# Building distributed systems with RMI

Sara Bouchenak

Sara.Bouchenak@imag.fr
http://membres-liglab.imag.fr/bouchenak/teaching/

## Motivations

- Sockets are a simple and flexible technology for data communication in distributed systems
- Sockets are restricted to the transmission of data
- Sockets leave the semantics of this data unconsidered
- Protocols which provide the semantic interpretation of the data must be developed on the application level
- The development of such protocols is often time-consuming and error-prone



## Motivations (2)

- Object-oriented programming already provides a framework for semantics of data – the objects
- In local applications, objects communicate via methods
- It would be desirable, for distributed applications, to have a similar communication paradigm available
- Such a communication paradigm would permit the remote call of methods
- Java provides the Remote Method Invocation mechanism: *RMI*



## Remote Procedure Call (RPC)

- RPC is a technology developed in the 80s to call procedures on remotes computers (with the procedural programming paradigm)
- RPC allows the call of procedures located in another process space on a remote computer (or on the same machine)
- Technical issues for implementing RPC
  - Different addess spaces
  - Heterogeneous machines

## RPC implementation

- Different address spaces
  - In the local case:
    - Data used in a procedure call is simply passed as a reference (a pointer)
    - This reference refers to a physical memory address
    - Such a reference has no correct meaning in a different address space
  - In the distributed case:
    - The referenced data used in a remote call needs to be passed as a copy



## RPC implementation (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



## Outline

1. Motivations
2. **Overview of RMI-based distributed applications**
3. A simple example
4. Methodology to build distributed applications using RMI
5. The architecture of RMI
6. A detailed example step by step



## Overview of RMI-based distributed applications

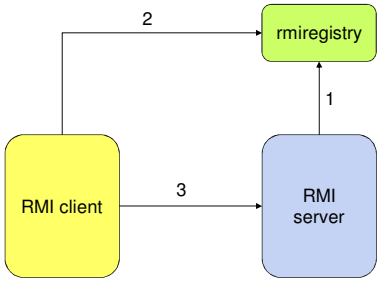
- RMI applications comprise two separate programs, a server and a client
- The server program:
  - creates some remote objects,
  - makes references to these objects accessible,
  - and waits for clients to invoke methods on these objects.
- The client program:
  - obtains a remote reference to one or more remote objects on a server,
  - and then invokes methods on them.
- RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

## Overview of RMI-based distributed applications (2)

- Distributed object applications follow these steps:
  - Client locates remote objects on server
    - Various mechanisms to obtain references to remote objects
    - An application server can register its remote objects with RMI registry (RMI's simple naming facility)
    - An application server can return remote object references as part of other remote invocations
  - Client communicates with remote objects on server
    - To the client programmer, remote communication looks similar to regular Java method invocations
    - Details of remote communication are handled by RMI and transparent to client and server programs
  - Load class definitions for objects that are passed around
    - Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definition



## Overview of RMI-based distributed applications (3)






## Remote objects

- Objects with methods that can be invoked across Java virtual machines are called *remote objects*
- An object becomes remote by implementing a *remote interface*, which has the following characteristics:
  - A remote interface extends the interface *java.rmi.Remote*
  - Each method of the interface declares *java.rmi.RemoteException* in its throws clause, in addition to any application-specific exceptions



## Outline

1. Motivations
2. Overview of RMI-based distributed applications
3. **A simple example**
4. Methodology to build distributed applications using RMI
5. The architecture of RMI
6. A detailed example step by step

# A simple example

**Remote object – Interface definition**

```
import java.rmi.*;


public interface Hello
    extends Remote {

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

}
```



# A simple example (2)

**Remote object – Interface definition**

```
import java.rmi.*;


public interface Hello
    extends Remote {

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

}
```

**Remote object – Class implementation**

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

public  class HelloImp
    implements Hello {

    private String message;

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

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

}
```



# A simple example (3)

**Server**

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

public class HelloServer {

  public static void  main(String [] args){
     ...

     try {
        // Create a Hello remote object
        HelloImp h =
              new HelloImp ("Hello world !");
        Hello h_stub = (Hello)
        UnicastRemoteObject.exportObject(h, 0);

        // Register the remote object in RMI
        // registry with a given identifier
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("Hello1", h_stub);

        System.out.println ("Server ready");

     } catch (Exception e)  {
              System.err.println("Error on server :"
              + e) ;
              e.printStackTrace();
              return;
     }
  }
}
```



# A simple example (4)

**Server**

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

public class HelloServer {

  public static void  main(String [] args){
     ...

