.class.getMethod("sayHello", new
java.lang.Class[] {})
public java.lang.String sayHello()  throws java.rmi.RemoteException
    {
    try {
        Object $result = ref.invoke(this, $method_sayHello_0, null,
6043973830760146143L);
        return ((java.lang.String) $result); // the RMI is synchronous
    } catch (java.lang.RuntimeException e | java.rmi.RemoteException e)
        throw e;
    } catch (java.lang.Exception e) {throw new
    java.rmi.UnexpectedException("undeclared checked exception", e);
    }}
```

RMI and threads

- A Remote Method Invocation is triggered by a thread at caller side
- But run by another thread, at callee side
- The caller side thread is blocked until the thread at callee side terminates and returns the result (or void)
  - This will make the stub object at caller side resumes its execution
- If many callers, multiple threads run at callee side
  - A remote object is by essence a multithreaded object
  - One must manage concurrency between these threads, by adding necessary synchronization Java mechanisms (synchronized, wait, notify, etc)
- There is no explicit link between a caller thread and the thread at callee side

RMI and threads

- The implementation regarding threads is not specified
- In practice
  - When a method call arrives (at skeleton side), RMI creates a thread to handle that call
  - Once the call has finished, that thread can be recycled to serve a future method invocation
  - If many calls arrive “simultaneously”, new threads may be created
- Known as the Thread-pool technic
- Problems raised by re-entrant method calls
  - A makes a remote call on remote object B, which makes a remote call to A
  - This creates a cycle within the graphs of involved objects
  - This does not raise problems in most of the cases (besides latency)
  - But it can raise deadlocks if methods are synchronized (not reentrant)

Same synchronization monitor

A (distributed) Deadlock

CLIENT (also server)

```
public synchronized foo() {
   .... server.bar() //foo is blocking
}

public synchronized toto() {
  ......
}
```

1

2

SERVER implements bar

```
public  bar( ) {
   client.toto() // blocking
}
```

2 remote objects interact
There is an oriented cycle in the graph representing distributed method invocations
The RMI thread (step 2) cannot enter into toto() method while the other thread on CLIENT has not yet terminated execution of foo(), as the synchronization monitor associated to CLIENT is occupied
=> DEADLOCK !
A situation sometimes hard to identify
- As no global view of the application exists because it is distributed
- No information from the JVMs about such scenarii

See Elective course of SI5 : "A distributed approach of distributed systems" solutions !

Principle of RMI Distributed Garbage Collector (DGC)

- Based upon the standard GC of each involved JVM
  - Counting object references: when an object is not anymore referenced from a 'root' object, it can be deallocated from the JVM memory
- When a stub is received by a JVM, the corresponding remote object becomes referenced:
  - The receiving JVM increments the remote-references counter of the remote object (by sending a specific message to the JVM hosting the remote object)
- When <u>the stub is not anymore used</u>, it should have as side-effect to decrement the remote-references counter of the remote object:
  - [in theory] Happens when the object holding the stub address in an attribute is itself deallocated, or when that attribute is overridden.
  - [in practice] Happens if the client does not make use of the stub within a certain time period ('lease'=bail period), whose duration can be set via *java.rmi.dgc.leaseValue=tms*
    - When the lease (managed at remote object / server side) for a given reference reaches 0, the server-side GC decrements by 1 the remote-references counter
    - To avoid the removal of the remote object, the only way at client side is to regularly invoke remote methods!
- When the remote-references counter held at server side reaches value 0:
  - **The remote object** is marked by the GC as "**garbageable**"
  - It will be deallocated by the standard Java GC only when no more local references to the object locally exist

What about stubs registered in the RMI registry?

- RmiRegistry is itself an RMI server/remote object that stores stubs
- While a stub is kept in the RMI Registry, this (should) **blocks the GC of the remote object** referenced by this stub
  - the RMI registry (acting as a RMI client) regularly reconducts its lease, so that the remote object gets not garbaged,
    - otherwise, the stored stub at registry side may reference an object that does not exist anymore
- But… as soon as there exists a remote object, for which a RMI reference (a stub) has been bound using *bind* in the registry, it is ready to receive method invocations, and holds resources in its JVM
  - This could be useless if no clients invoke the remote object !!
  - **Activatable (remote) object** concept:
    - instantiated only at the moment a first client invokes a method remotely
    - Specific type of stub stored in RMI registry, even if remote object not yet created
    - Usage of such kind of stub triggers the remote object instantiation by the RMI deamon of the remote/server side JVM