Programação em Sistemas Distribuídos

MEI-MI-MSI

2018/19

Java Remote Method Invocation (RMI)

Prof. António Casimiro

What is RMI and Java RMI ?

Ciências ULisboa

- Remote method invocation (RMI) similar to RPC but extended into the world of distributed objects
- RMI:
  - calling object can invoke a method in a potentially remote object
  - power of O_O: objects, classes and inheritance, tools.
  - unique object references which can be passed as parameters
  - supports creation of applications with distributed objects
- Java RMI
  - Java-to-Java only
  - Client-Server protocol
  - High-level API (O_O)
  - Transparent
  - Lightweight

Related technologies

Ciências ULisboa

- **RPC** (“Remote Procedure Calls”)
  - Developed by Sun
  - Platform-specific, lower level (RPC)
- **CORBA** (“Common Object Request Broker Architecture”)
  - Developed by Object Management Group (OMG)
  - Access to Java and non-Java objects
- **DCOM** (“Distributed Component Object Model”)
  - Developed by Microsoft
  - Access to Win32 objects
- **LDAP** (“Lightweight Directory Access Protocol”)
  - Lookup for network resources

Java RMI

- Part I: RMI concepts
- Part II: RMI API and usage

RMI

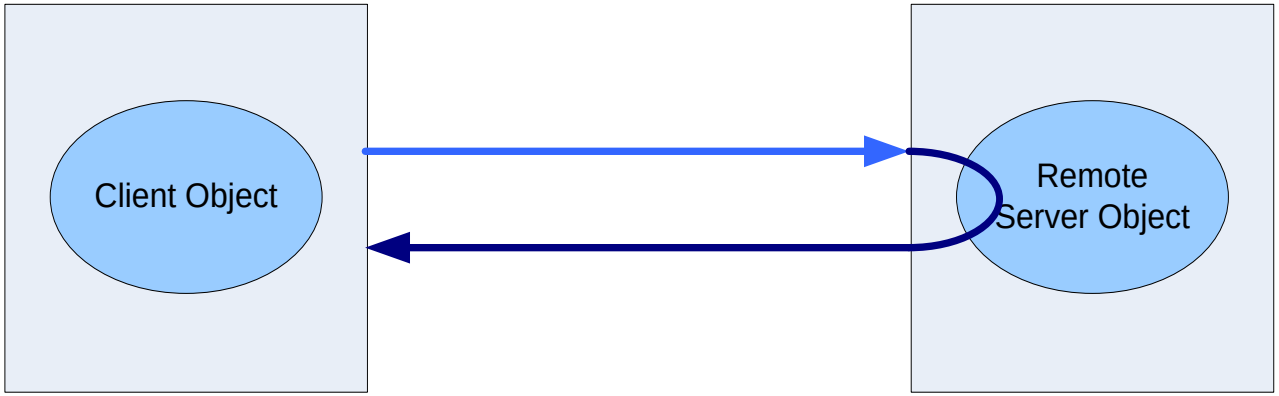
Ciências ULisboa

- **Client** – the process that invokes a method on a remote object
- **Server** – the process that contains the remote object
- **Object Registry** – the name server that relates names with objects
  - Objects are registered with the Object Registry using a unique name
  - The Object Registry is used to gain access to remote objects through their names

