Università degli Studi di Bologna
Facoltà di Ingegneria

Principles, Models, and Applications for Distributed Systems M

Java RMI

(Remote Method Invocation)

Luca Foschini

RMI: motivations and main characteristics

Java Remote Procedure Call (RPC): RMI allows remote **Java methods execution** seamlessly integrated with **OO paradigm**.

Definition and main characteristics

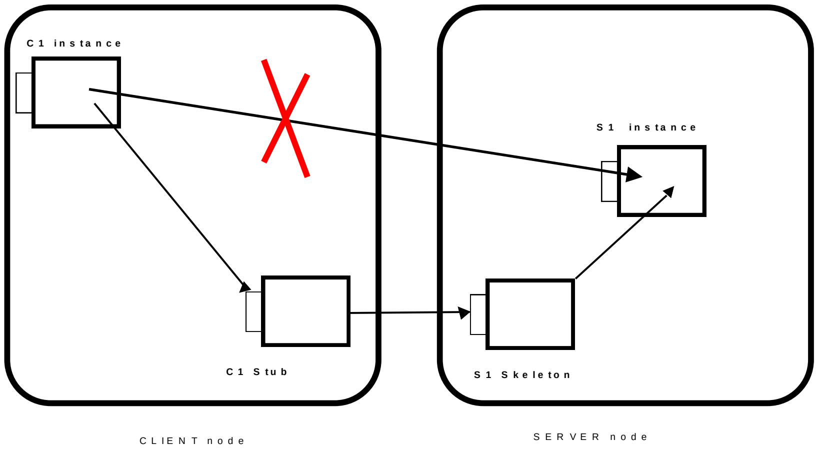
Set of **tools, policies, and mechanisms** that allow a Java application to **call the methods of an object instatiated on a remote host**.

- RMI locally creates a **reference to remote object** (instantiated on a remote host).
- The client application calls the needed remote methods using this **local reference**.

Single working environment on **heterogeneous systems** thanks to **Java bytecode portability**.

Remote objects access

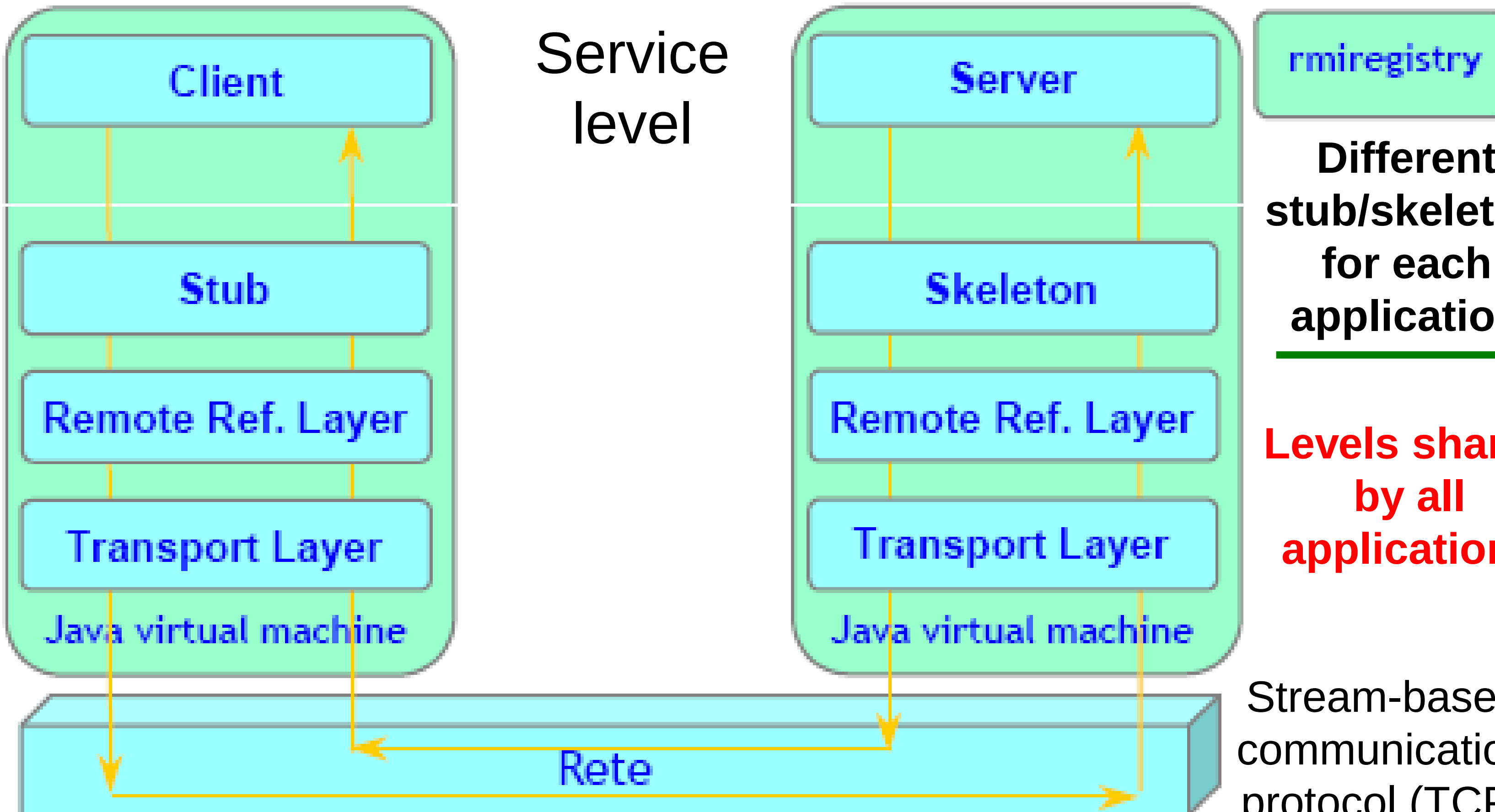
- Java **does not directly provide remote references**, but **using RMI** it is possible **to built them**.
- Remote Method Invocation
  - Two **proxies**: **stub** client-side and **skeleton** server-side
  - **Proxy pattern**: these components hide the distributed aspect of the application.