     try {
        // Create a Hello remote object
        HelloImp h =
              new HelloImp ("Hello world !");
        Hello h_stub = (Hello)
        UnicastRemoteObject.exportObject(h, 0);

        // Register the remote object in RMI
        // registry with a given identifier
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("Hello1", h_stub);

        System.out.println ("Server ready");

     } catch (Exception e)  {
              System.err.println("Error on server :"
              + e) ;
              e.printStackTrace();
              return;
     }
  }
}
```

**Client**

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

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

       try {
       if (args.lenght < 1) {
          System.out.println("Usage:
                java HelloClient <server host>");
          return;
       }
       String host = arg[0];

       // Get remote object reference
       Registry registry =
          LocateRegistry.getRegistry(host);
       Hello h = (Hello) registry.lookup("Hello1");

       // Remote method invocation
       String res = h.sayHello();

       System.out.println(res);

       } catch (Exception e)  {
              System.err.println("Error on
              client: " + e) ;
       }
  }
}
```

# Outline

1. Motivations
2. Overview of RMI-based distributed applications
3. A simple example
4. **Methodology to build distributed applications using RMI**
5. The architecture of RMI
6. A detailed example step by step



# Steps to build distributed applications with RMI

- Using RMI to develop a distributed application involves the following general steps:
  1. Designing and implementing the components of the distributed application.
  2. Compiling sources.
  3. Making classes network accessible.
  4. Starting the application.



## Designing and implementing the components of the distributed application

- 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)
- Define remote interfaces
  - A remote interface specifies the methods that can be invoked remotely by a client on remote objects
  - The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods
  - If any of these interfaces or classes do not yet exist, they need to be defined as well
  - Client program accesses remote interfaces, not to the implementation classes of those interfaces



## Designing and implementing the components of the distributed application (2)

- Implementing remote objects
  - Remote objects must implement one or more remote interfaces
  - The remote object class may include implementations of other interfaces and methods that are available only locally
  - If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well
- Implementing servers
  - Servers that create remote objects and provide access to them can be implemented at any time after the remote objects are implemented
- Implementing clients
  - Clients that use remote objects can be implemented at any time after the remote interfaces are defined

## Steps to build distributed applications with RMI

- Using RMI to develop a distributed application involves the following general steps:
  1. Designing and implementing the components of the distributed application
  2. **Compiling sources**
  3. Making classes network accessible
  4. Starting the application



## Compiling source code

- 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



## Steps to build distributed applications with RMI

- Using RMI to develop a distributed application involves the following general steps:
  1. Designing and implementing the components of the distributed application
  2. Compiling sources
  3. **Making classes network accessible**
  4. Starting the application



## Making classes network accessible

- Certain class definitions are made network accessible
  - such as the definitions for the remote interfaces
  - and their associated types,
  - and the definitions for classes that need to be downloaded to the clients or servers
- Class definitions are typically made network accessible through a web server

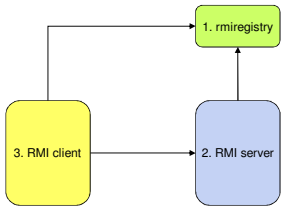
# Steps to build distributed applications with RMI

- Using RMI to develop a distributed application involves the following general steps:
  1. Designing and implementing the components of the distributed application
  2. Compiling sources
  3. Making classes network accessible
  4. **Starting the application**



# Starting the application

- Starting the application includes
  - running the RMI remote object registry
    - using the *rmiregistry* tool
  - the server
    - using the *java* tool
  - and the client
    - using the *java* tool






# Outline

1. Motivations
2. Overview of RMI-based distributed applications
3. A simple example
4. Methodology to build distributed applications using RMI
5. **The architecture of RMI**
6. A detailed example step by step



# The architecture of RMI

- Transparency regarding distribution
  - RMI offers full trasparency regarding distribution
  - After a first initialization, a call can be used in exactly the same way as in the local case

# The architecture of RMI (2)

Server

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

public class HelloServer {

  public static void  main(String [] args){
     ...
     try {
        // Create a Hello remote object
        HelloImp h =
              new HelloImp ("Hello world !");
        Hello h_stub = (Hello)
        UnicastRemoteObject.exportObject(h, 0);

        // Register the remote object in RMI
        // registry with a given identifier
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("Hello1", h_stub);

        System.out.println ("Server ready");

