EDA095
Remote Method Invocation

Pierre Nugues

Lund University
http://www.cs.lth.se/home/Pierre_Nugues/

April 21, 2010

Covers: Elliotte Rusty Harold, *Java Network Programming*, 3rd ed., Chapter 18, pages 610–640, O'Reilly.

Are Sockets a Good Programming Paradigm?

To request a service from a server, sockets use explicit input and output methods:

1. `send command`
2. `receive value`

This does not fit well with the paradigm of most programming languages: functions or methods
In addition, communications take the form of unstructured byte streams
The programmer must manage communication explicitly
The architecture of a distributed system must use different structures in the networked part and the local part

The Remote Procedure Call Model

The remote procedure call (RPC) approach is a unified model to deal with local as well as with remote services

It allows a program or a class to call a function (a procedure) or a method running in another process

The other process can be running on the same machine or on a remote one

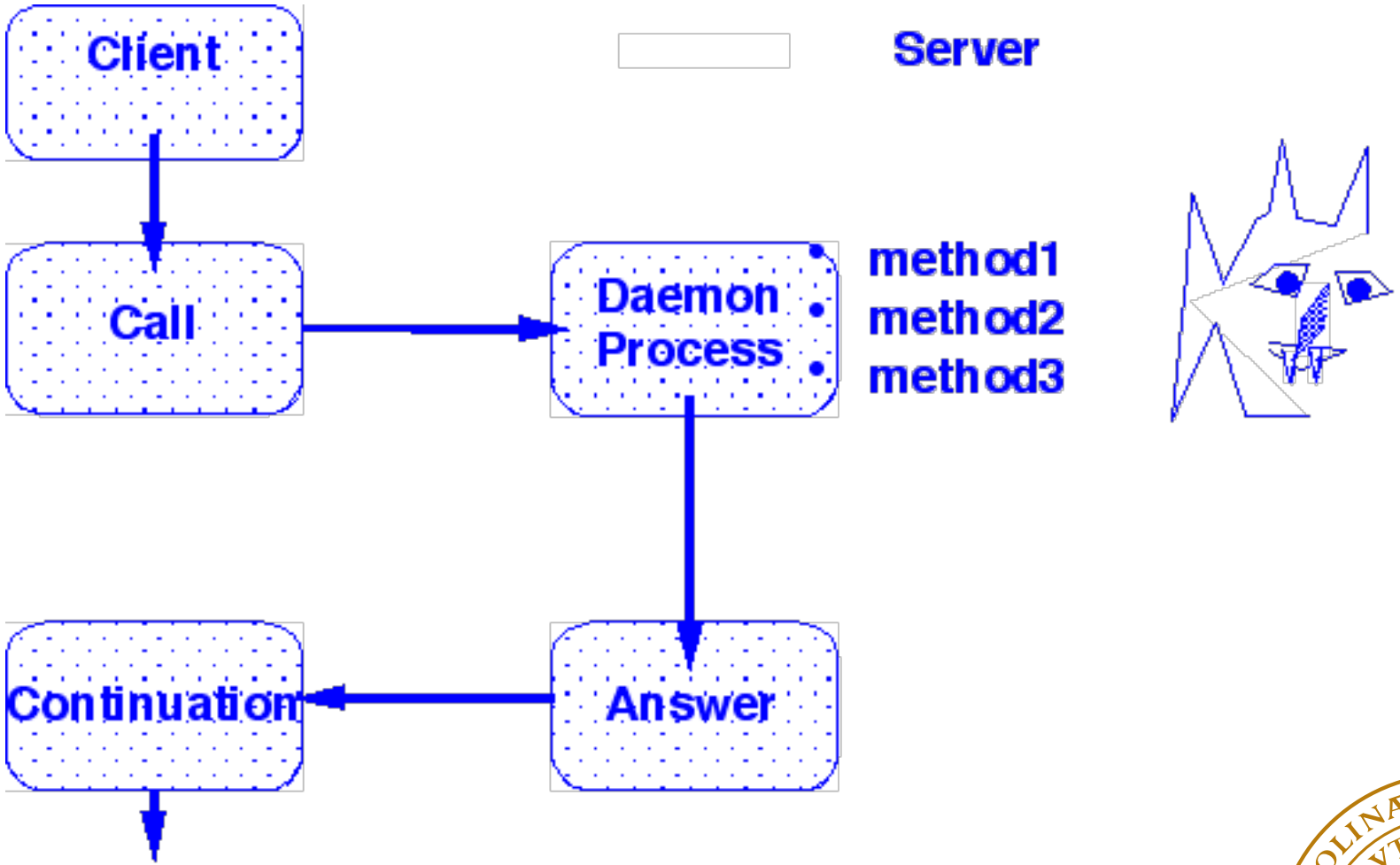
The location of the procedure is transparent. There is no explicit send or receive

Instead of:

```
send(command1, server, params)                    function1(params)
send(command2, server, params)   you have         function2(params)
send(command3, server, params)                    function3(params)
```