- What are the differences respect to calls to methods of a local object?
  - Reliability, semantics, duration, ...
- **NOTE: it is not possible** to directly refer to a remote object → **need of an active distributed framework**

RMI architecture

Only SYNCHRONOUS and BLOCKING interactions



Different stub/skeleton for each application

Levels shared by all applications

Stream-based communication protocol (TCP)

RMI layered architecture

- **Stub e skeleton**:
  - **Stub**: **local proxy** that receives method invocations on behalf of the remote object
  - **Skeleton**: **remote entity** that receives method invocations made on the stub and invokes them on the real object
- **Remote Reference Layer (RRL)**:
  - Manages remote references, parameters and stream-oriented connection abstraction
- **Transport Layer**
  - Manages connections between different JVMs
  - Can use different transport protocols, as long as they are connection-oriented → typically **TCP**
  - Uses a proprietary protocol
- The **name system, Registry**: **name service** that allows to the server to publish a service and to the client to obtain the proxy to access it.

RMI features

Distributed objects model

For the Java distributed objects model, a **remote object** is:

- **An object whose methods** can be invoked from another JVM, that may be running on a different host;
- The object is described by **remote interfaces** that declare available methods.

Local invocation vs. remote invocation

The client invokes **remote object methods** using the **remote reference** (**interface variable**)

- **Same syntax** →transparent

  Always synchronous blocking invocation
- **Semantic**: different
  - Local invocation →maximum reliability
  - Remote invocation: communication could fail

    → **"at most once" semantic** (using CTP)
  - Remote server: each invocation is processed **independently** and **parallel to others (typically multi-threaded parallel servers)**

Interfaces and implementation

A few practical observations

- Separation between
  - Behavior definition → interface
  - Behavior implementation → class

- **Remote components** are referred to using interface variables
  1. Behavior **definition** using
     - an interface that must extend java.rmi.Remote
     - each method must declare that it may throw java.rmi.RemoteException
  2. Behavior **implementation**, class that
     - implemenst the previously described interface;
     - extends java.rmi.UnicastRemoteObject

Steps to develop a Java RMI

1. **Define** **interfaces** and **implementations** of the component to be used from remote hosts
2. **Compile** the classes (using javac) and **generate** **stub** and **skeleton** (using rmic) of the classes to be used from remote hosts
3. **Publish** the service on the **registry** name service
   - start the **registry**
   - register the **service** (the server must send a bind request to the registry)
4. **Obtain** (client-side) the reference to the remote object sending a **lookup request** to the **registry**

After the last step, **client and server** can interact.

Note: this is a **simplified workflow**, next slides will give more details on the registry and dynamic class loading.

Implementation: interface

- The interface **extends the Remote interface**
- Each method:
  - Has to declare that it could throw a **RemoteException**, i.e. remote method invocation is **NOT** completely transparent
  - Returns **only one** result and has **zero, one or more input** parameters (no output parameters)
  - Accepts parameters either **by value** (**primitive** data types or **Serializable** objects) or **by reference** (**Remote** objects) → more details in the following slides.

```
public interface EchoInterface
      extends java.rmi.Remote {

   String getEcho(String echo)
      throws java.rmi.RemoteException;
}
```

Implementation: Server

The **class** that **implements the server**

- Has to extend the UnicastRemoteObject class
- Has to implement **all the methods declared by the interface**

A **process running** on the server host registers all the services:

- Makes as many bind/rebind as the **server object** to register, each one with a **logic name**

Registering service

- Accepts bind and rebind requests **only** by the local registry

```
public class EchoRMIServer
 extends java.rmi.server.UnicastRemoteObject
 implements EchoInterface{

// Costruttore
 public EchoRMIServer()
  throws java.rmi.RemoteException
  { super(); }

 // Implementazione del metodo remoto
      dichiarato nell'interfaccia
 public String getEcho(String echo)
  throws java.rmi.RemoteException
  { return echo; }

 public static void main(String[] args){
  // Registrazione del servizio
try
 {
  EchoRMIServer serverRMI =
          new EchoRMIServer();
  Naming.rebind("EchoService", serverRMI);
 }
catch (Exception e)
{e.printStackTrace(); System.exit(1); }
  }
}
```

Implementation: Client

Services used exploiting an **interface** **variable** obtained by sending a **request to the registry**