     } catch (Exception e)  {
              System.err.println("Error on server :"
              + e) ;
              e.printStackTrace();
              return;
     }
  }
}
```

Client

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

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

     try {
     if (args.lenght < 1) {
        System.out.println("Usage:
              java HelloClient <server host>"); return;
     }
     String host = arg[0];
     // Get remote object reference
     Registry registry =
        LocateRegistry.getRegistry(host);
     Hello h = (Hello) registry.lookup("Hello1");

     // Remote method invocation
     String res = h.sayHello();
     System.out.println(res);
     } catch (Exception e)  {
              System.err.println("Error on
              client: " + e) ;
     }
  }
}
```

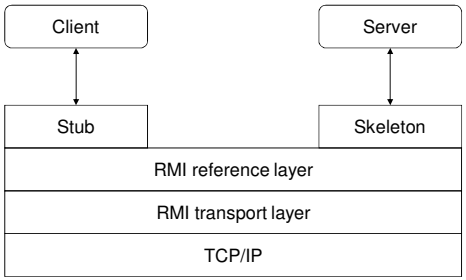


# The architecture of RMI (3)

- Transparency regarding distribution
  - RMI offers full trasparency regarding distribution
  - After a first initialization, a call can be used in exactly the same way as in the local case
- RMI client and server
  - Both client and server are normal objects implemented in Java
  - The server must document the interface it provides for remote access
  - From this description, additional classes are automatically created by a special compiler
  - These classes internally take care of communication handling between client and server
  - These classes are known as *Stub* (client-side) and *Skeleton* (client-side)



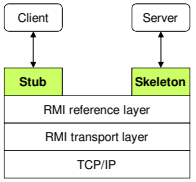
# Layers of RMI architecture





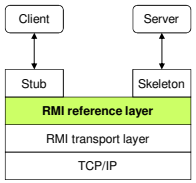

# The architecture of RMI (4)

- The stub and skeleton layer in RMI
  - Stub
    - A placeholder object which offers the same interface as the server object
  - Skeleton
    - The skeleton takes the calls of the stub
    - It processes them
    - It forwards the call to the server object
    - It waits for the result
    - It sends the result back to the stub



# The architecture of RMI (5)

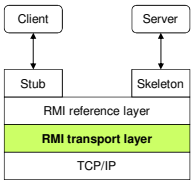
- The reference layer in RMI
  - It finds the respective communication partners
  - It includes the name service, the *registry*






# The architecture of RMI (6)

- The transport layer in RMI
  - It manages communication connections
  - It handles communication
  - It must not be confused with the network transport layer (e.g. TCP/IP)



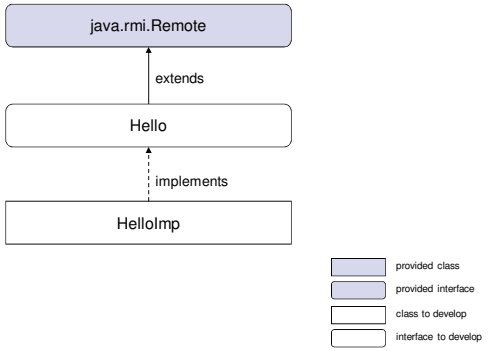



# Outline

1. Motivations
2. Overview of RMI-based distributed applications
3. A simple example
4. Methodology to build distributed applications using RMI
5. The architecture of RMI
   - Architecture
   - **Class hierarchy in RMI**
6. A detailed example step by step



# Class hierarchy in RMI



# Class hierarchy in RMI (2)

- Package ***java.rmi.server***
  - Provides classes and interfaces for supporting the server side of RMI
- Class ***java.rmi.server.RemoteObject***
  - Distributed objects do not inherit directly from *Object*
  - They inherit from *RemoteObject*
  - *RemoteObject* implements the *Object* behavior for remote objects (e.g. methods *hashCode*, *equals*, and *toString* are reimplemented)
- Class ***java.rmi.server.UnicastRemoteObject***
  - Provides support for point-to-point active object references (invocations, parameters, and results)
  - Uses TCP streams



# Class hierarchy in RMI (3)

- Package ***java.rmi***
  - Provides the RMI package
- Interface ***java.rmi.Remote***
  - Identifies interfaces whose methods may be invoked from a non-local virtual machine
  - Any object that is a remote object must directly or indirectly implement this interface
  - Only those methods specified in a "remote interface" (an interface that extends java.rmi.Remote) are available remotely
- Class ***java.rmi.RemoteException***
  - Communication-related exceptions that may occur during the execution of a remote method call
  - Each method of a remote interface (an interface that extends *java.rmi.Remote*) must list *RemoteException* in its throws clause