RMI page: `http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html`

The Remote Procedure Call Architecture



This model is very simple and very powerful at the same time

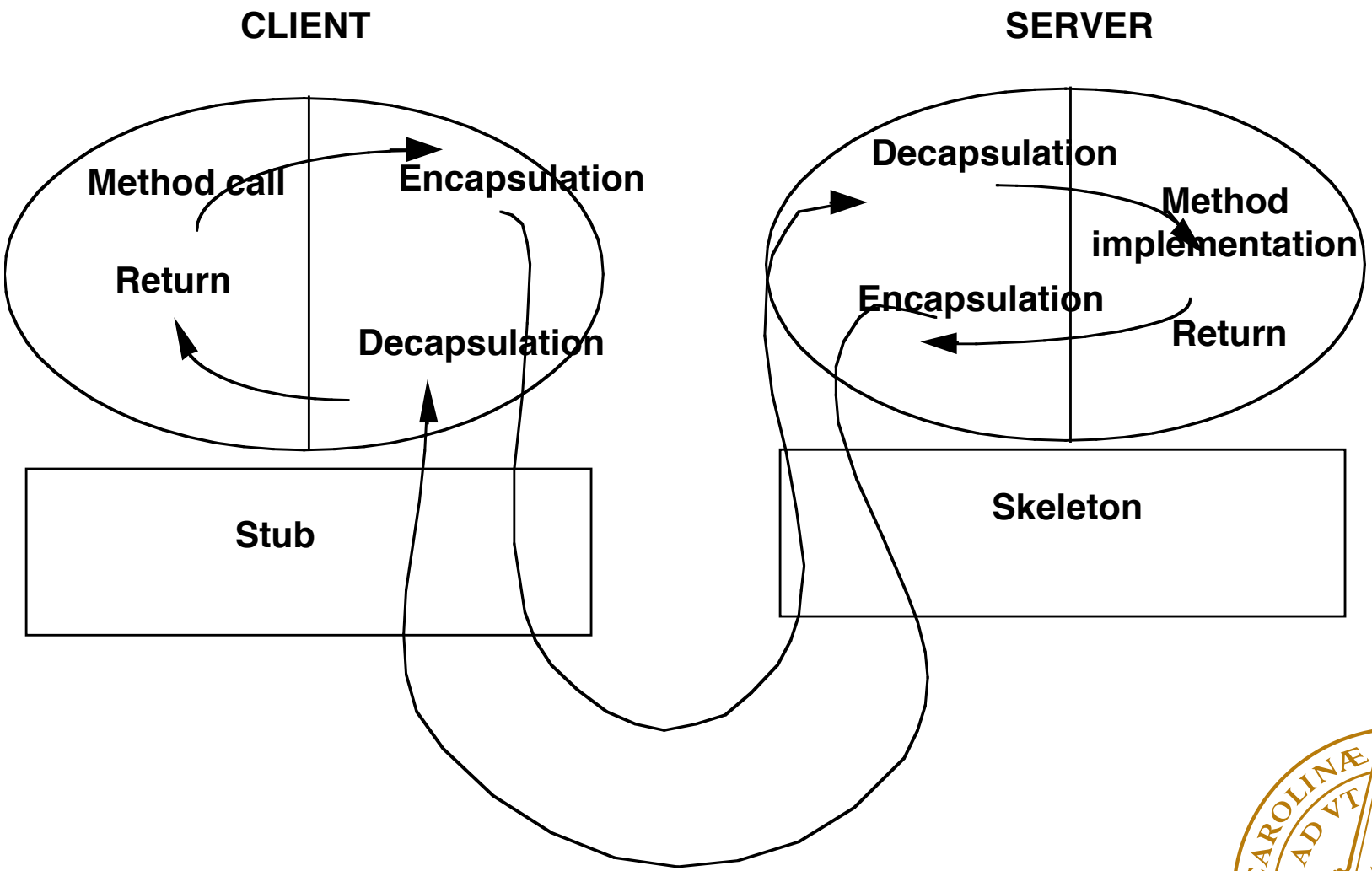
Parameter Passing with RMI

There is no miracle however
The trick is in a hidden layer called a stub on the client side and a skeleton on the server side
It converts the remote call `method(params)` into
`send(command, parameters)`
The stub encapsulates the function name and the arguments in a network packet

```
method(a, b) --> method, a, b
```

The encapsulation is sometimes a complex operation because it takes into account local and distant objects and references
This process is called **marshaling**

