Remote Method Invocation (RMI)

RMI

- **Procedural** programming → RPC
- **Object** programming → distributed objects

- RMI is the distributed object technology proposed by SUN
- Enables to invoke methods on objects located in **different** Java Virtual Machines

The local model

- OOP relies on:
  - **Class**: abstract data type
  - **Object**: instance of a class
- Objects encapsulate **data** and **code**
- Objects are composed of a **status** (hidden) and an **interface** (public)
- The interaction happens by means of **method invocation**
  - In a **client-server** style
- In Java, all the objects of an application reside in the **same** JVM
- **Multithreading** is possible
  - JVM → process
  - Objects of Thread class → thread

The remote model

- In the local model, even if two JVMs execute on the same host, the objects of one JVM **cannot** invoke the methods of the objects of the other JVM
- In the remote model, an object **can invoke** a method of an object executing in **another** JVM
- **High-level** interaction
  - **Masks** lower-level socket interaction

Granularity

- The granularity specifies which is the **minimum remote entity** that can be addressed
- As **entities**, an **object** can reference remote objects
- As **execution**, an object can invoke the **methods** made available by the remote objects

Remote objects and distributed applications

- Remote object
  - Makes available methods that can be invoked from **remote**
  - Implements a remote (Java) **interface** that declares the remote methods
    - Sub interface of `Remote`
- Distributed application
  - Composed of objects that reside on **different** hosts
  - Remote objects as **servers**
  - **Client** objects