Lookup of a **remote reference**, namely a **stub instance** of the remote object (using a **lookup** and assigning it to a **interface variable**)

Remote method invocation:

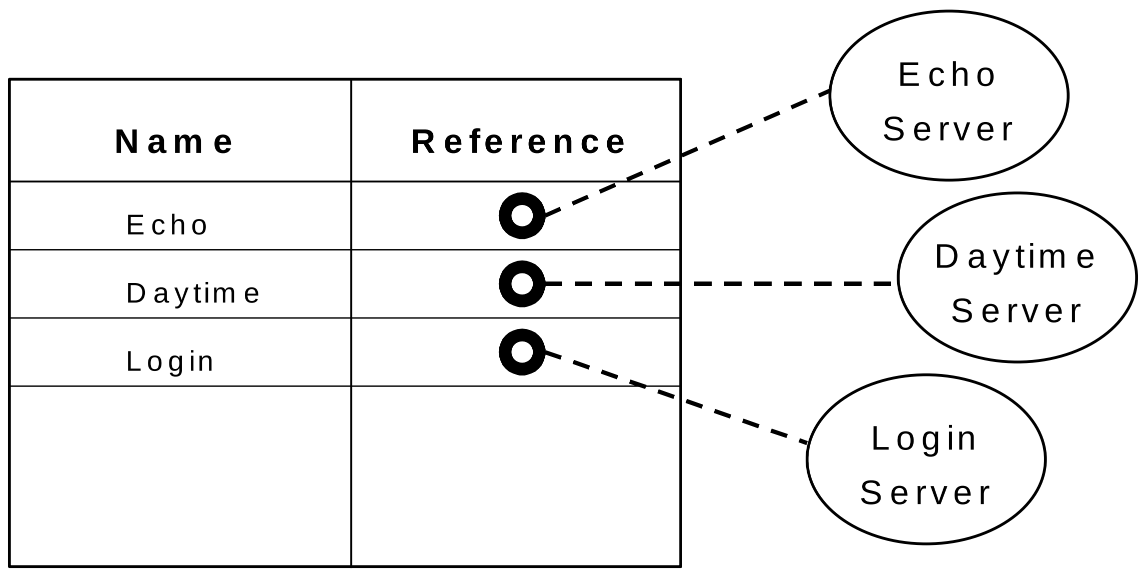
- **Synchronous blocking** method using the parameters declared in the interface

```
public class EchoRMIClient
{
 // Avvio del Client RMI
 public static void main(String[] args)
 {
   BufferedReader stdIn=
    new BufferedReader(
      new InputStreamReader(System.in));
 try
 {
  // Connessione al servizio RMI remoto
  EchoInterface serverRMI = (EchoInterface)
  java.rmi.Naming.lookup("EchoService");

  // Interazione con l'utente
  String message, echo;
  System.out.print("Messaggio? ");
  message = stdIn.readLine();

  // Richiesta del servizio remoto
  echo = serverRMI.getEcho(message);
  System.out.println("Echo: "+echo+"\n");
 }
 catch (Exception e)
 { e.printStackTrace(); System.exit(1);

 }
}
```

RMI Registry

- **Service localization**: a client running on a host that needs a service, has to find a server, running on another host, that provides it.
- Possible solutions:
  - The client knows the address of the server
  - The user manually configurates the client and selects the server's address
  - A standard service (**naming service**) with a well known address that the client knows, takes the forwarder role

- Java RMI uses a naming service called **RMI Registry**
- The Registry mantains a set of couples **{name, reference}**
  - Name: **arbitrary string**
- **There is NO** location transparence



Naming Class and Registry activation

java.rmi.Naming class methods:

```
public static void bind(String name, Remote obj)
public static void rebind(String name, Remote obj)
public static void unbind(String name)
public static String[] list(String name)
public static Remote lookup(String name)
```

Each of these methods **sends a request to the RMI registry** identified by **host** and **port** as location

`name` ⇨ combines the registry location and the **logic name** of the service, formatted as: **`//registryHost:registryPort/logical_name`**

- **`registryHost`** = address of the host where the register is running
- **`registryPort`** = port where the registry is listening (default 1099)
- **`logical_name`** = name of the service that we want to access

There is NO location transparency

Registry activation (on the server): use the **rmiregistry** application, started in a shell on its own, optionally specifying the port to use (default 1099):

`rmiregistry` or **`rmiregistry 10345`**

N.B.: the registry is activated in a new JVM instance

Compilation and Execution

Compilation

1. Compilation **interface** and **classes**

```
javac      EchoInterface.java
           EchoRMIServer.java
           EchoRMIClient.java
```

2. Build **Stub and Skeleton executables** ⇨ EchoRMIServer_Stub.class, EchoRMIServer_Skel.class

```
rmic [-vcompat] EchoRMIServer
```

Note: when using Java 1.5 and above pass the **-vcompat** option to **rmic**

Execution

1. Server side (registry and server)
   - Start registry: `rmiregistry`
   - Start server: `java EchoRMIServer`

2. Execution: `java EchoRMIClient`

Parameters passing – remote

Type	Local Method	Remote Method
Primitive data type	By value	By value
Objects	By reference	By value (`Serializable` interface, deep copy)
Remote object		By remote reference (`Remote` interface)

Local:

- **Copy → primitive data types**
- **By reference → all Java objects ("by address")**

Remote: (problems when referring to non local entities and contents)

- **By value → primitive data types and Serializable Object**
  - Objects whos location **is not relevant to the state** can be passed **by value**: the object is serialized, sent to the destination and deserialized to build a local copy
- **Passing by remote reference → Remote Object** ***via RMI***
  - Object whose utility is bound **to their locality** (server) are passed by **remote reference**: the stub gets serialized and dispatched to the other peer. Each stub instance identifies a single remote object using an identifier (ObjID) which is unique in the context of the JVM in which the target object exists.