Parameter Passing with RMI



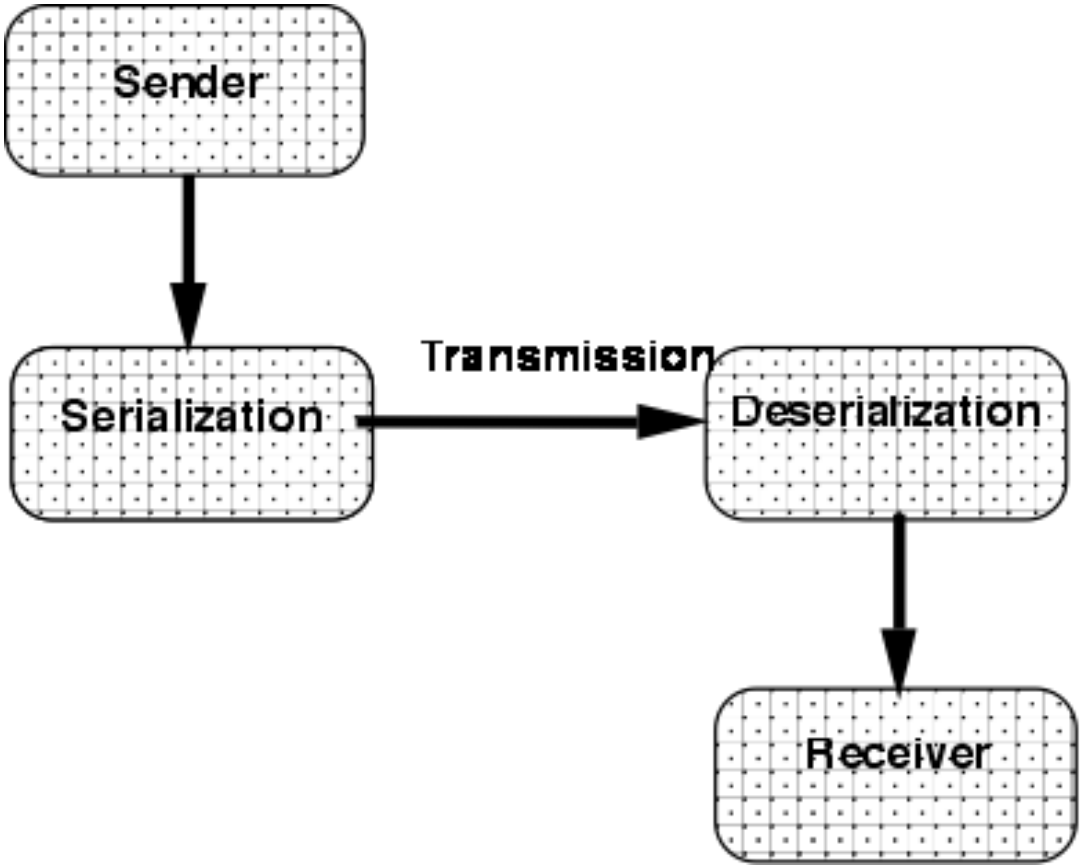
Marshaling

Java methods pass values for primitive types and references for the other objects
References are addresses and they must be linked to values by the methods
In the call `myMethod(myArray)`, `myArray` is living in the memory of the client. The server has no access to it.
The RMI mechanism copies the content of objects and sends it to the remote party

```
myMethod, myArray[0], myArray[1], myArray[2], ...,
 myArray[n - 1], myArray[n]
```

Objects are serialized: they are converted into a stream of bytes and deserialized – reassembled – by the receiver
RMI objects must implement the `Serializable` interface

Serialization



Serialization in Java

Serialization is also used to save/load the state of an object to/from a file
This operation is recursive when fields correspond to other objects or for vectors
The serialized object must include a version – a key that documents its version
Otherwise it is computed automatically. May introduce bug with changes in Java versions

Serialization in Java (II)

```
class MBox extends Object implements Serializable {
  private static final long serialVersionUID = 1L;
  protected String message;
  public MBox() { message = null; }
  public MBox(String message) { this.message = message; }
  public String getMessage() {
    String tempMessage = message;
    message = null;
    return tempMessage;
  }
  public void setMessage(String message) {
    this.message = message;
  }
}
```

Serialization in Java (III)

```
import java.io.*;
public class SerialMBox extends Object {
  public static void main (String args[]) {
    MBox mBox1 = new MBox("first box");
    MBox mBox2 = new MBox("second box");
    try {
      File file = new File("myObjects");
      ObjectOutputStream oos =
        new ObjectOutputStream(new FileOutputStream(file));
      oos.writeObject(mBox1);
      oos.writeObject(mBox2);
      oos.close();
      ObjectInputStream ois =
        new ObjectInputStream(new FileInputStream(file));
      MBox mbox3 = (MBox) ois.readObject();
      ois.close();
```

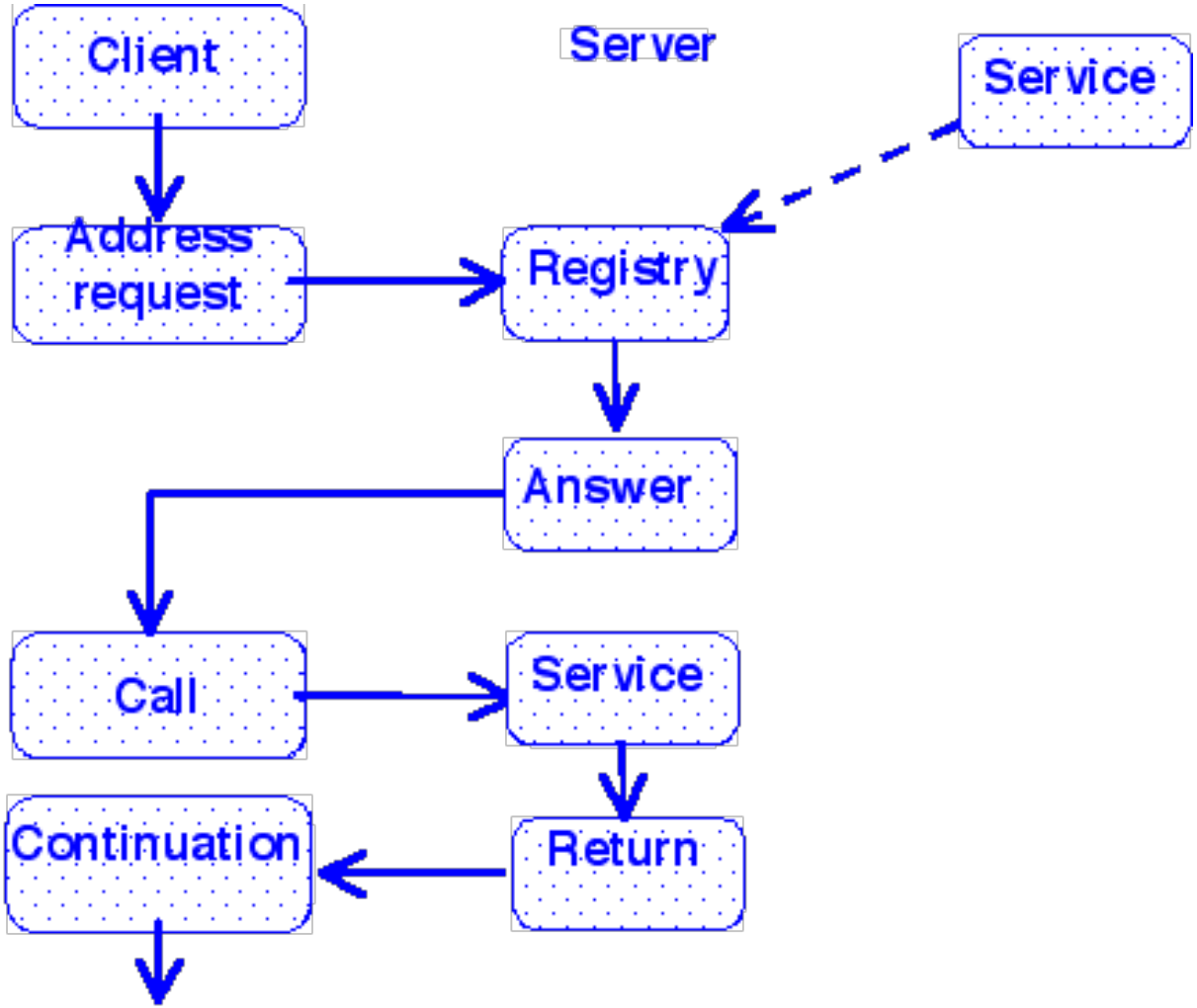
Serialization in Java (IV)

```
      System.out.println(mbox3.getMessage());
    } catch (Exception e) {
      e.printStackTrace();
      System.err.println(e);
    }
  }
}
```