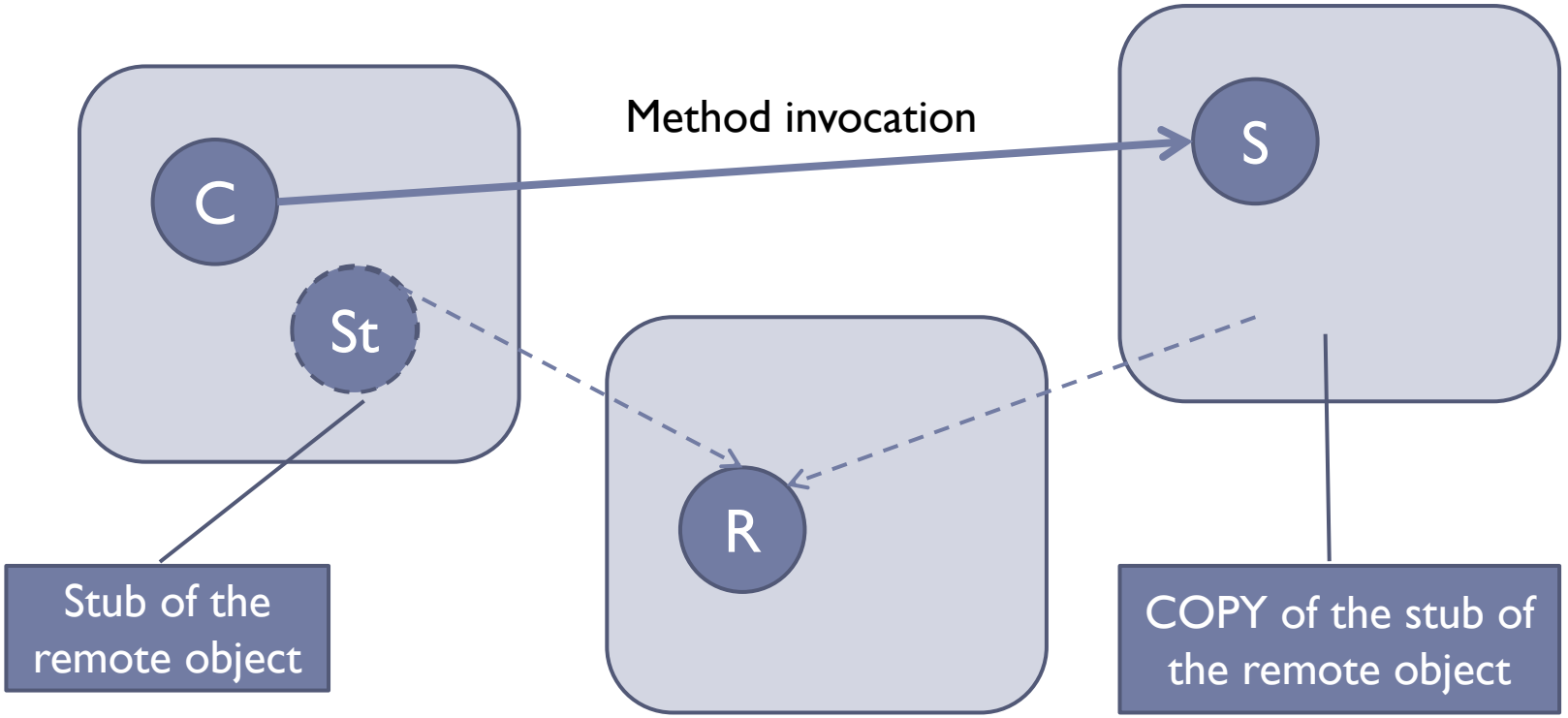
Parameter passing

- In the **local** case, a **copy** is passed as parameter:
  - Of a **primitive** type
    - Modifications of the callee are not seen by the caller
  - Of an **object** reference
    - Actually, a passing by reference → the caller sees the modifications of the callee
- In the **remote** case, even passing a reference to an object causes a passing by **value**
  - The referred object is **serialized** and sent to the server object
  - **Deep** copy
- Note that **not all** objects can be serialized
  - For instance, **files** and **sockets** cannot be passed as parameters

Parameter passing (2)

- A different case is when a reference to a **remote** object is passed as parameter
- In this case, a **copy of the stub** (not of the object) is passed as parameter
- Both client and server have a copy of the stub, which refers to the **same** remote object
- The modifications made by the server on the remote object **are seen** by the client

Parameter passing (3)



Parameter passing (4)

- Summary
  - **Primitive** types and references to **local** objects are passed by **value**
  - References to **remote** objects are passed by **reference**

- **Different semantics** from local case (where local objects are passed by reference)
- **No** syntactic difference
  - The programmer **must** know the type of the parameter

Static vs. dynamic invocation

- RMI adopts a **static** invocation mechanism
- **Dot notation**
- **Same** invocation as methods of local objects
- Local and remote calls are **uniform**

- A **dynamic** invocation can be realized by means of **reflection**

Reliability

- RMI provides a specific class for **exceptions** deriving from the distributed nature of the technology
- Class `RemoteException`
- **Every** remote method must declare that it can throw the `RemoteException`
- Each call to a remote method must happen inside a **try-catch** construct
- **Less** transparency
  - The client must know which methods are remote
  - The client must manage the network and server failures
- Perhaps the best **trade-off** between simplicity and safety

Object distribution

- RMI does **not** provide any tool for the deployment of the objects
- The system administrator must **manually** deploy the different objects on the hosts of the system
- RMI supports the **dynamic** retrieving of classes (bytecode files)
  - By means of **web server** (see later)

Available services

- The first version of RMI did **not** provide any facility service
- Only the **naming** service was provided
  - RMI registry, described later
- In the second version (bound to Java 2) an **activation** service is provided
  - A server object can be suspended
  - And resumed when a client request arrives

Homogeneous vs. heterogeneous technology

- RMI allows only **one language**: Java
- This makes RMI a **homogeneous** technology
- Pros
  - Marshalling and unmarshalling are **easier**
  - **More efficient**
  - **Easier** to program
- Cons
  - All components **must** be written in Java
  - **No reuse** of components written in different languages