Serialization

In general, RPC systems apply a double transformation to **input and output parameters** to solve problems related to heterogeneous representations:

- **Marshalling**: action that codes arguments and results to be transmitted
- **Unmarshalling**: reverse action that decodes arguments and results

Thanks to the use of bytecode (interpreted and independent of the local system), **Java does not need un/marshalling**, the objects are simply de/serialized using mechanisms provided by the language

- **Serialization**: transformation of complex objects into simple byte sequences
  - **writeObject()** method on an output stream
- **Deserialization**: decoding of a byte sequence and building of a copy of the original object
  - **readObject()** method on an input stream

Stub and **skeleton** use these mechanisms to exchange input and output with the remote host

Using streams for object TX/RX

Sample serializable “Record” object written on streams

```
Record record = new Record();
FileOutputStream fos = new FileOutputStream(“data.ser”);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(record);

FileInputStream fis = new FileInputStream(“data.ser”);
ObjectInputStream ois = new ObjectInputStream(fis);
record = (Record)ois.readObject();
```

This technique is appliable only to serializable objects, i.e. objects that:

- Implement the **`Serializable`** interface
- Contain only serializable objects (internal fields)

NOTE:

The real object is NOT transferredo, Java only sends the data that characterize the specific instance

- **no** methods, **no** costants, **no `static`** variables, **no `transient`** variables

Upon deserialization, Java **builds a a copy** of the “received” instance exploiting the .class file (that must be available!) of the object and the data received.

Serialization: example

Modify the echo server

⇨ Message sent as **serializable object** instead of String

```
public class Message implements Serializable
{ String content;
  // … other fields

  // Costructor
  public Message(String msg){ content=msg; }

  public String toString(){ return content; }
}
```

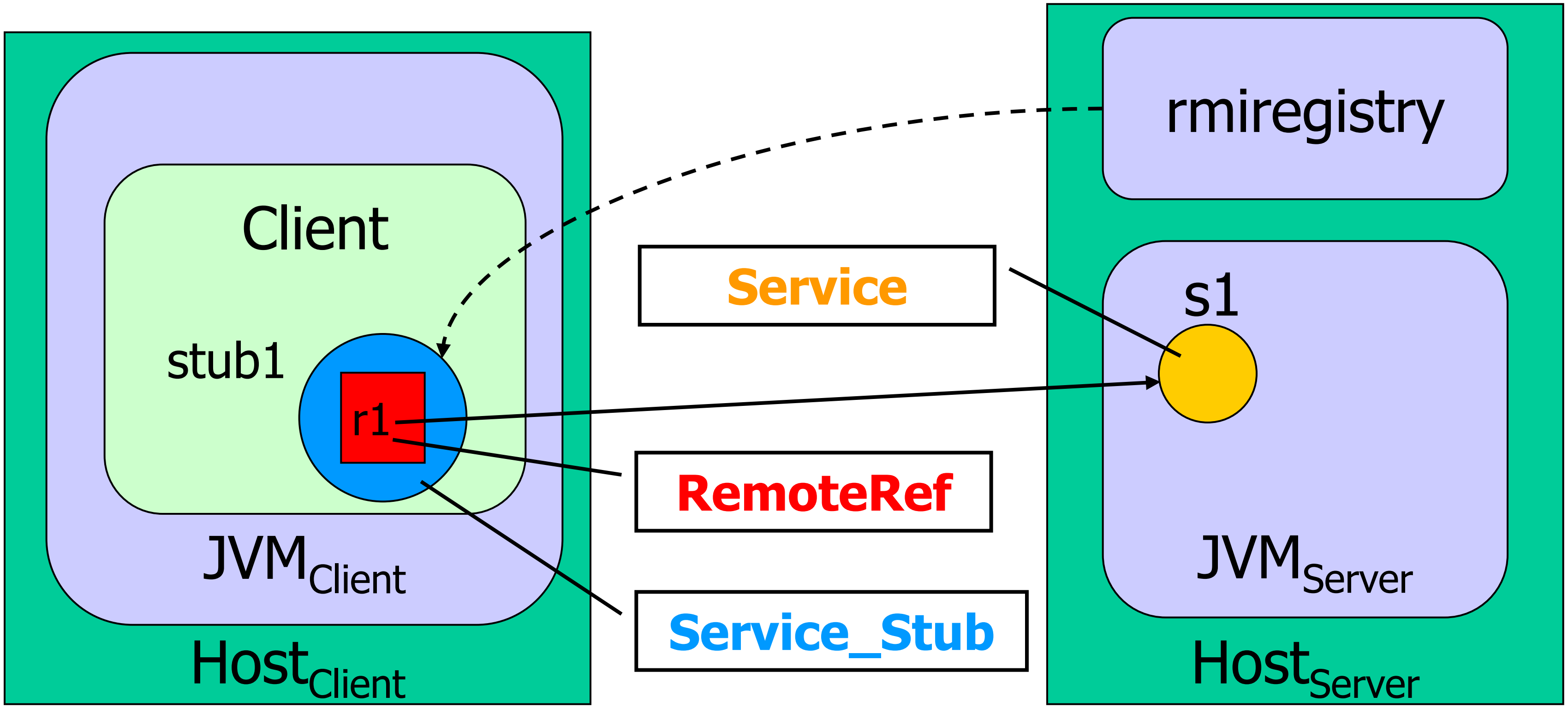
The object gets tranferred as its **whole content**

