IBD – Intergiciels et Bases de Données

RMI-based distributed systems

Fabien Gaud, Fabien.Gaud@inrialpes.fr

http://www-ufrima.imag.fr/ ⇨ Placard électronique ⇨ M1 Info ⇨ IBD

Overview of lectures and practical work

- Lectures
  - Introduction to distributed systems and middleware
  - Socket-based distributed systems
  - **RMI-based distributed systems**
  - Servlet-based distributed systems
  - Introduction to multi-tier distributed Internet services
- Practical work
  - Programming distributed systems with Sockets
  - Programming distributed systems with RMI
  - Programming distributed systems with Servlets
  - Project on multi-tier Internet services

Motivations

- Sockets are a simple and flexible technology for data communication in distributed systems
  - Sockets are restricted to the transmission of data
  - Sockets say nothing about the semantics of transferred data
- Application-level protocols provide the semantics
  - Often time-consuming and error-prone to develop

Remote Procedure Call (RPC)

- Old technology – Still in use
  - Developed in the 80s, for procedural languages
  - Integrates transparently remote calls into the language
- Remote definition
  - Across address spaces
  - Across networks
- Technical issues for implementing RPC
  - Different address spaces
  - Heterogeneous machines

RPC challenges

- Local Procedure Call (same address space)
  - Arguments are either passed as a pointer or a value
  - Pointers refer to physical memory addresses
  - Values are primitive types (int, float, long, double, etc.)
- Remote Procedure Call (different address spaces)
  - Pointers are only valid within one address space
  - Pointed-to data must be copied across address-space boundaries

RPC challenges (2)

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

RPC versus RMI

- RPC Limitations
  - The naming of RPC destinations (IP, port)
  - Copy-only semantics for arguments
- RMI
  - Suited for Object Oriented Programming Languages (OOPL)
  - Use object identity to "name" the destination of the invocation
  - Can pass "objects" by value or reference

Object oriented programming

- Objects
  - Object identity (unique)
  - Object state (data)
- Classes
  - A class is a factory for its instances (objects)
  - A class defines the structure of its instances
  - Classes define the methods available on objects (behavior)

Object oriented programming

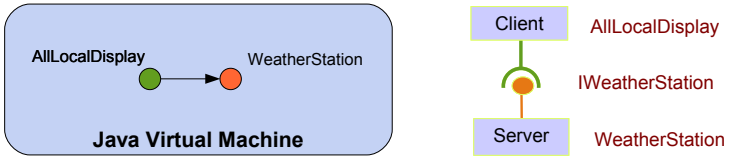
- Interfaces
  - Interfaces are abstract classes
    - Interfaces define a contract as a behavior (methods)
  - A client-server pattern
    - The server class implements the interface
    - The client class invokes methods of the interface

Outline

- Motivations
- **A simple example**
- The architecture of RMI
- A detailed example step by step

Simple Example

- Weather station
  - You just bought a weather station for your week-end house
  - It provides a Java package to access the weather sensors
    - Temperature
    - Wind speed and direction

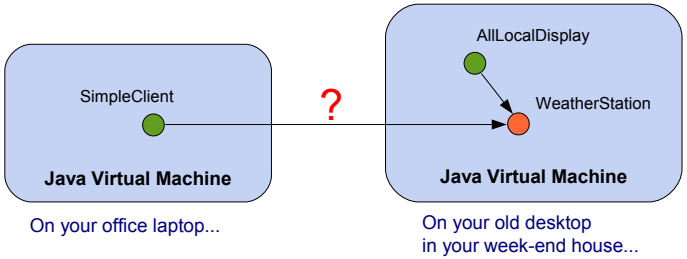


Simple Example

- Looking at the source
  - The all-local case...