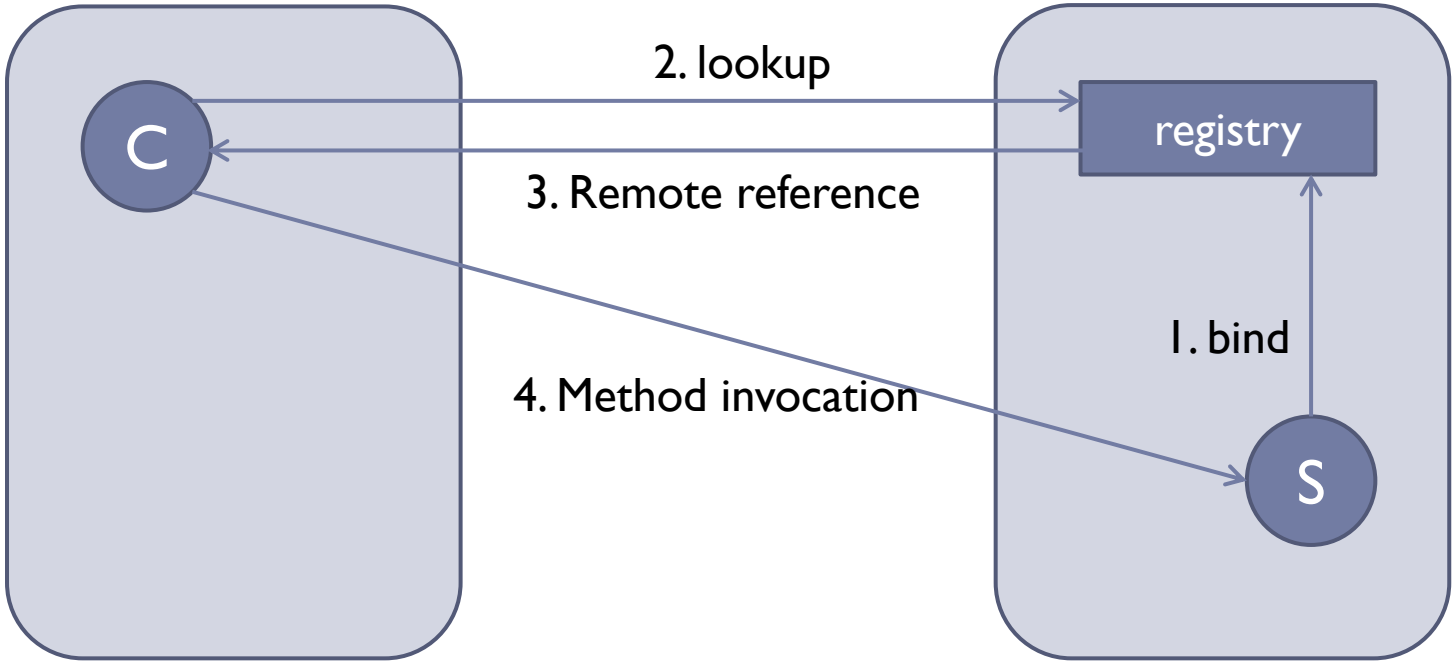
RMI architecture

- Besides the client and the server, the RMI architecture involves three more **components**
- Registry
  - Naming service
- Stub
  - Proxy to grant transparency
- Web server
  - To retrieve the class bytecode

Naming service: rmiregistry

- Rmiregistry is a **name server**
  - Keeps the **associations** between **names** and **objects**
  - It is a daemon that listens to on port **1099** (default)
- It works as follows
  1. A remote object is **registered** by a registry (**bind**)
  2. The client **queries** the registry providing a name (**lookup**)
  3. The registry returns the **reference** of the associated object
  4. From now on, the client can call the object's methods in the normal way (dot notation)

Naming service: rmiregistry (2)



Naming service: rmiregistry (3)