Naming Services

Socket services have a port that might be different on different machines.
This might be suitable for services with a limited distribution
Widely used services such as ftp or telnet need to have "well-known" port numbers reserved on nearly all the machines in the world
This is not very flexible
RPC and RMI used directory services called respectively the port mapper – or sunrpc – and the registry
Services register to the registry and clients call the registry to know the address of a service before they request the service

The Registry



The RMI URL

A *uniform resource locator* (URL) identifies RMI services

`rmi://hostname/ServiceName`

For instance

`rmi://torin.cs.lth.se/MyService`

The registry is launched with the command

```
% rmiregistry
```

The server classes must be accessible through the class path

By default, the registry uses the port number 1099, but you can change it

```
% rmiregistry 2002
```

The service address is then:

`rmi://torin.cs.lth.se:2002/MyService`

The Protocol Stack

The RMI and RPC applications use classical network protocols
The RPCs can use both TCP and UDP and RMIs use TCP for the transport
On top of TCP, Sun uses the RMI transport protocol by default, but there are other standards
See `http://java.sun.com/javase/6/docs/platform/rmi/spec/rmi-protocol.html`
This protocol is handled in the stubs and is transparent
A new protocol makes RMI and Corba compatible: RMI-IIOP (Internet Inter-Orb Protocol)
It is available with Java 1.5 and 6.0 (also 1.3 and 1.4)

Programming with RMI

Although they are widely used, sockets may be considered to be lower level
On the contrary, RMI are closer to classical functional programming
They enable a programmer to build more easily distributed objects and distributed applications
A disadvantage is that the programmer has to resort more on existing classes, tools, and the deployment is much more complex
First steps in RMI design resemble the assembly of classes and the administration of existing software rather than creative programming

Parts of a RMI System

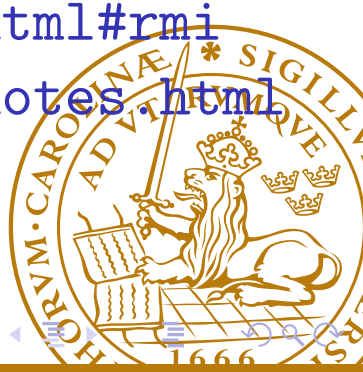
A RMI system has several components:

- The description of the remote services (methods): a Java interface
- The implementation of the remote services (methods): a Java class
- The naming service: the registry that is launched by `rmiregistry`
- The server that launches the remote services: a Java class
- The client that will use the remote services: a Java class

Before Java 1.5, you also needed to generate stubs and skeletons using the `rmic` tool. Now this is automatic

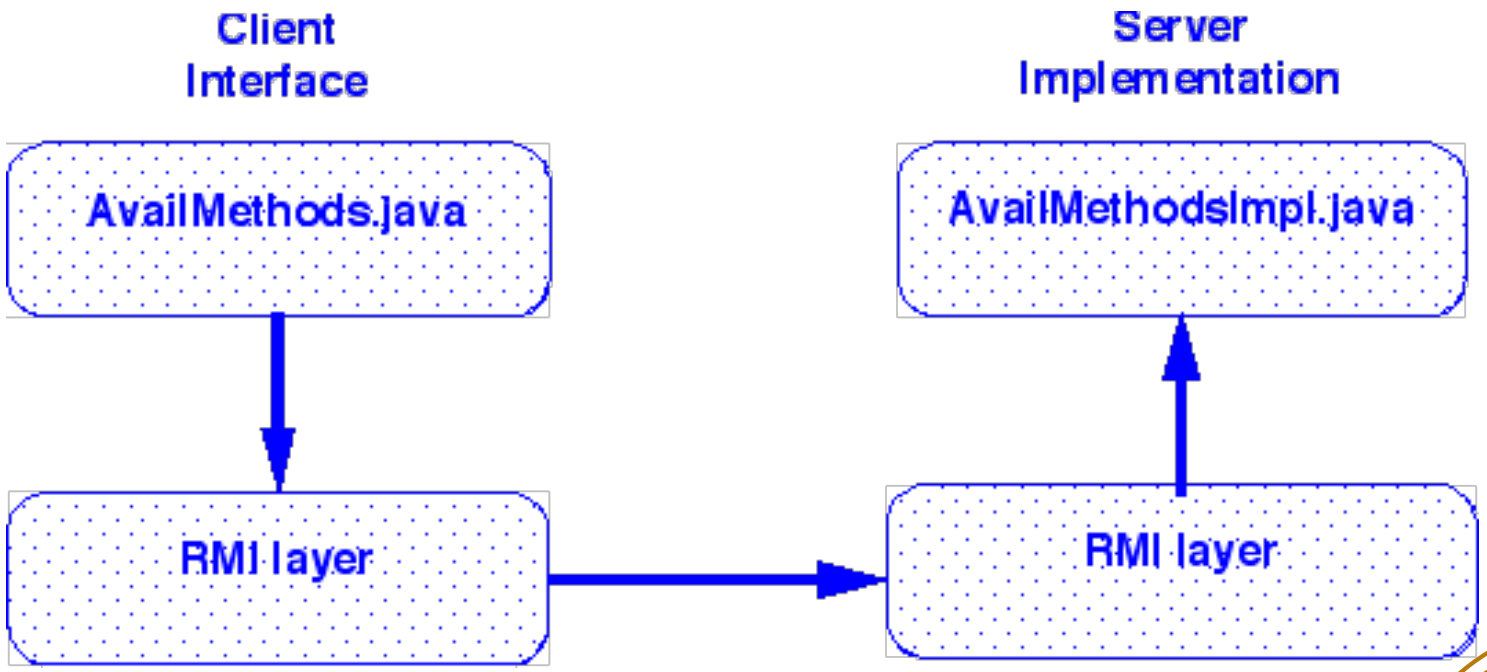
`http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#rmi`