Stub and Skeleton

- **Stub and Skeleton**
  - Allow calling a remote service as if it was local (they act as proxies)
  - Are generated by the RMI compiler
  - Java environment natively supports de/serialiation
- **Communication algorithm**
  1. The client obtains a **stub instance**
  2. The client calls the desired methods on the stub and waits for result
  3. The **stub**:
     - Serializes the information needed for the method invocation (method id and arguments)
     - Sends informations to the skeleton exploiting the RRL abstractions
  4. The **skeleton**:
     - Deserializes the received data
     - Calls the method on the object that implements the server (dispatching)
     - Serializes the return value and sends it to the stub
  5. The **stub**:
     - Deserializes the return value
     - Returns the result to the client

RMI details

Stub and Remote References



The **Client** uses the **RMI Server** implemented by the **Service** class exploiting the reference to the local stub *stub1* (instance of the **Service_Stub** class provided to the client by the registry)

Service_Stub contains a **RemoteRef** (*r1*) that allows the RRL to reach the server

Registry implementation

The Registry is itself a RMI Server

- Interface: java.rmi.registry.**Registry**
- Class that implements it: sun.rmi.registry.**RegistryImpl**

```
public interface Registry extends Remote {
    public static final int REGISTRY_PORT = 1099;
    public Remote lookup(String name)
      throws RemoteException, NotBoundException, AccessException;
    public void bind(String name, Remote obj)
      throws RemoteException, AlreadyBoundException, AccessException;
    public static void rebind(String name, Remote obj)
      throws  RemoteException, AccessException;
    public static void unbind(String name)
      throws RemoteException, NotBoundException, AccessException;
    public static String[] list(String name)
      throws  RemoteException, AccessException;
}
```

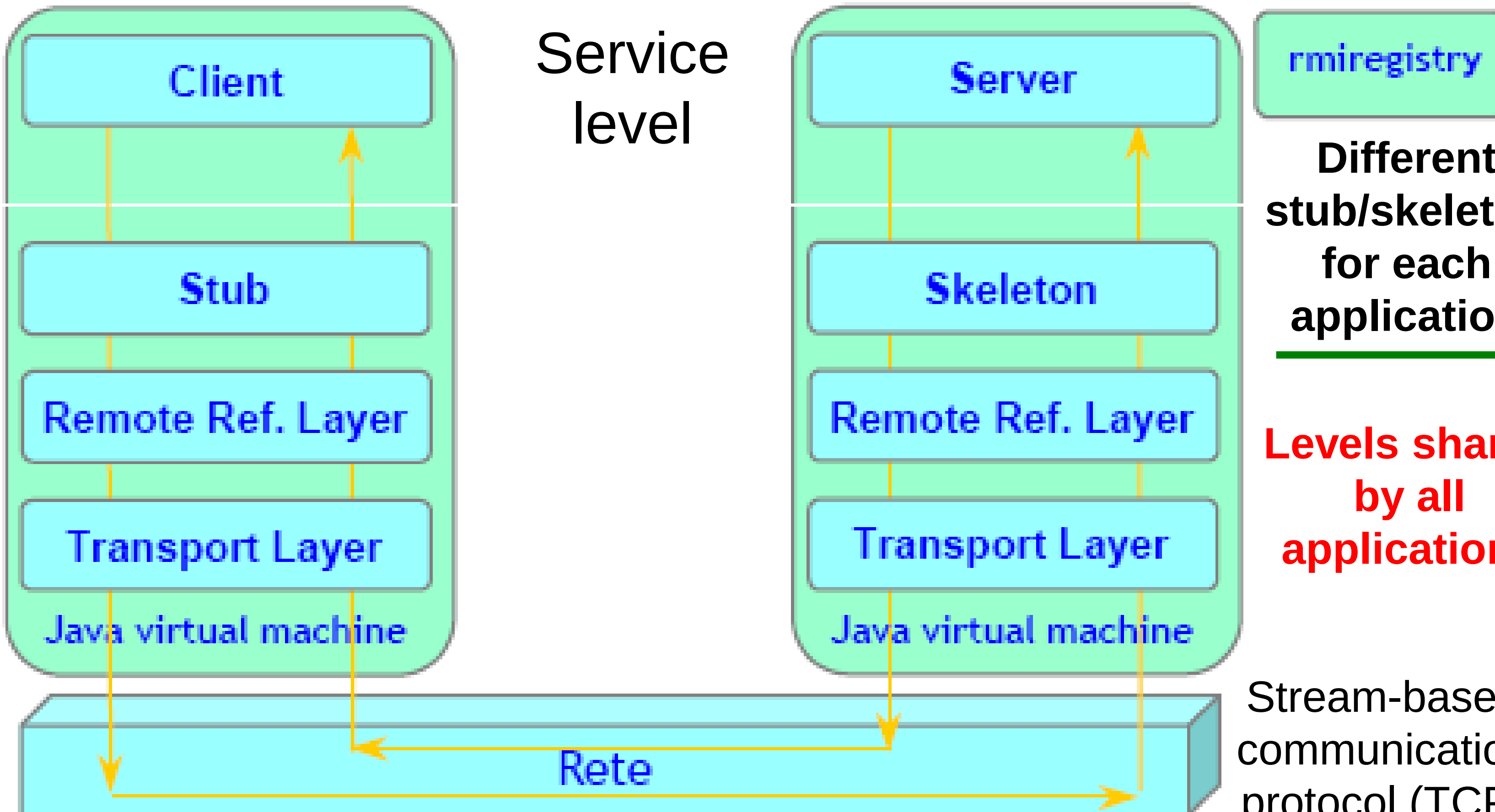
It is possible to instantiate a **new registry** using the following method:

`public static Registry createRegistry(int port)` that is provided by the **`LocateRegistry`** class

This method creates a registry in the **same instance** of the JVM of the calling process

RMI architecture (again)

Only SYNCHRONOUS and BLOCKING interactions



Different stub/skeleton for each application

Levels shared by all applications

Stream-based communication protocol (TCP)

Stub

- It relies on the **Remote Reference Layer** (RRL)
  - Extends **java.rmi.server.RemoteStub**
  - Implements **java.rmi.Remote** and the **remote interface of the server** (e.g. EchoInterface)
  - Contains an instance of the reference to the remote object (**`super.ref`**, class **`java.rmi.server.RemoteRef`**)
- The stub invokes methods, manages the de/serialization, and sends/receives arguments and results

Integer that identifies the method requested

```
…
// call creation
java.rmi.server.RemoteCall remotecall =
super.ref.newCall(this, operations,
   0, 6658547101130801417L);
// parameters serialization
try{
  ObjectOutput objectoutput =
    remotecall.getOutputStream();
  objectoutput.writeObject(message);
}
…
// method invocation, using RRL
super.ref.invoke(remotecall);
…
```

```
// de-serialization of the return value
String message1;
try{
   ObjectInput objectinput =
     remotecall.getInputStream();
   message1 = (String)objectinput.readObject();
}
…
// signal end of call to RRL
finally{
  super.ref.done(remotecall);  //why is it needed?
}
// return result
// to application layer
return message1;
…
```

Skeleton

- **Skeleton** manages de/serialization, sends/receives data relying on RRL, and invokes requested methods (**dispatching**)
- **dispatch** method invoked by RRL, having a input parameters
  - Reference to the server (**`java.rmi.server.Remote`**)
  - Remote call, operation id and interface hash

```
public void dispatch(Remote remote,
  RemoteCall remotecall,
  int opnum, long hash)throws Exception{
  …
  EchoRMIServer echormiserver =
              (EchoRMIServer)remote;
  switch(opnum){
   case 0: // operation 0
     String message;
    try{ // parameters de-serialization
      ObjectInput objectinput =
         remotecall.getInputStream();
      message =
       (String)objectinput.readObject();
    }
    catch(…){…}
    finally{ // free the input stream
       remotecall.releaseInputStream();
    }
```

```
 // method invocation
 String message1 = echormiserver.getEcho(message);
 try{ // serialization of the return value
   ObjectOutput objectoutput =
       remotecall.getResultStream(true);
   objectoutput.writeObject(message1);
 }
 catch(…){…}
 break;
… // manage other methods
default:
  throw new UnmarshalException("invalid ...");
} //switch

} // dispatch
```

Transport level: concurrency

- Specification very open
  - Communication and concurrency are **crucial aspects**
  - Freedom to realize different implementations **but**
- Implementation → **Parallel thread-safe server**

  i.e. application layer must manage concurrency-related aspects → use locks: **`synchronized`**
- Process for each service request

  RMI typically uses Java threads → **built on request**

  This means that there is a thread for each invocation on the remote object running on a JVM
- Given the thread building policy, who does implement it?

  read the skeleton code → **typically does not build threads**

  (which component can manage concurrency and instantiate threads?)

Transport level: communication

- The specification is open
  - It only defines some guidelines about reasonable resource usage
    - If there is already a connection (transport level) between two JVM, try to reuse it.
- Many possibilities
  1. Open a single connection and use it to send one request at time → strong request serialization effects
  2. **Use an already established connection if it is free, else open another connection → uses more resources (connections), but the serialization effects are smoothed**
  3. Use a single connection (transport layer) to send multiple requests, and use demultiplexing to send requests and receive responses

↑ Serialization effects

Deployment issues

- A RMI application needs local access to the **.class** files (for exection and de/serialization)

- **Server needs to access**:
  - Interfaces that define the service → compile time
  - Service implementation → compile time
  - stub and skeleton of the class that implement the service → run time
  - other classes used by the server → compile time and run time
- **Client needs to access**:
  - Interfaces that define the service → compile time
  - stub of the class that implements the service → run time
  - other classes used by the server needed by the client (e.g. return values)→ compile time and run time
  - other classes used by the client → compile time and run time

RMI Registry: the bootstrap problem

How does the system start (**bootstrap**) and how does it find the remote reference?