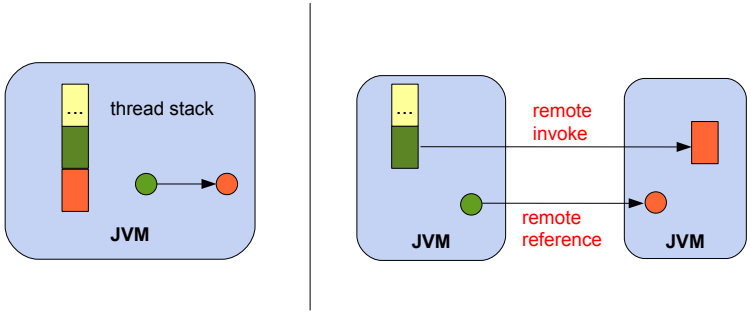
Simple Example

- Planning your week-ends
  - You would like to read the information from your office...
  - How do you access it?
    - Problem: object references are only local to a JVM



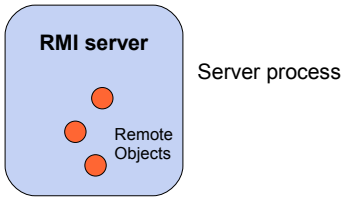
A simple example

- What we need
  - A way to refer to a remote object in a different JVM
  - The ability to remotely invoke methods



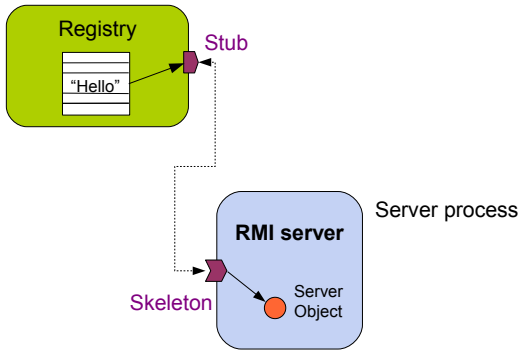
A simple example

- Creates a remote server...
  - Need a process that hosts the Java Virtual Machine
- Hosts one or more remote objects
  - Each remote object
  - Instance of a class that implements one or more ***Remote*** interface



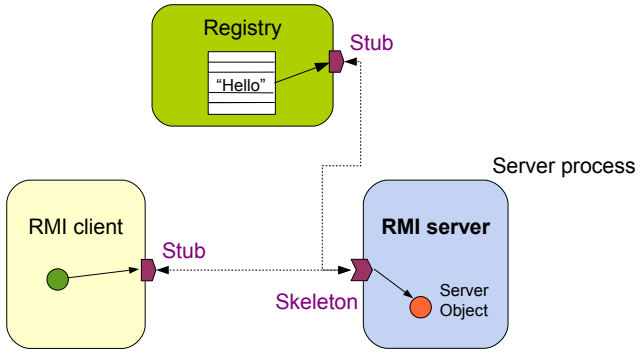
A simple example

- Naming the remote object



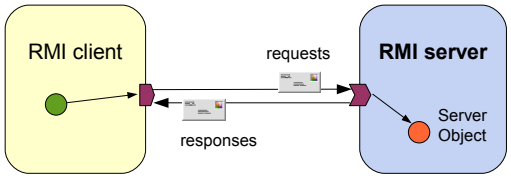
A simple example

- Client looks up the name



A simple example

- Client invokes methods on the remote object
  - Does not involve the registry anymore
    - Goes directly through stubs and skeletons
  - Parameters are marshalized back and forth
    - Strings are passed by value (copied)
    - More on this later...



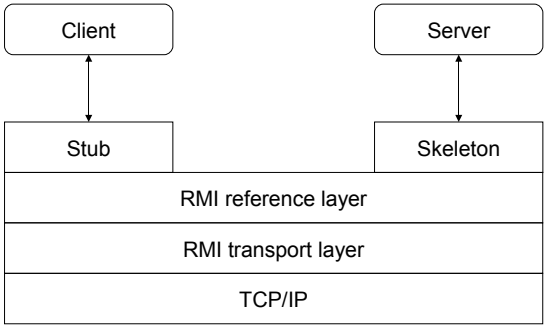
Simple Example

- Looking at the source
  - The making of an RMI server
  - The naming of a remote object
  - The lookup of that remote object by a client
  - The use of that remote object by a client