`http://java.sun.com/j2se/1.5.0/docs/guide/rmi/relnotes.html`



The Interface and the Implementation

The interface is the description of the methods available to a client
The implementation is the real Java code



Designing the Interface

The interface breaks down the server into services
A service is mapped onto one method
Originally, RPCs were used to carry out heavy computations on supercomputers and to off-load client machines
A simple example with three arithmetic methods

- `double add(double a, double b)`
- `double subtract(double a, double b)`
- `double sqrt(double a)`

The interface extends the `java.rmi.Remote` class
Each method throws a `java.rmi.RemoteException`

The SimpleArith Interface

```
import java.rmi.*;

public interface SimpleArith extends Remote {
  public double add(double a, double b)
    throws RemoteException;
  public double subtract(double a, double b)
    throws RemoteException;
  public double sqrt(double a) throws RemoteException;
}
```

The SimpleArith Implementation

It is the real method code

```
import java.rmi.*;
import java.rmi.server.*;
public class SimpleArithImpl extends UnicastRemoteObject
    implements SimpleArith {
  public SimpleArithImpl() throws RemoteException {}
  public double add(double a, double b)
      throws RemoteException {
    return a + b;
  }
  public double subtract(double a, double b)
      throws RemoteException {
    return a - b;
  }
  public double sqrt(double a) throws RemoteException {
    return Math.sqrt(a);
  }
```

The SimpleArith Implementation

The class must implement all the methods of the interface
It may contain other methods but the client cannot call them directly
It extends the `UnicastRemoteObject` that enables it to "export" the methods to the RMI system.
They can be called then.
The server can listen on an anonymous port

Stubs and Skeletons

For versions before 1.5, the RMI compiler produces the stub and the skeleton from the implementation

```
% rmic SimpleArithImpl (or rmic -d . SimpleArtithImpl)
% ls

 SimpleArith.class                SimpleArithImpl_Stub.class
 SimpleArithImpl_Skel.class       SimpleArithImpl.java
 SimpleArithImpl.class            SimpleArith.java
%
```

From version 1.2, Java has a reflection API that enables to inspect classes. The skeleton is not necessary if the server and the clients are all running Java version 1.2 or higher.

The option `-keep` keeps the Java files and `-v1.2` doesn't generate skeleton files. (Default in Java 1.5)

The Server

The server launches the services

```
import java.rmi.*;
public class SimpleArithServer {
  public static void main(String [] args) {
    try {
      SimpleArith simple = new SimpleArithImpl();
      Naming.rebind("SimpleArith", simple);
      System.out.println("RMI server running");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The Client

```
import java.rmi.*;
public class SimpleArithClient {
  public static void main(String [] args) {
    try {
      SimpleArith simple = (SimpleArith)
      Naming.lookup("rmi://torin.cs.lth.se/SimpleArith");
      System.out.println(simple.add(1.0, 2.0));
      System.out.println(simple.sqrt(2.0));
      System.out.println(simple.sqrt(-2.0));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