- Java provides the **Naming** class, that in turn provides static methods for un/binding and to locate the server
- The methods to send requests to the **registry** need the stub of the registry
- How to obtain a stub instance of the registry without using the registry?

  Locally built stub using:

  - Server ***address and port*** contained in the remote object
  - Identifier (local to the server host) of the registry object mandated by the RMI specification → ***fixed constant***

Security and registry

Problem: accessing the registry (that can be found with a port scanner) it is possible to **maliciously redirect the invocations to registered RMI servers**

(e.g. `list()+rebind()`)

Solution:

The methods bind(), rebind() and unbind() can be invoked **only from the host on which the registry is running**

⇨ external nodes **can not** modify the client/server structure

Note: this means that on the machine that hosts servers that invoke registry methods, there must be **at least one registry running**

Bibliography

- Oracle website:
  - http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

- W.Grosso, ***"Java RMI"***, Ed. O'Reilly, 2002

- R. Öberg, ***"Mastering RMI, Developing Enterprise Applications in Java and EJB"***, Ed. Wiley, 2001

- M. Pianciamore, ***"Programmazione Object Oriented in Java: Java Remote Method Invocation"***, 2000

Contacts – Luca Foschini:

- E-mail: luca.foschini@unibo.it
- Home page: www.lia.deis.unibo.it/Staff/LucaFoschini

Classpath and execution

Rmiregistry, server and client must access to the various classes needed for execution. **It is important to take care at the working directory of registry, server andclient**

Assuming that all the .class files are in the current directory ("."), and that we are starting the registry, client, and server from the current directory, we must **add that directory to the classpath**.

Using Linux: edit in your HOME directory the "**`.profile`**" file (create it if it does not exist). The `.profile` file must contain the following lines to add the current directory to the CLASSPATH:

```
CLASSPATH=.:$CLASSPATH
export CLASSPATH
```

The course's FAQ describes the **PATH** environment variable too

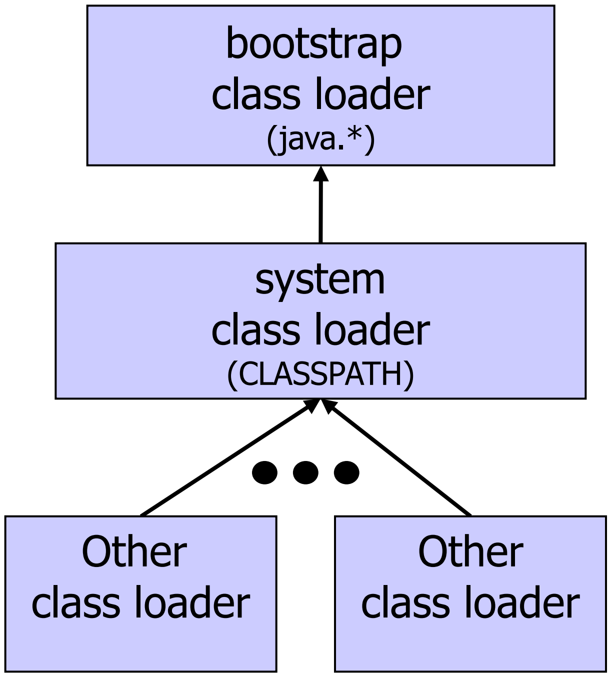
What if we want to start client, server and registry from different directories?

RMI Class loading

Java uses a ***ClassLoader***, namely an **entity that can dynamically load classes** and can refer to and find classes whenever such necessity raises

Classec can be loaded both from the local disk and from the network (e.g. applets) enforcing different **security levels**

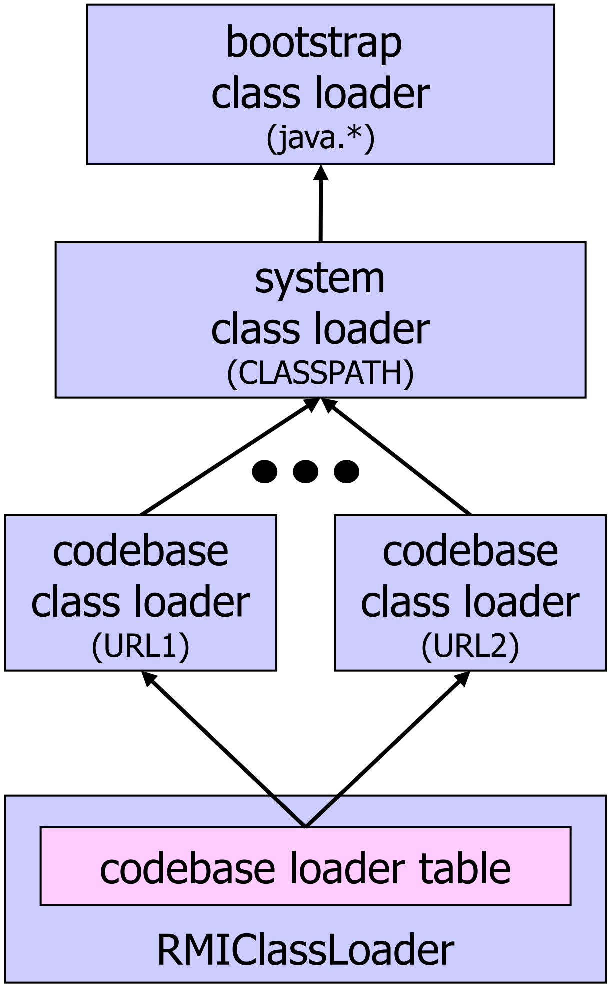
Java allows the definition of a ***hierarchy of different ClassLoaders***, each one responsible for the loading of different classes. They can be even defined by the user



ClassLoaders have separate domains and can not interfere with each other. They can even be inconsistent.