Outline

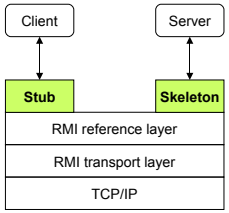
- Motivations
- A simple example
- **The architecture of RMI**
- A detailed example step by step

RMI Architecture



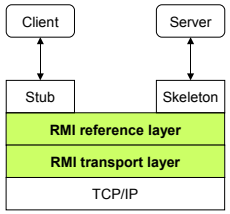
RMI Architecture

- The stub and skeleton layer in RMI
  - Stub
    - Offers the **same remote interfaces** as the remote object
    - Marshalls method arguments in a message
  - Skeleton
    - Receives messages from the stub
      - Forwards calls to the server object
    - Waits for results
      - Sends results back to the stub



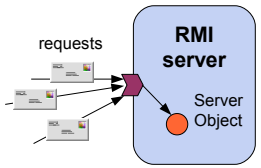
RMI Architecture

- The reference layer in RMI
  - Manages stubs and skeletons
    - Dispatch messages on skeletons
    - Worker thread selection
  - Includes the name service (the registry)
  - Includes distributed garbage collection
- The transport layer in RMI
  - It manages communication connections
    - Either over TCP/IP or HTTP
  - It must not be confused with the network transport layer (e.g. TCP/IP)



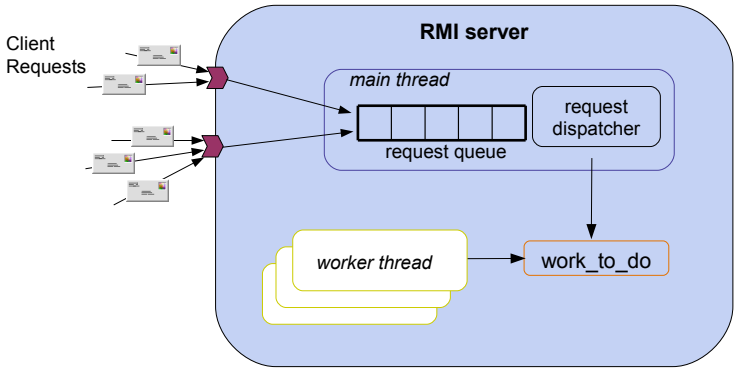
RMI Architecture

- Multi-threaded execution model
  - Server objects may be invoked from several clients
    - Method invocations happen in parallel
  - Server objects must be developed assuming multiple threads
    - Use synchronized methods
    - Use synchronized blocks
- RMI thread pool
  - Manages a pool of threads
  - Pick one thread to carry one invocation



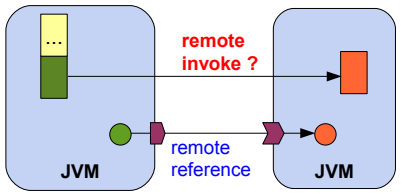
RMI Architecture

- Thread pool details



About invocations

- Coming back on our simple example
  - Remote references are accessible through stubs and skeletons
  - What about remote invocations?
    - How are the integer values passed around?
    - How are the strings passed around?



Argument semantics

- Two semantics
  - By-value or by-reference
    - By-value means a copy
    - By-reference means no copy
  - Applies to arguments and to returned results
- Primitive types
  - They are boolean, byte, char, short, int, float, double
  - Always marshalled by value through stubs and skeletons
- What about objects?
  - Can be either by-value or by-reference...