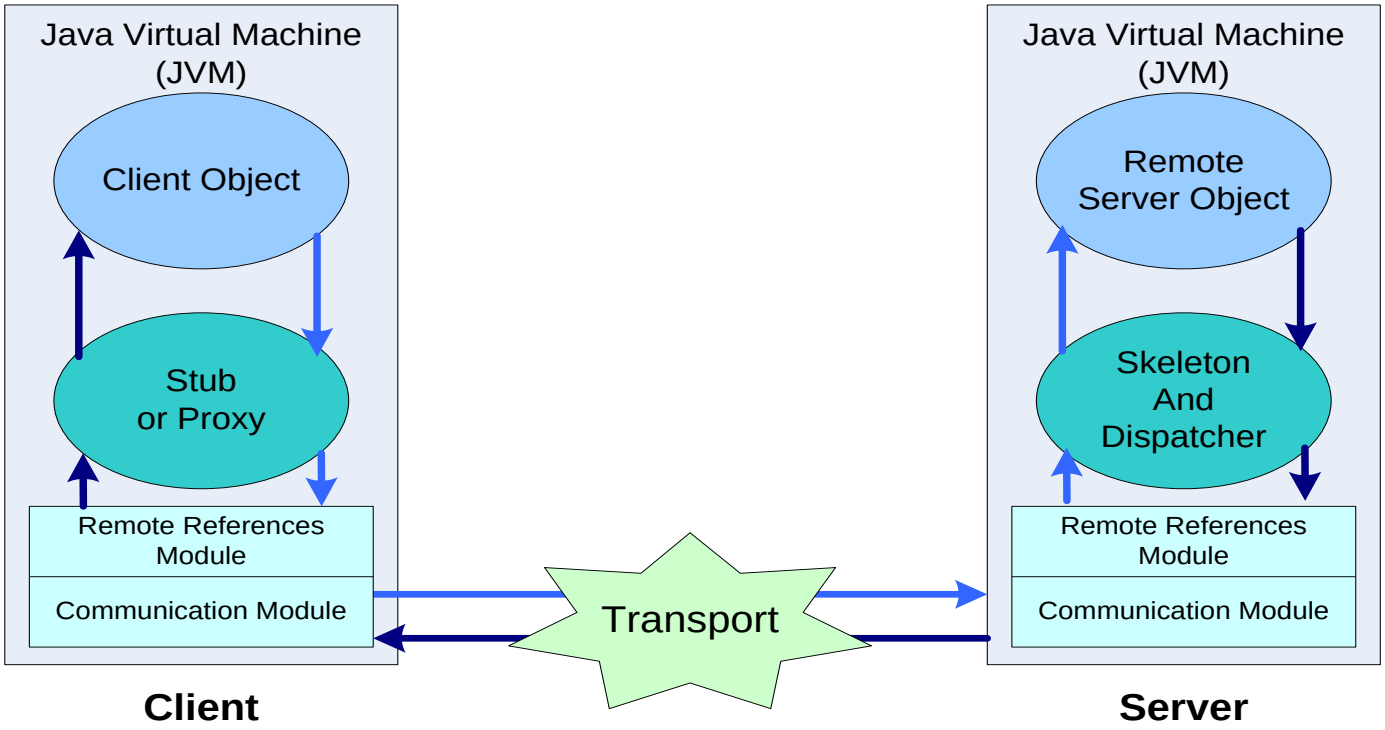
RECALL: Remote Method Invocation
(Remote vs. local transparency)

Ciências ULisboa

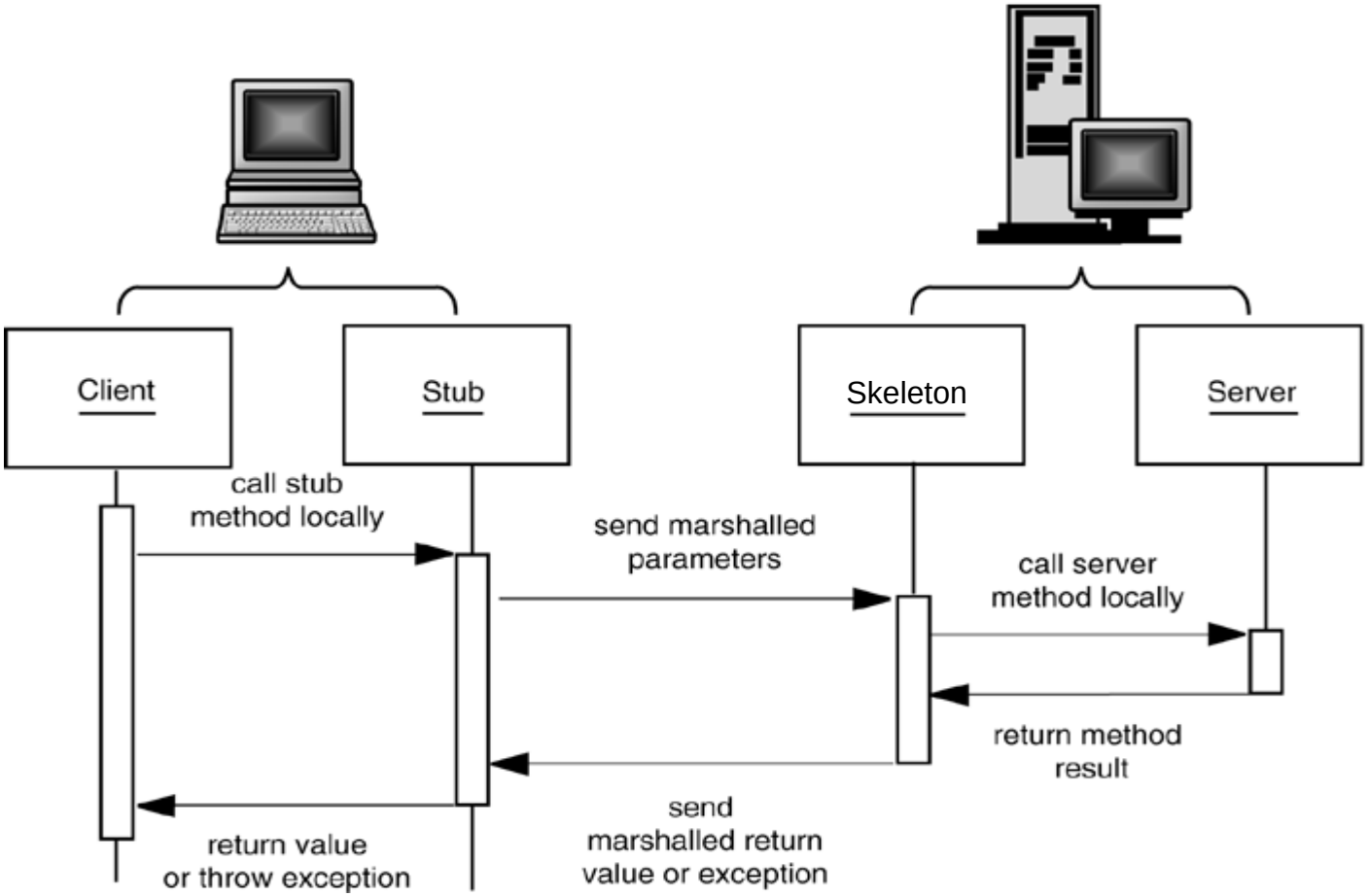
- Client accesses remote object as if it were local
- Client request to invoke a method of an object is sent in a message to the remote server managing the object
- The call is carried out by executing a method of the object at the server
- The result is returned to the client in another message