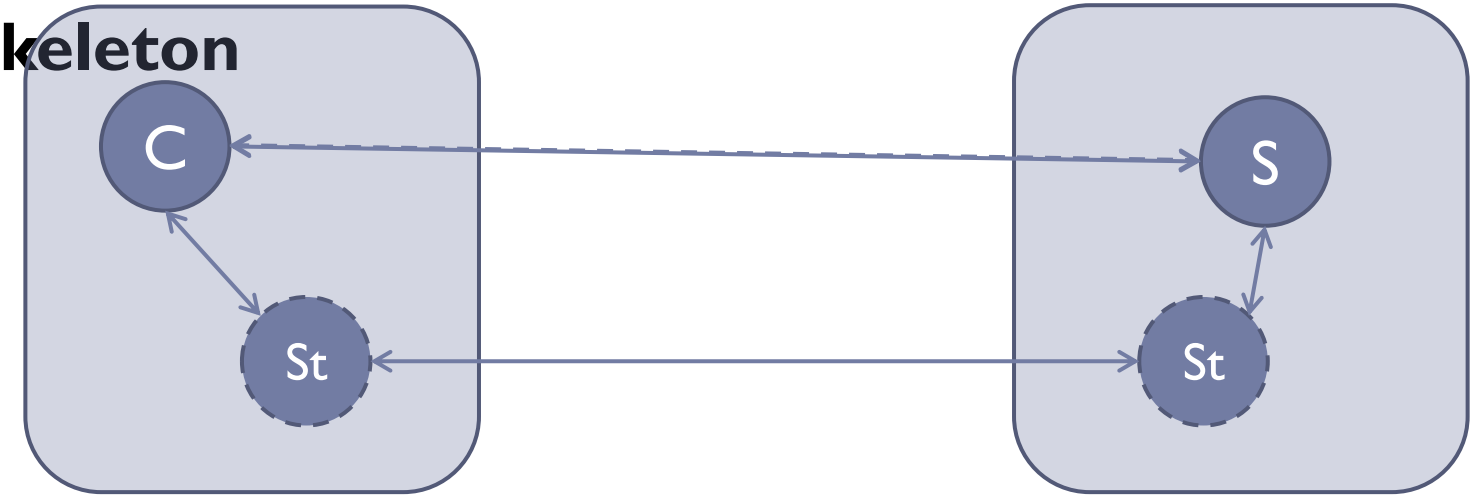
- Objects can register only on the **local** rmi registry
- Pros
  - Simpler
  - More secure
- Cons
  - One registry per host
  - Less flexible

Naming service: `Naming` class

- The class `java.rmi.Naming` provides methods for managing the **naming** service
  - **Static** methods
- Two services
  - **Registration** of a server object
    - `bind()`
    - `rebind()`
  - **Resolution** of names
    - `lookup()`
- The `lookup()` method returns a **reference** of type `Remote`
  - A **cast** is needed to call the interesting methods
- Names are strings in the form **`//host/service`**

Stub (1)

- **Stubs** are **proxies** that mask details about:
  - Parameters (marshalling and unmarshalling)
  - Communication
- The **stub** "pretends" to be the server for the client and to be the client for the server
- In the first versions of RMI the server stub was called **skeleton**



Stub (2)

- Stubs are **Java objects**
- Dynamically created

- Until some Java versions ago, the code of the stub (and skeleton) classes was created by a **special compiler**: rmic
- The rmic compiler also **compiles** the classes' sources
  - And **deletes** the stub and skeleton source files
  - Use **-keep** option to keep the sources

Web server

- The client and the server **may not have** the needed class files
- For instance, the stub and skeleton classes are not needed for compiling the client, but are needed at **runtime**
- As another example, the **return value** can be an object instance of a subclass of the return type

```
Object MyMethod() {
   return new MyStrangeClass(); }
```

- So, a **runtime mechanism** that enables to **load classes** from remote is needed

Web server (2)