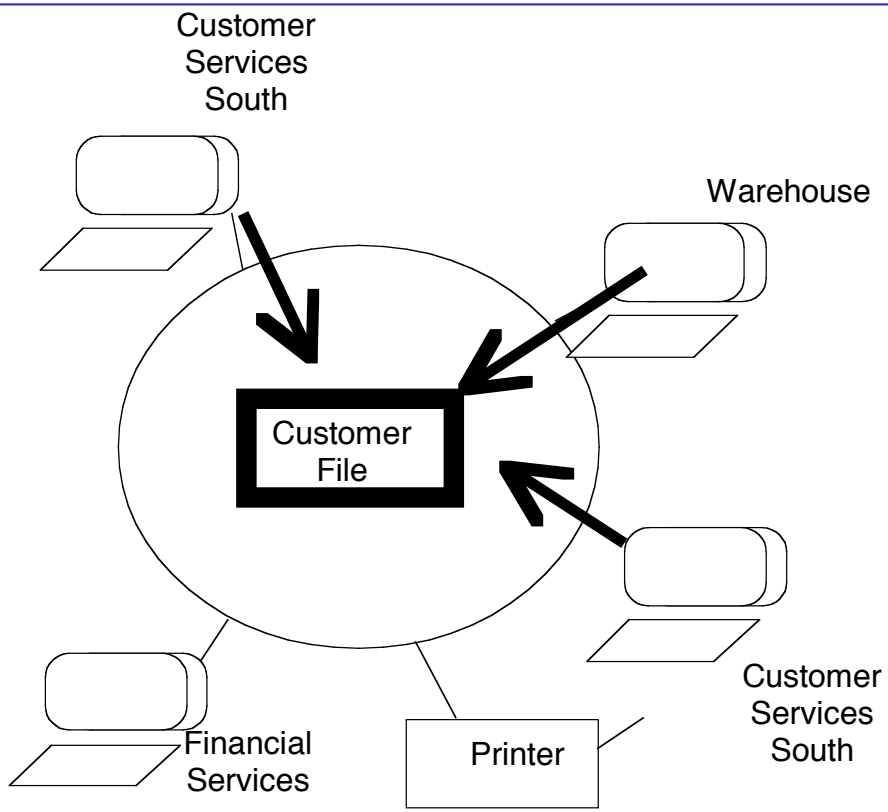
Running the Application

Files for the server class loader	Files for the client class loader
Remote service interface definitions	Remote service interface definitions
Remote service implementations	All other client classes
All other server classes	

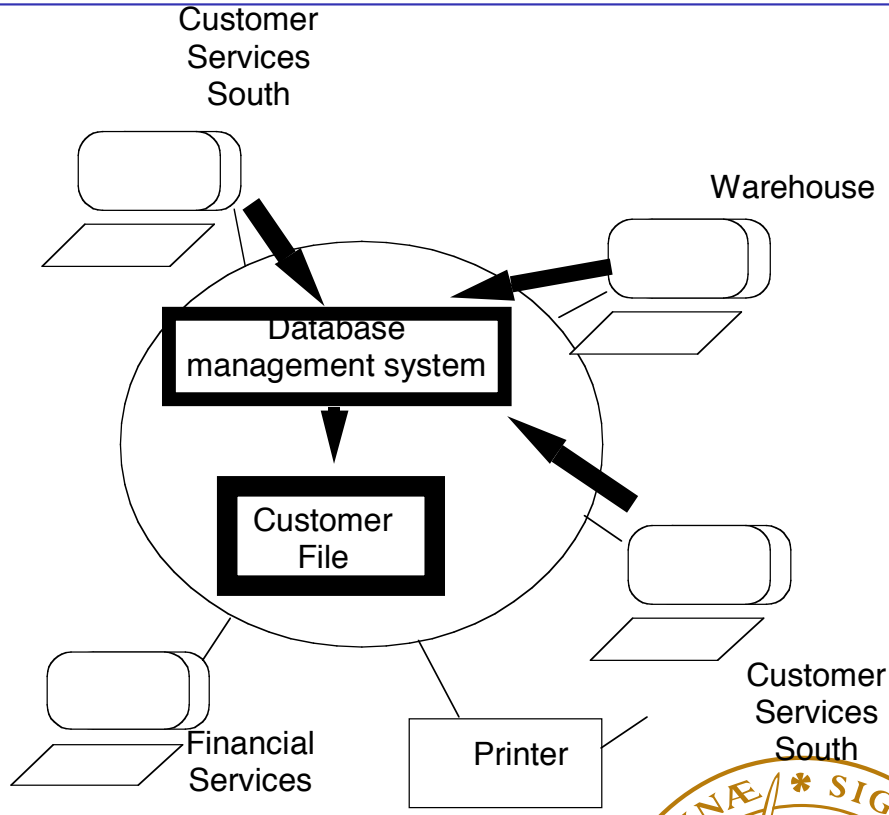
Start the registry on the server: `% rmiregistry`
Start the server: `% java SimpleArithServer`
Start the client: `% java SimpleArithClient`
(The files are in the RMI1 folder)