Java RMI layers



Stubs and Skeletons (1)



Stubs and Skeletons (2)

Ciências ULisboa

- **Stub or Adapter or Proxy**
  - Located on the client
  - Represents the remote object – is a proxy for the remote object
  - Transparently executes the functions needed for preparing the remote method invocation
- **Skeleton or Receptor**
  - Located on the server
  - Receives requests from the stub, interacts with the remote object and delivers the response to the stub. Connects the stub and the remote object.

Stubs and Skeletons in action

Ciências ULisboa

- Stub
  - Initiates remote method invocations
  - Performs **marshalling**: transforms call arguments into a suitable network transmission format
  - Informs the remote references module that the method must be called in the server
  - Performs **unmarshalling**: transforms the return value from a network format to a local representation
  - Is informed by the remote references module that the call has been completed with the server reply
- Skeleton
  - Is informed by the remote references module of a new call
  - Performs unmarshalling of received arguments
  - Calls the remote object
  - Performs marshalling of the result to be sent to the client
  - Passes the response to be returned to the remote references module

Remote References Module

Ciências ULisboa

- Remote References layer
  - Creates and manages **references between clients and remote objects**
  - Establishes the connection and manages **communication between the Stub and the Skeleton**

- Examples of invocation protocols that might be implemented in this layer:
  - Point-to-point invocations with unicast
  - Group invocations to replicated objects
  - Support for persistent references to remote objects
  - Replication and recovery (re-connection) strategies

Communication Module

Ciências ULisboa

- The transport layer is responsible for the **low-level connections between the two Java VMs** and also for the respective connection management
- Possible protocols:
  - By default uses Java Remote Message Protocol (JRMP), which works **over TCP/IP**
  - Can also use CORBA Internet Inter-Orb Protocol (IIOP), dubbed RMI-IIOP, which allows the interoperabilityty between Java RMI and CORBA implementations
  - If these protocols cannot be used, then it is possible to use an HTTP tunnel, a strategy to go through firewalls