- RMI exploits **web servers** to retrieve the code of the classes
- **Wide-spread** servers
- **Tested** protocol and architecture
- Port 80 is usually **open** in firewalls

Implementation of a RMI application

- The phases of the development of an RMI application are:
  - Definition of the **code** of the components
  - **Distribution** of the classes
  - **Execution** of the server and the client

Definition of the remote interface

- A remote interface is a **Java interface** that declares the remote methods
  - Only such methods can be invoked from remote
- It **extends** the `java.rmi.Remote` interface
- Each method must **declare** that it can throw the `java.rmi.RemoteException` (throws clause)

Definition of the server object

- The server object is implemented by a **Java class**
- **Subclass** of `UnicastRemoteObject`, in the `java.rmi.server` package
- **Implements** the remote interface previously defined
- An instance of the server object must be **registered** in the registry
  - By means of the `bind()` or `rebind()` method

Definition of the client object

- The client object is implemented by a **Java class**
- **No** specific requirements
- Retrieve the **reference** to a remote object
  - By means of the `lookup()` method
- Exploits the remote reference as a **local** reference
- The remote methods can throw a `RemoteException`
  - **Try-catch** is needed

Distribution of the classes

- To be **compiled**, the client and the server need the source/bytecode of the **remote interface**
- To **run**, the client and the server need the bytecode of:
  - The **class** that implements the client or the server respectively
  - The **stub** and **skeleton** classes
- If the classes are not available locally, their location can be specified by means of the **codebase**
  - A set of URLs (eg: http://… ftp://… file://)

Distribution of the classes - codebase

- The codebase can be specified as a co**mmand line argument** of the interpreter
  - `-Djava.rmi.server.codebase=…`
- Otherwise, it can be specified in the **code** of the program, by means of a **system property**
  - `System.setProperty("java.rmi.server.codebase","…");`

Execution of the server and the client

- The rmiregistry must be **running**
- The first step is to run the **server**
  - It registers itself (or it is registered) in the rmiregistry
  - It is suspended waiting for client requests
- Then the **client** can run
  - Retrieves the reference to the remote object
  - Calls its methods (the ones declared in the remote interface)

Example

- Let us develop a server that returns a **greeting** message
- Steps:
  - Definition of the remote interface
  - Implementation of the server
  - Implementation of the client

Example – the interface

- The remote interface must:
  - **Extend** the `java.rmi.Remote` interface
  - **Declare** throws `RemoteException` for each method

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

Example – the server

- The **server** class must **import** the appropriate packages

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

- The server class must **extend** `UnicastRemoteObject` and **implement** the `Hello` remote interface

```
public class HelloImpl extends
UnicastRemoteObject implements Hello
{
```

- The server keeps its **name** in a private variable

```
  private String name;
```

Example – the server (2)

- The server **constructor** calls the superclass constructor and assigns the name to the name variable
- It can throw RemoteException

```
 public HelloImpl(String s) throws
RemoteException
   {
      super();
      name = s;
   }
```

Example – the server (3)

- The server must define the **code** of the `sayHello` method
- The method can throw `RemoteException`

```
public String sayHello() throws
RemoteException
   {
     return "Hello World!";
   }
```

Example – the server (4)

- Finally, the server defines a **main** method that starts the execution
- First, it sets a **security manager**

```
public static void main(String args[])
   {
     if (System.getSecurityManager() == null) {
          System.setSecurityManager(new
SecurityManager());
     // RMISecurityManager is deprecated from 8
     }
```

- Then, it creates an **instance** of the server object

```
try {
          HelloImpl obj = new
HelloImpl("HelloServer");
```

Example – the server (5)

- The server object is **registered** in the local rmiregistry
  - By means of the (static) method `rebinds()` of the `Naming` class
- The associated name is **`HelloServer`**

```
Naming.rebind("HelloServer", obj);
```

- Some **prints** to notify the user

```
System.out.println();
System.out.println("HelloImpl:
HelloServer bound in registry");
System.out.println();
```