Security Enforcing managed by the Security Manager

RMI Class loading



Java defines a **hierarchy of different ClassLoaders**, each one responsible for a different set of classes. They can be specialized by the user-

Classloader: resolves class names used in class definitions (code – bytecode)

Java RMI **Codebase classloader**: responsible for the loading of classes that can be reached using a standard URL (codebase) → **even remote**

RMIClassLoader **IS NOT** a real ClassLoader, instead it is a RMI support component that executes two crucial tasks:

- Extracts the codebase field from the reference of the remote object
- Uses the codebase classloader to load the needed classes from the remote location.

RMI Security

Every JVM, can have a **Security Manager**, a component that checks the correct execution of each operation and makes sure that there are no security breaches

- Both the client and server must be started specifying the file containing the **requested privileges** (**policy file**) interrogated by the security manager (for dynamic security control)
- To safely execute code, Java RMI requires a **RMISecurityManager**
  - RMISecurityManager **checks the accesses** (specified in the **policy file**) to system resources and blocks **unauthorized accesses**
  - The security manager is created **within the RMI application** (both **client side**, and **server side**), if there isn't already one

```
if (System.getSecurityManager() == null)
    {System.setSecurityManager(new RMISecurityManager()); }
```

- Examples:
  - Client: `java -Djava.security.policy=echo.policy EchoRMIClient`
  - Server: `java -Djava.security.policy=echo.policy EchoRMIServer`

Policy file

- Policy file structure:

```
grant {
  permission java.net.SocketPermission "*:1024-65535", "connect, accept";
  permission java.net.SocketPermission "*:80", "connect";
  permission java.io.File Permission "c:\\home\\RMIdir\\-", "read";
};
```

- The first permission allows client and server **to establish connections for remote interaction** (non-privileged ports)
- The second permission allows to **get bytecode** from a **http server listening on port**
- The third permission allows to **get bytecode** from the **root of the allowed directory**.

Dynamic code downloading

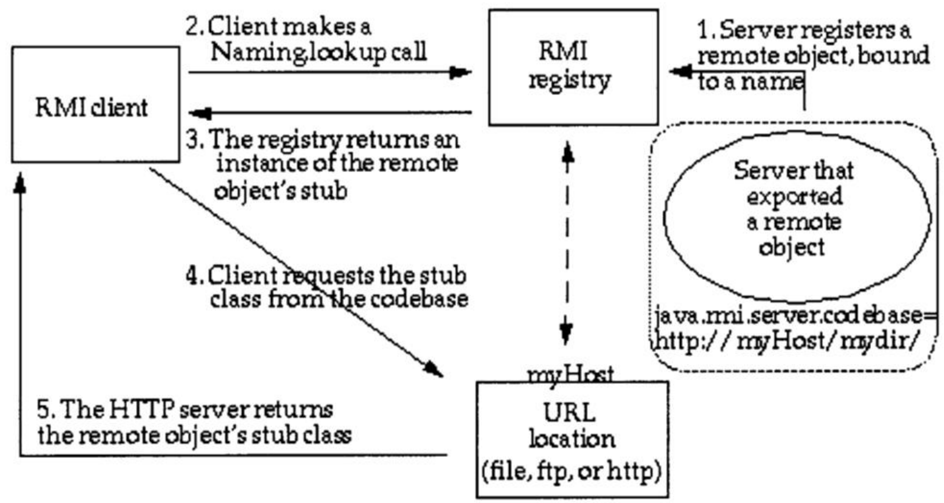
It may be necessary to dynamically load code (stub or classes)

- Steps:
  1. **Find the code** (local or remote)
  2. **Download it** (if it is remote)
  3. **Safely execute the code**

- The information about code repositories are stored at server side and are **sent to the client when needed**:
  - **RMI Server** started specifying the option
    java.rmi.server.codebase the URL where necessary classes are stored
  - The URL can be
    - A HTTP server (**http://**)
    - A FTP server (**ftp://**)
    - A local directory (**file://**)
- codebase is a server property that is ***stored in the RemoteRef*** published on the registry (i.e. contained in the stub instance)
- Classes are looked for first in the local CLASSPATH, then in the **codebase**

Using codebase

- **codebase** (stored in a RemoteRef) is used by the client to download the classes related to the **server** (interfaces, stub, objects sent as return value)
  - **NOTE**: difference between stub instance and class



- What happens when there is a parameter passing by value (from the client to the server) of an object that is an **instance** of a class **unknown to the server**?
  - The **server** uses the codebase to download the classes related to the client (objects passed as calling parameters)