Object Registry

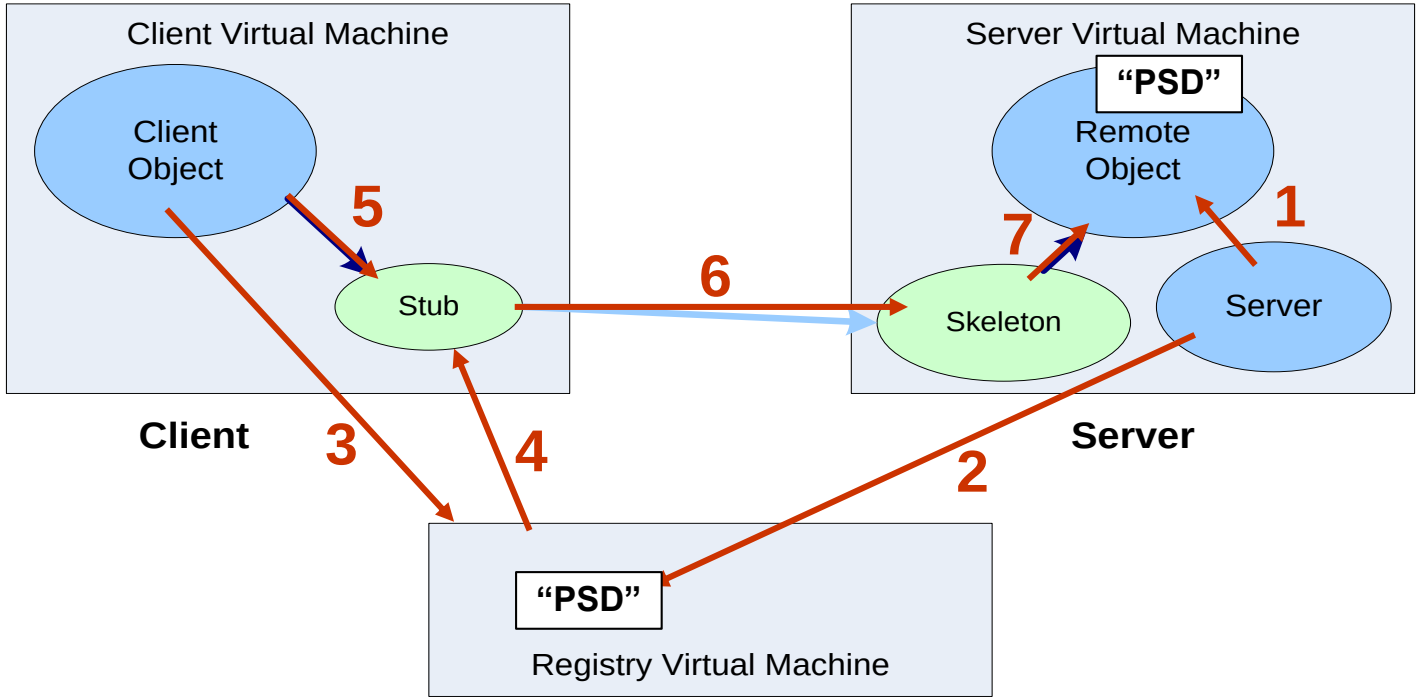
Ciências ULisboa

- A client can use the Java Naming and Directory Interface (JNDI) or the **RMI Registry** (simpler) to locate remote objects
- Object Registry
  - The Object Registry is also a remote object
  - It registers remote objects through a name
  - Clients obtain a remote reference for a remote object

Java RMI in action

Ciências ULisboa

5. Client calls the Stub method
6. Stub communicates with the Skeleton
7. Skeleton calls the remote object method



3. Client requests object to the Registry
4. Registry returns the remote reference
1. Server creates the remote object
2. Server registers the remote object

Threads

- A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a **separate thread**
- Some calls, from the same client VM, will be executed in the same thread and others in separate threads
- Calls originating from different client VMs will be executed in separate threads
- The RMI runtime makes no guarantees with respect to mapping remote object calls to threads (except those from the same client VMs)

Parameter passing

Ciências ULisboa

- In RMI there are three types of arguments or return values:
  - **Primitive types**: passed by copy
  - **Non-remote objects**: since the references in a VM are meaningless in another VM, the objects are serialized and passed by copy. Changes in the object properties on the remote VM do not reflect on the local VM object
  - **Remote objects**: the stub for that remote object is passed instead of the object. A remote object passed as a parameter can only implement `remote` interfaces

Object models in Java

Ciências ULisboa

- Normal *vs* distributed models: **similarities**
  - A reference to a remote object can be passed as a parameter or retuned as a result of any call
  - A remote object can be transformed (cast) in any remote interface supported by some implementation
  - The operator `instanceof` may be used to learn the remote interfaces supported by a remote object
- Normal *vs* distributed models: **differences**
  - Clients only interact with remote interfaces and not with the implementation classes of the remote objects
  - Primitive types and non-remote object are passed by copy and not by reference
  - During remote method invocations, clients must deal with additional exceptions and failure modes