Distributed Applications

Sharing data

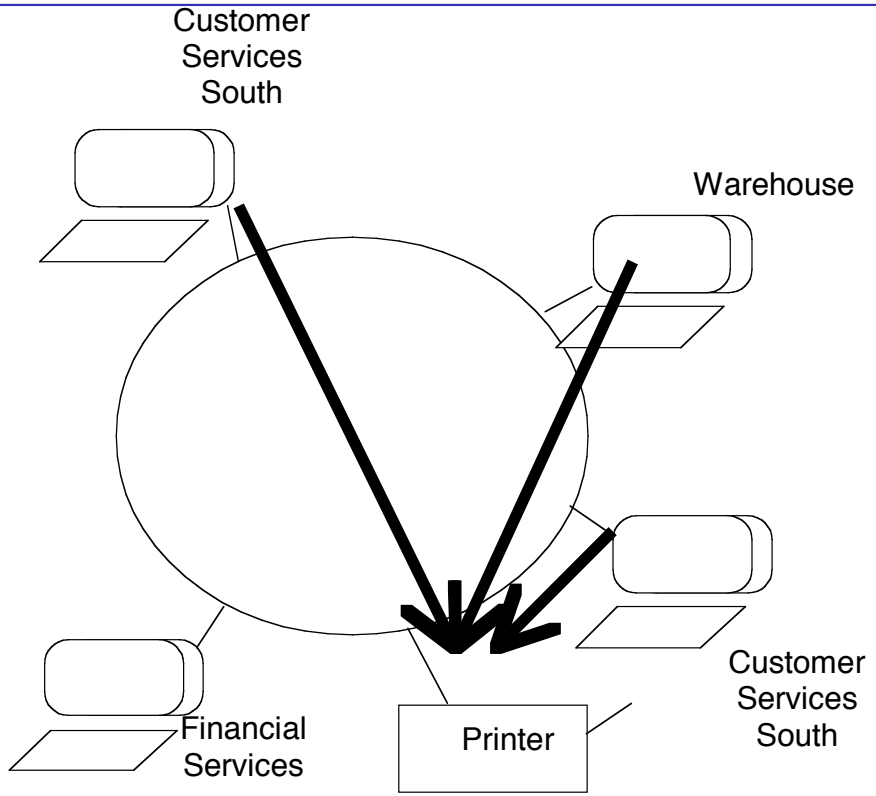


Sharing applications

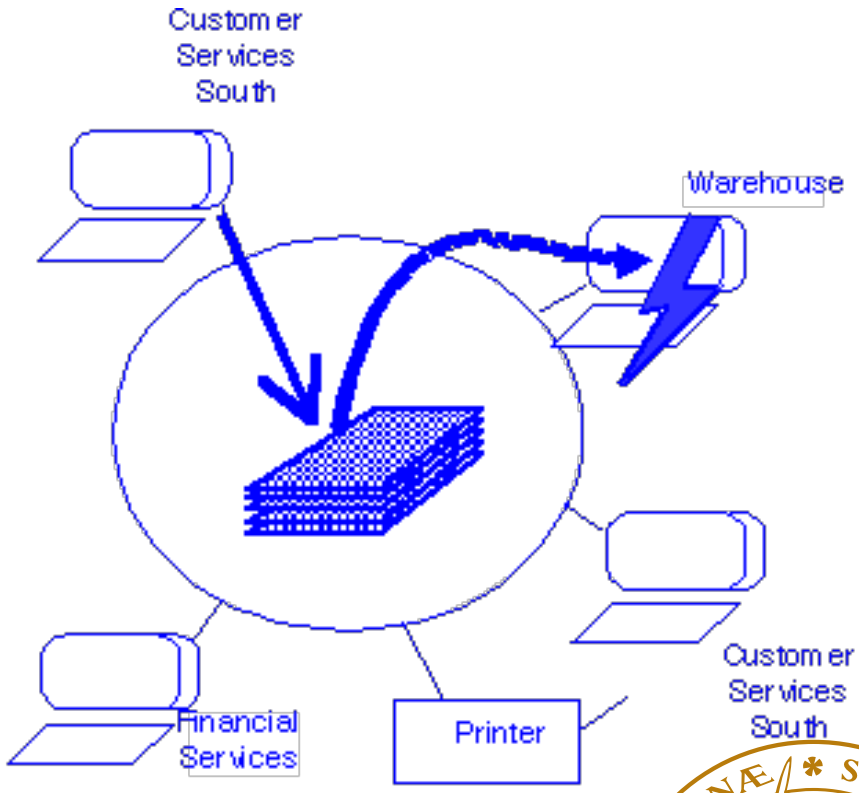


Distributed Applications (II)