# Class hierarchy in an example

**Remote object – Interface definition**

```
import java.rmi.*;


public interface Hello
    extends java.rmi.Remote {

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

}
```

**Remote object – Class implementation**

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

public  class HelloImp
    implements Hello {

    private String message;

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

    public String sayHello ()
            throws java.rmi.RemoteException
    {
            return message ;
    }

}
```



# Outline

1. Motivations
2. Overview of RMI-based distributed applications
3. A simple example
4. Methodology to build distributed applications using RMI
5. The architecture of RMI
6. **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 regsitry | |
| Start the server | Start the client |



## A detailed example step by step (2)

- Define the remote interface provided by the remote object:
  - Extends *java.rmi.Remote*
  - Defines the set of methods that can be called remotely
  - 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;

}
```



## A detailed example step by step (3)

- 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 () RemoteException {
          return message ;
    }
}
```



## A detailed example step by step (4)

- Implement the remote object in a class – Passing objects in RMI
  - Arguments to remote methods or return values from remote methods can be of any type
    - Primitive data types (e.g. int, float, etc.)
    - Remote objects
    - Local objects
  - Objects passed to or returned from remote methods must be *serializable*
    - They must implement the *java.io.Serializable* interface
  - Some object types do not meet any of these criteria; they cannot be passed to or returned from remote methods
    - Most of these objects, such as threads or file descriptors, encapsulate information that makes sense only within a single address space
    - Many of the core classes (e.g. classes in the packages *java.lang* and *java.util*) implement the *Serializable* interface

## A detailed example step by step (5)

- How arguments and return values are passed in remote method invocations
  - Remote objects are essentially passed by reference
    - A remote object reference is a stub
    - It is a client-side proxy that implements the remote interface that the remote object implements
    - Passing a remote object by reference means that any changes made to the state of the object by remote method invocations are reflected in the original remote object
  - Local objects are passed by copy
    - Using object serialization
    - By default, all fields are copied except fields that are marked static or transient
    - Default serialization behavior can be overridden on a class-by-class basis
    - A copy of the object is created in the receiving Java virtual machine
    - Any changes to the object's state by the receiver are reflected only in the receiver's copy, not in the sender's original instance
    - Any changes to the object's state by the sender are reflected only in the sender's original instance, not in the receiver's copy



## A detailed example step by step (6)

- Implement the server program:
  - Create and install a security manager
  - Create and export remote objects
  - 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);
       System.out.println ("Server ready");
     } catch (Exception e)  {
            System.err.println("Error on server :" + e) ; e.printStackTrace(); return;
     }
   }
}
```



## A detailed example step by step (7)

- Implement the server program – Create and install a Security Manager
  - The first task of the server program is to create and install a security manager
  - This protects access to system resources from untrusted downloaded code running within the Java virtual machine
  - A security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations
  - If an RMI program does not install a security manager, RMI will not download classes (other than from the local class path) for objects received as arguments or return values of remote method invocations
  - This restriction ensures that the operations performed by downloaded code are subject to a security policy



## A detailed example step by step (8)

- Implement the client program:
  - Create and install a security manager
  - Get a remot 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;
   }
  }
}
```

## A detailed example step by step (9)

- Compile source files
  - 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**



## A detailed example step by step (10)

- 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 detailed example step by step (11)

- A note about security
  - 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 (named *server.policy*) to use with the server

```
grant codeBase "file:/home/ann/src/" {
   permission java.security.AllPermission;
};
```

  - Example of a policy file (named *client.policy*) to use with the client

```
grant codeBase "file:/home/john/src/" {
   permission java.security.AllPermission;
};
```



## 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 partly based on lectures given by Sacha Krakowiak, http://sardes.inrialpes.fr/people/krakowia/

# Agenda

| Lecture, Tuesday, 09:45 – 12:45 | Lab, Tuesday, 09:45 – 12:45 |
|---|---|
| *Introduction to distributed systems* | |
| | **Distributed applications with RMI (Part I)** |
| Distributed Web applications | |
| | Distributed applications with RMI (Part II) |
| Interruption week | |
| Event-based systems & MapReduce systems | |
| | Distributed Web applications with Servlets (Part I) |
| Cloud computing | |
| | Distributed Web applications with Servlets (Part II) |
| Advanced techniques for efficient distributed systems | |
| | Caching with Memcached |
| Event-based systems & MapReduce systems | |
| Interruption week | |
| Advanced techniques for dependable distributed systems | |
| | Evaluation |

53