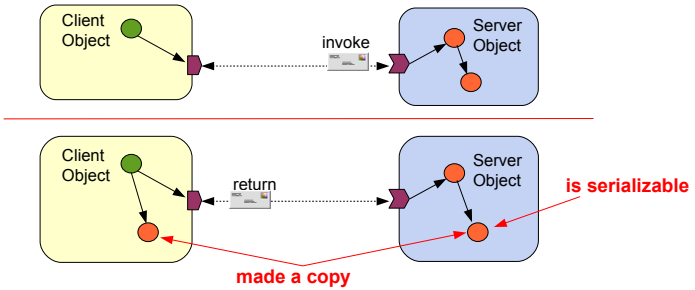
Argument semantics

- Objects by-value
  - Any object which is “serializable”
    - The class of the object implements `java.io.Serializable`
  - Copy semantics
    - Deep copy... yields two objects: both on server and client sides
    - Updates impact only the local copy

Argument semantics

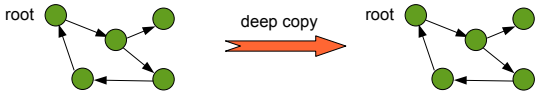
- Objects by-value
  - An example – a simple method returning a reference

```
public Object getObject();
```



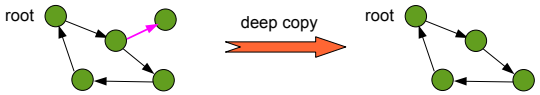
Argument semantics

- Serialization
  - Deep copy
    - Recursive depth-first copy of an object graph from a root
    - If any object encountered is not serializable, an exception is thrown
  - Cycles are properly handled



Argument semantics

- Serialization
  - Individual object copy
    - By default, all instance fields are copied
    - Except for instance fields that are declared transient
  - **Attention**
    - Static fields are part of the class
    - Not part of the instances of that class



Argument semantics

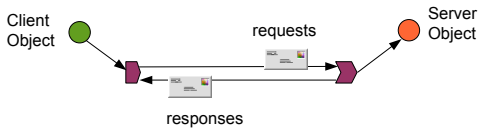
- Java Runtime Environment
  - Most JRE classes are serializable
  - Their instances will be passed by value
- Examples
  - Java collections such as hash tables or vectors
  - String objects
  - Arrays are serializable objects
- Some classes are not serializable
  - Only make sense locally, such as files, sockets, threads, etc.

Back to our example

- Marshalling
  - Integers passed by-value as arguments and return values
  - String objects are passed by-value
- Improving performance
  - Reducing the number of remote method invocations
    - One remote invocations per information
  - Introducing an WeatherData object
    - Gathers all weather information
    - Passed by-value, so it implements serializable

Argument semantics

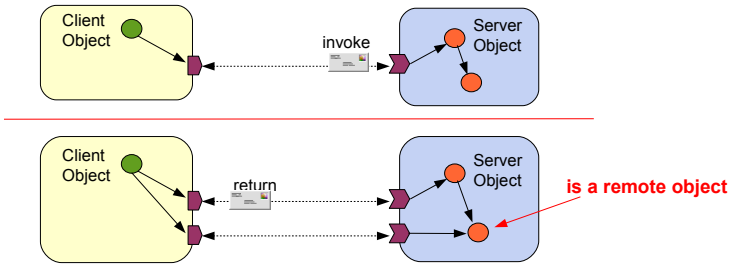
- Objects by-reference
  - All objects whose classes implement Remote interfaces
    - A remote interface extends `java.rmi.Remote`
  - Creates a stub-skeleton chain
    - Carries method invocations up to the remote object
    - Marshalling and unmarshalling arguments along the way
    - Changes appear in the remote object



Argument semantics

- Objects by-reference
  - An example – a simple method returning a reference

```
public Object getObject();
```

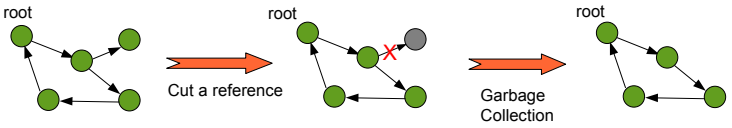


Back to our example

- Introduce weather stats
  - We want to log weather information
  - We introduced a logger object
    - It collects periodically the weather information
    - It is a remote object that gives the last set of weather information
- Impacts on the code
  - Introduces a new interface, a new class, and a new remote object
  - The remote object is a UnicastRemoteObject
  - It is not named however
  - It is returned by reference from a method on the weather station object