Dynamic class loading

Ciências ULisboa

- In RMI, dynamic class loading works similarly to applet loading in Web browsers and thus requires a Security Manager: **`RMISecurityManager`**
- Advantages
  - A unique RMI characteristic is its capacity to download the bytecode of an object's' class if the class is not defined in the receiver's VM
  - The type and behaviour of an object, which was previously available in a single VM, can be transmitted to another VM, which might even be a remote one
  - Java RMI passes objects by their actual classes, so the behaviour of those objects is not changed when they are sent to another VM
  - Allows new types to be introduced into a remote Java VM, thus dynamically extending the behaviour of an application

Object activation

Ciências ULisboa

- **Object activation** is a mechanism for providing persistent references to objects and managing the execution of object implementations
- In RMI, activation allows objects to **begin execution on an as-needed basis**, without continuously using machine resources
- When an **`activatable` remote object** is accessed (via a method invocation), if that remote object is not currently executing, the system initiates the object's execution inside an appropriate JVM
- Object activation is provided by the **Remote Reference layer**
- The client stub is not connected to the active remote object, but to the **activation system**
- The activation system is in a **RMI daemon `rmid`**, with the ability to start new instances of remote objects whenever needed
- The **registration process** of an **`activatable`** remote object is substantially different, but this is transparent to the client

Distributed garbage collection

Ciências ULisboa

- Distributed garbage collection (DGC) allows collecting remote server objects that are no longer referenced by any clients
- The RMI runtime keeps **client-side reference counts** and uses **server-side validity periods** for references, to perform DGC
  - A reference count indicates the number of references to the remote object on the client side. When it gets to 0, the remote object server is informed
  - On the server side, when a reference is not used during the defined validity period (10 min. by default) it is marked for collection and the client is notified

Ciências
ULisboa

Design elements

“As is”, culled from several sources on manuals and the Internet, with aim of giving design and programming examples