Example – the server (6)

- All the code is inside a **try-catch** because the method calls can throw an exception

```
} catch (Exception e) {
    System.out.println("HelloImpl err: "
+ e.getMessage());
    e.printStackTrace();
   }
  }
}
```

Example – the client

- The **client** class must **import** the appropriate packages

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

- The client is implemented by the **`HelloApp`** class with no specific features
- It defines a **`main`** method

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

Example – the client (2)

- The client defines **two string variables**
  - One for the **message** to be received
  - One for the **name** of the service

```
String message = "";
String Server = "//magroup1/HelloServer";
```

- The name of the service is the **same** as the one exploited by the server to register the server object
- The name of the service is **independent** of the name of the class
- The name of the server corresponds to an **object**, not to a method

Example – the client (3)

- The client must retrieve the **reference** to the remote object
  - By means of the (static) method `lookup()` of the `Naming` class

```
try {
        Hello obj =
(Hello)Naming.lookup(Server);
```

- The `obj` variable is of type remote interface
  - We do not need to know the implementation
- A **cast** is needed since lookup returns a reference to `Remote`

Example – the client (4)

- Now, the client can call the `sayHello` method on the `obj` reference, as it was a reference to a **local** object
- The RMI system is in charge of translating this call into a **remote** call

```
message = obj.sayHello();
```

- Then, we can **print** the returned message

```
System.out.println("Message received: " +
message);
```

- All the code is inside a **try-catch** because the method calls can throw an exception

```
} catch (Exception e) {
        System.out.println("HelloApp exception: "+ e.getMessage());
        e.printStackTrace();
    }  } }
```

Example – compiling and running

- Compiling of the classes and interface

```
javac Hello.java HelloImpl.java
HelloApp.java
```

- Execution of the rmiregistry on the server host

```
rmiregistry
```

- Server execution

```
java HelloImpl
```

- Client execution

```
java HelloApp
```

Security issues

- From version 2, Java provides a **more flexible** security management
  - Java 1 → **sandbox** for applets, no restriction for applications
  - Java 2 → chance of defining **granular permissions** for both applets and applications
- Policy file

```
$JAVA_HOME/jre/lib/security/java.policy
```

- If our server does not have the permission of using sockets, it throws an **exception**

```
java.security.AccessControlException:
access denied (java.net.SocketPermission
127.0.0.1:1099 connect,resolve)
```

Security issues (2)

- The **socket permission** must be granted
- In the **system** policy file (if allowed)

```
grant {
...
permission java.net.SocketPermission "*:1024-",
"accept, resolve, connect";
...
};
```

  - The wildcard "*" means "all the hosts"
- In a **secondary** policy file, only the needed permission
  - For instance ~/.local.policy
- In the latter case, the **location** of the policy file must be specified

```
java -Djava.security.policy=~/.local.policy
HelloImpl
```

Other issues

- Check if the compiler (`javac`), the interpreter (`java`), the registry (`rmiregistry`) and the RMI compiler (`rmic`, if used) belong to the same version
- Check if the registry can access the classes
  - Start it in the classes' directory

Considerations

- Transparency
  - When invoking the remote method we have:
    - **Uniformity**: the remote call is equal to a local call
    - **Transparency**: the RMI systems hides the network details
  - Limitations to transparency
    - The client must know the **location** (host) of the server
    - The client must know which methods are **remote** (to wrap them in a try-catch)
- Object transfer
  - Objects are **transferred** (by **copy**) from one JVM to another
    - When they are passed as **parameters**
    - When they are **returned** by a method

Considerations

- Code mobility
  - An object is composed of **data** and **code**
  - When an object is transferred, its **code is transferred** as well
  - The object **class** contains the code of the methods
  - If the destination host does not have the class (byte)code, the **ClassLoader** of the JVM is in charge of searching for it