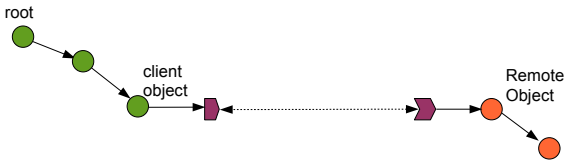
Distributed Garbage Collection

- Local garbage collection
  - Java is a garbage collected language
    - An object is garbage when it is no longer reachable from roots
    - Roots are thread stacks and class statics
  - The garbage collector detects and recycles garbage objects
    - This is done automatically and periodically



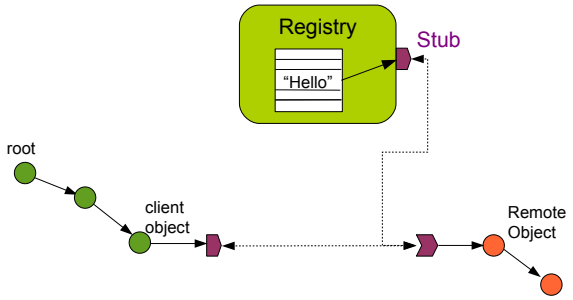
Distributed Garbage Collection

- Distributed garbage collection
  - Natural extension to the local case
    - If a stub is reachable, so is the skeleton
    - If the stub is reachable, so is the remote object



Distributed Garbage Collection

- Distributed garbage collection
  - The RMI registry is a root
  - Named objects are reachable and not garbage collected



Where are classes ?

- Back to our example
- Which classes are needed by
  - The server ?
  - The client ?

Where are classes ? (2)

- Client / Server must have
  - Interfaces for remote objects
  - Implementation for serializable objects
- Code downloading
  - Clients and/or servers may fetch unknown classes
  - Various protocols can be used (http, ftp, ...)
  - Uses a codebase (= path or URL)

Example

```
java ...
     -Djava.rmi.server.codebase=http://mywebsite.com/classes/compute.jar
     ...
     myServer
```


Outline

- Motivations
- A simple example
- **The architecture of RMI**
- A detailed example step by step

A detailed example step by step

- Main steps to create a distributed application with RMI:

Server side	Client side
Define the remote interface provided by the remote object	
Implement the remote object	
Implement the server program	Implement the client program
Compile the source files	Compile the source files
Start the RMI registry	
Start the server	Start the client

Application design

- Determine application architecture
  - Which components are local objects
  - And which components are remotely accessible
  - What components are servers (creators of remote objects) and which are clients (accessors to remote objects)

Remote Interface

- Define remote interfaces
  - A remote interface specifies the methods that can be invoked remotely by a client on remote objects
  - Determine types of objects that will be used as parameters and return values for these methods
    - Using copy
    - Using reference

Remote interfaces

- Define the remote interface provided by the remote object:
  - Extends *java.rmi.Remote*
  - Each method must declare *java.rmi.RemoteException*

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


public interface Hello extends Remote {

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

}
```


Remote objects

- Implement the remote object in a class:
  - Declare the remote interface being implemented
  - Implement the set of methods that can be called remotely
  - Implement any other local method that can not be invoked remotely

```
Import java.rmi.RemoteException;

public  class HelloImp implements Hello {
    private String message;

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

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


Remote objects (2)

- Implement the remote object in a class
  - Objects passed to or returned from remote methods must be Serializable or Remote
  - Remember: each method must declare a *RemoteException*
  - May extend *UnicastRemoteObject* for creating stub automatically otherwise stub must be manually created

Server side

- Implement the server program:
  - Create and install a security manager
  - Create remote objects
  - Eventually create a stub
  - Register remote objects with the RMI registry