Java RMI

Ciências ULisboa

- Part I: RMI concepts
- Part II: RMI API and usage

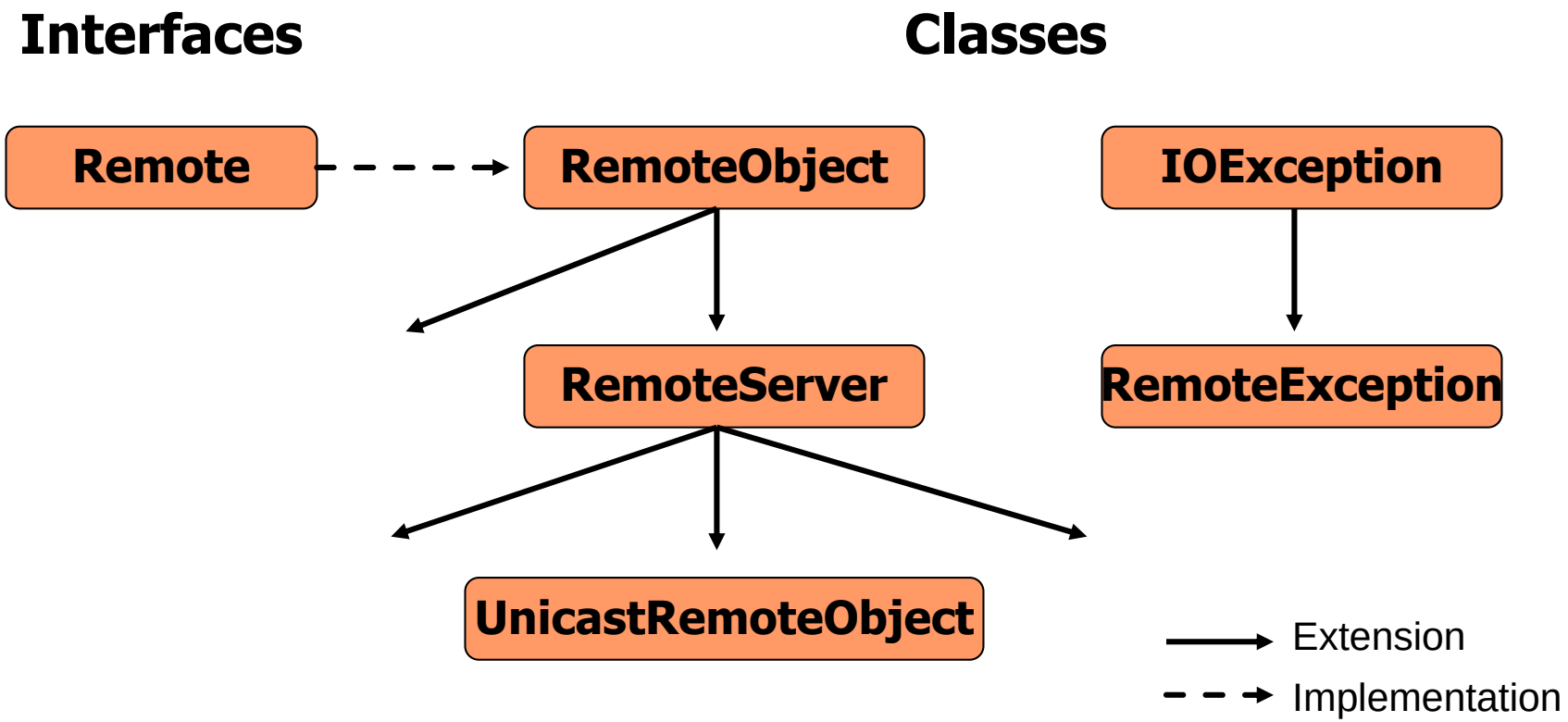
Java RMI API

Ciências ULisboa

- **Main packages**
  - `java.rmi` – Is the main RMI package; it contains the principal objects used in RMI clients and servers
  - `java.rmi.activation` – Supports dynamic activation of remote objects
  - `java.rmi.dgc` – contains an interface and two classes that support DGC in RMI. DGC is normally handled automatically by the RMI system, so this package is typically not necessary
  - `java.rmi.registry` – Provides an interface and an implementation for the various elements of the RMI object naming registry
  - `java.rmi.server` – Contains the classes that develop server implementations of remote objects

RMI interface e classes

- **Fundamental RMI classes**



Remote interfaces

Ciências ULisboa

- Remote interfaces
  - They extend the `java.rmi.Remote` interface
  - They declare the exported methods – those that can be called through remote invocations
  - The remote object implements this interface
  - It can be seen as a proxy for the remote object
  - And as a contract between the client and the server

Creating remote objects

Ciências ULisboa

- Define a **remote interface**
  - Extends `java.rmi.Remote`
  - Example:

```
(…)
public interface Hello extends Remote {
(…)
```

- Define the **class** that implements the remote interface
  - Extends `java.rmi.UnicastRemoteObject`
  - Example:

```
(…)
public class HelloImpl extends UnicastRemoteObject
  implements Hello {
(…)
```

Binding & looking for remote objects

Ciências ULisboa

- Provided by **`java.rmi.registry`** package
  - Bind remote objects with:
    - `static void Registry.bind(String, Remote)`

    Binds the name to the specified remote object
    - `static void Registry.rebind(String, Remote)`

    Rebind the name to a new object; replaces any existing binding
    - `static void Registry.unbind(String)`