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

public  class HelloServer {
   public static void  main(String [] args){
     try {
       if (System.getSecurityManager() == null) { System.setSecurityManager(new SecurityManager());}
       HelloImp h = new HelloImp ("Hello world !");
       Hello h_stub = (Hello) UnicastRemoteObject.exportObject(h, 0);
       Registry registry= LocateRegistry.getRegistry();
       registry.bind("Hello1", h_stub);
     } catch (Exception e)  {
            System.err.println("Error on server :" + e) ; e.printStackTrace(); return;
     }
   }
}
```


Client side

- Implement the client program:
  - Create and install a security manager
  - Get a remote object reference
  - Perform remote method invocations on the remote object

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

public class HelloClient {
  public static void main(String [] args) {
   if (args.lenght < 1) { System.out.println("Usage: java HelloClient <server host>"); return; }
   try {
     if (System.getSecurityManager() == null) { System.setSecurityManager(new SecurityManager()); }
     String host = arg[0];
     Registry registry = LocateRegistry.getRegistry(host);
     Hello h = (Hello) registry.lookup("Hello1");
     String res = h.sayHello(); System.out.println(res);
   } catch (Exception e)  {
           System.err.println("Error on client: " + e); e.printStackTrace(); return;
   }
  }
}
```


RMI Registry

- Make objects accessible “to the world”
  - **Bind** an object name with a reference
  - Provides object search facilities (**lookup**)
- Must be on the same machine as the server.

"For security reasons, an application can only bind, unbind, or rebind remote object references with a registry running on the same host”

- Generally on port 1099
- Accessible through
  - A registry object
  - Static methods of the *Naming* class
    - eg. Naming.lookup(//<host>:<port>/<object>)

Compiling source files

- As with any Java program, use javac compiler to compile the source files
- The source files contain
  - the declarations of the remote interfaces
  - their implementations
  - any other server classes
  - and the client classes
- With versions prior to Java Platform, Standard Edition 5.0
  - an additional step was required to build stub classes
  - by using the rmic compiler
  - however, this step is no longer necessary

Compiling source files (2)

- This example separates
  - The remote interface
  - The remote object implementation class
  - The server program class
  - The client program class
- Compile the remote interface and build a jar file that contains it
  - javac –d classes –classpath .:classes src/Hello.java
  - jar cvf lib/Hello.jar classes/Hello.class
- Compile the remote object implementation class and build a jar file that contains it
  - javac –d classes –classpath .:classes:lib/Hello.jar src/HelloImp.java
  - jar cvf lib/HelloImp.jar classes/HelloImp.class

Running example

- Compile and run server-side and client-side programs:
  - Server-side
    - Compile the server program
      - javac –d classes –classpath .:classes:lib/Hello.jar:lib/HelloImp.jar src/HelloServer.java
    - Start RMI registry
      - rmiregistry &
    - Start the server
      - java –classpath .:classes:lib/Hello.jar:lib/HelloImp.jar HelloServer
  - Client-side
    - Compile the client program
      - javac –d classes –classpath .:classes:lib/Hello.jar src/HelloClient.java
    - Start the client
      - java –classpath .:classes:lib/Hello.jar HelloClient

A note about security

- Enforce by the Security Manager
  - Checks permissions on system resources
    - Files
    - Sockets
    - AWT
    - ...
- RMI exploits the security manager
  - Setting needed permission on sockets (eg. accessing to port 1099)
  - Downloaded code and local code may need to run under different permissions
  - ...

See http://java.sun.com/javase/6/docs/technotes/guides/security/permissions.html

A note about security (2)

- The server and client programs run with a security manager installed
- When either program runs, a security policy file must be specified so that the code is granted the security permissions it needs to run
- Example of a policy file to use with the server

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

Incoming lectures and practical work on middleware

- Lectures
  - Introduction to distributed systems and middleware
  - Socket-based distributed systems
  - RMI-based distributed systems
  - **Servlet-based distributed systems**
  - Introduction to multi-tier distributed Internet services
- Practical work
  - Programming distributed systems with Sockets
  - Programming distributed systems with RMI
  - **Programming distributed systems with Servlets**
  - Project on multi-tier Internet services

References

This lecture is extensively based on:

- Sun Microsystems. *Java Tutorial on RMI*. http://java.sun.com/docs/books/tutorial/rmi/
- M. Boger. *Java in Distributed Systems: Concurrency, Distribution and Persistence*. Wiley, 2001.

- This lecture is based on lectures given by Sara Bouchenak