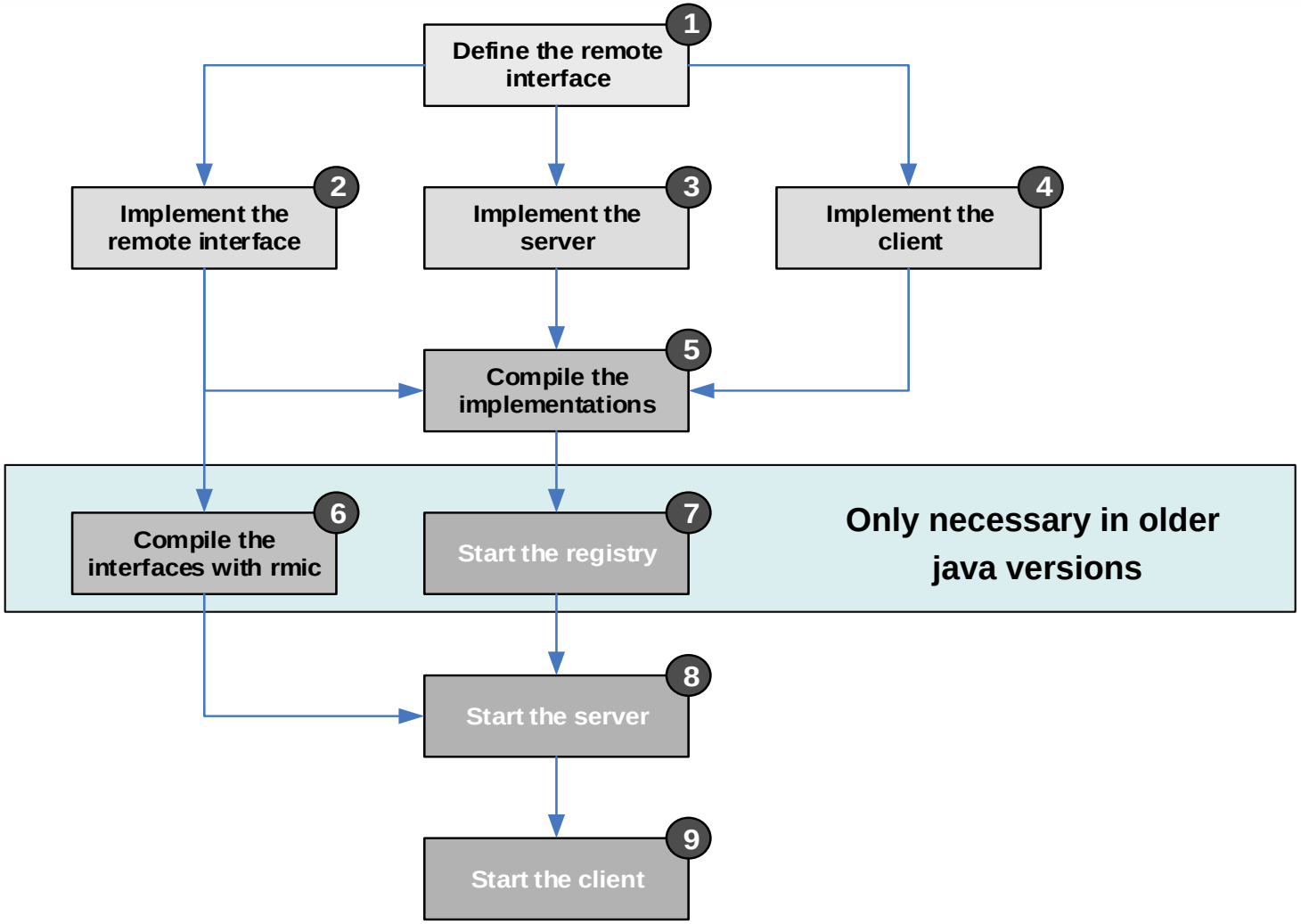
    Removes the binding for the specified name in the registry
  - Lookup remote objects with:
    - `static String[] Registry.list()`

    Returns an array of the names bound in the registry
    - `static Remote Registry.lookup(String)`

    Returns the remote reference bound to the specified name

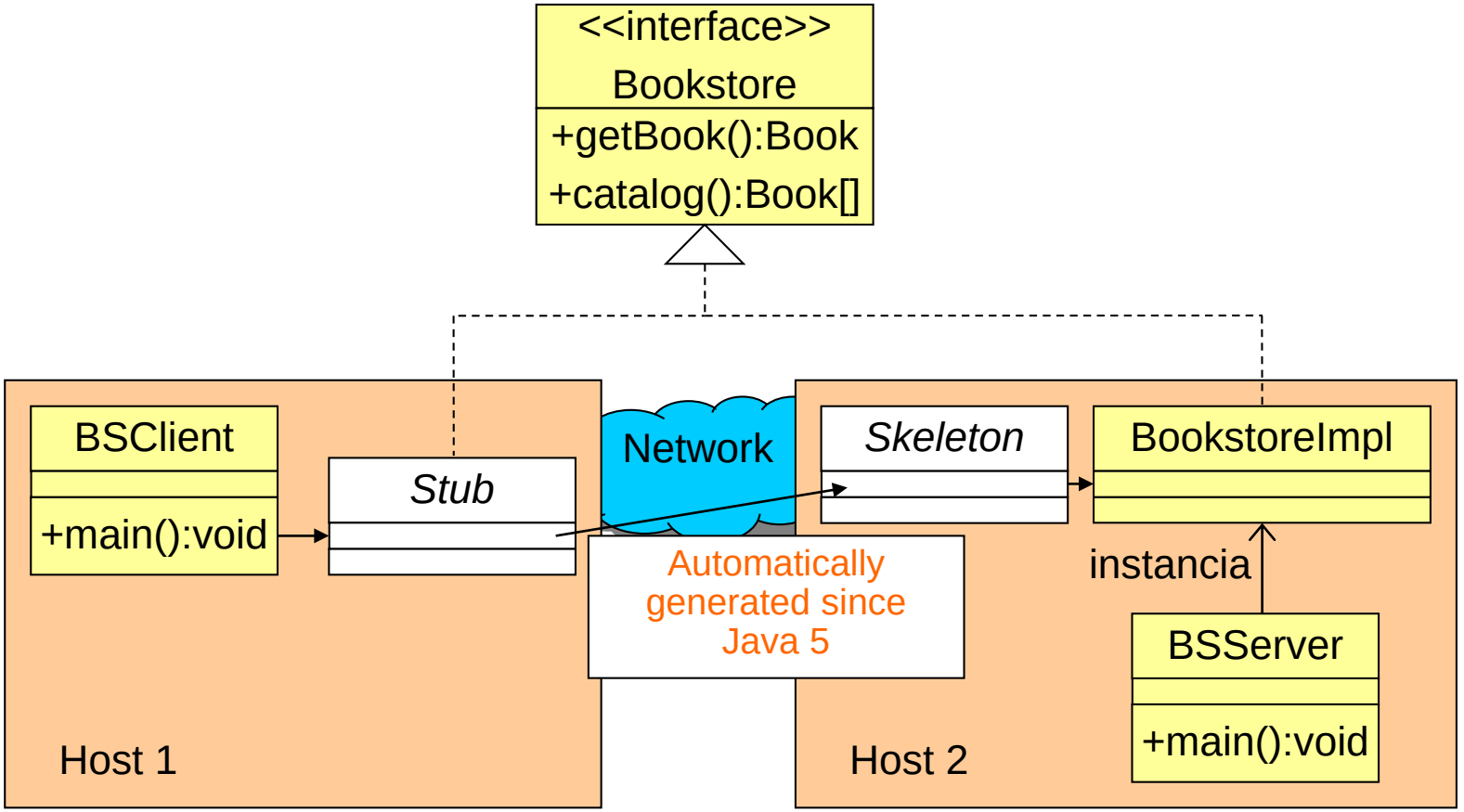
Implementing a Java RMI application

Ciências ULisboa



Example: Java distributed object model

- Java RMI in action!



Example: Java distributed object model

Ciências ULisboa

Bookstore.java

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

public interface Bookstore extends Remote {
    public Book getBook(String name) throws RemoteException;
    public Book[] catalog() throws RemoteException;
    ...
}
```

Book.java

```
import java.io.Serializable;

public class Book implements Serializable {

    private String name;
    private String author;
    private float price;

    public Book(String name, String author, float price) {
        this.name = name;
        this.author = author;
        this.price = price;
    }
}
```

Example: Java distributed object model

Ciências ULisboa

BookstoreImpl.java

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

public class BookstoreImpl extends UnicastRemoteObject implements Bookstore {
    private HashMap<String,Book> books = new HashMap<String,Book>();

    public BookstoreImpl() throws RemoteException {
    }

    public Book getBook(String name) throws RemoteException {
        return books.get(name);
    }
    public Book[] catalog() throws RemoteException {
        return books.values().toArray(new Book[0]);
    }
    ...
}
```

Example: Java distributed object model

Ciências ULisboa

BSServer.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
Import ...

public class BSServer {
  public static void main(String[] args) throws Exception {
    Bookstore bookstore = new BookstoreImpl();
    String address = null;

    try {
        address = System.getProperty("java.rmi.server.hostname");
    } catch (Exception e) { System.out.println("Can't get inet address."); }

    String myID = new String(address + ":" + "Bookstore");

    Registry registry = null;
    try {
        registry = LocateRegistry.createRegistry(BOOKSTORE_PORT);
        registry.rebind("BookstoreServer", bookstore);
    } catch (Exception e) {
        System.out.println("Bookstore: ERROR trying to start server!");
    }
}
```

Example: Java distributed object model

Ciências ULisboa

BSClient.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import ...

public class BSClient {
  public static void main(String[] args) throws Exception {
    String serverHost = (args.length < 1) ? null : args[0];

    try {
        Registry registry = LocateRegistry.getRegistry(serverHost,BOOKSTORE_PORT);
        Bookstore boostoreStub = (Bookstore) registry.lookup(" BookstoreServer ");
        System.out.println(bookstoreStub.getBook("The Filth"));
    } catch (Exception e) {
        System.err.println("Client exception: " + e.toString());
        e.printStackTrace();
    }
}
```