Sharing resources

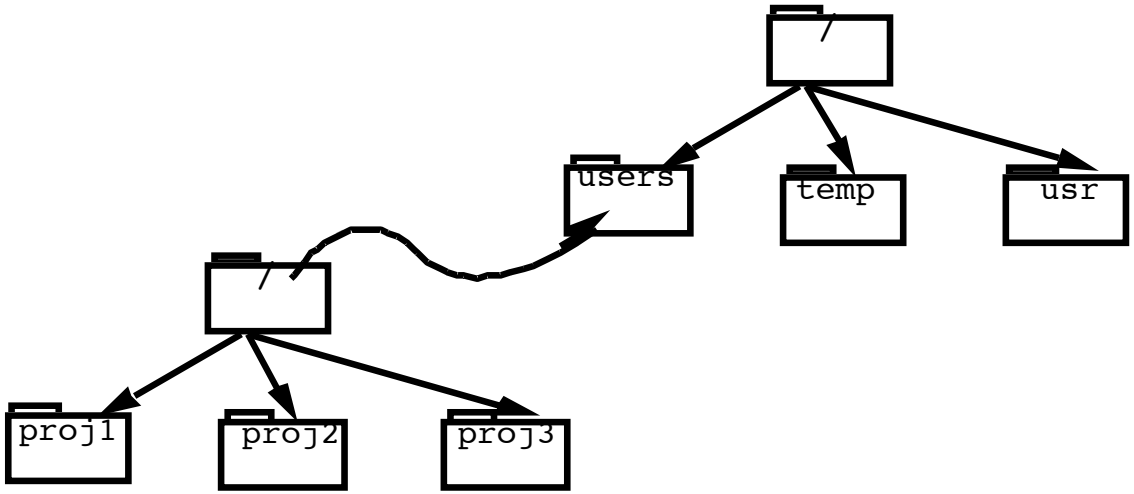


Sharing work

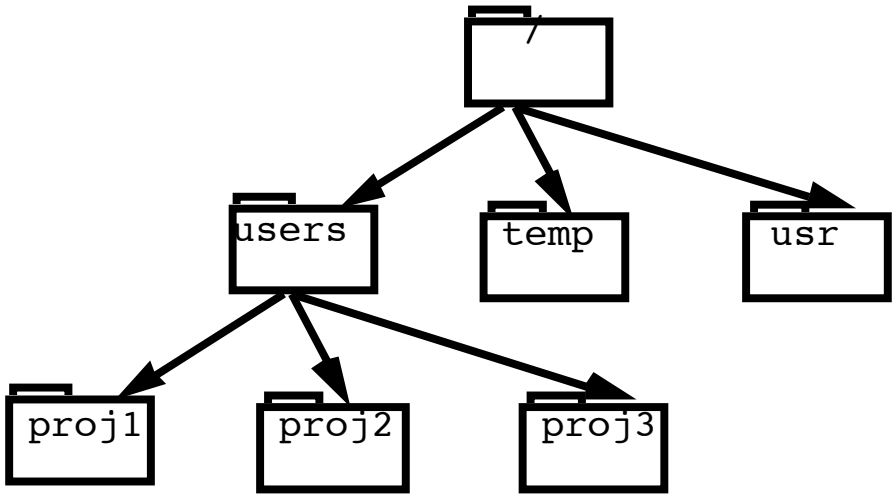


Mounting Files

Before



After



NFS: Mounting

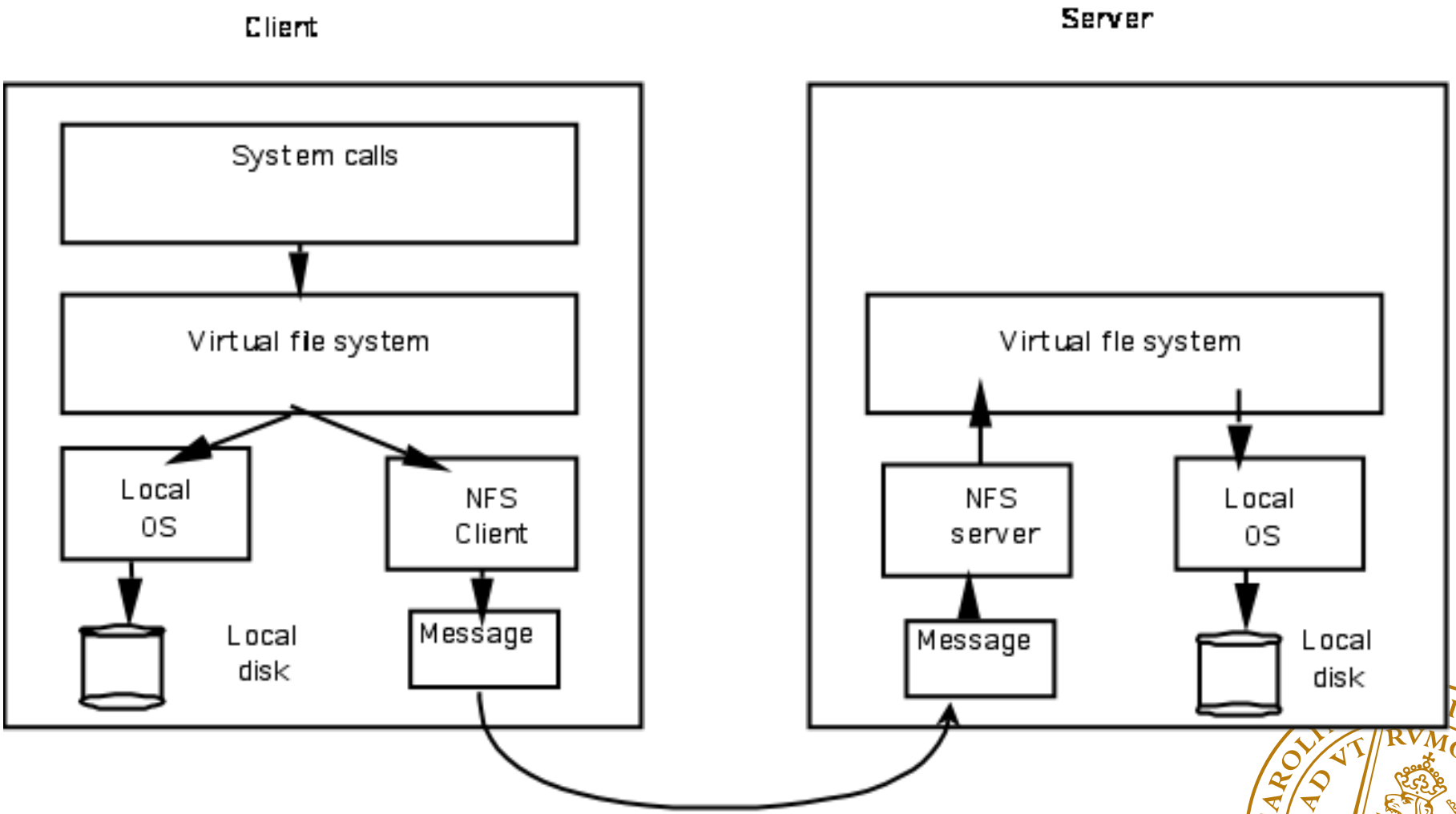
```
program MOUNTPROG {
  version MOUNTVERS {
    void MOUNTPROC_NULL(void) = 0;
    fhstatus MOUNTPROC_MNT(dirpath) = 1;
    mountlist MOUNTPROC_DUMP(void) = 2;
    void MOUNTPROC_UMNT(dirpath) = 3;
    void MOUNTPROC_UMNTALL(void) = 4;
    exportlist MOUNTPROC_EXPORT(void) = 5;
  } = 1;
} = 100005;
```

http://www.rfc-editor.org/in-notes/rfc1094.txt with many revisions

NFS: File Services

```
program NFS_PROGRAM {
  version NFS_VERSION {
    void NFSPROC_NULL(void) = 0;
    attrstat NFSPROC_GETATTR(fhandle) = 1;
    attrstat NFSPROC_SETATTR(sattrargs) = 2;
    void NFSPROC_ROOT(void) = 3;
    diropres NFSPROC_LOOKUP(diropargs) = 4;
    readlinkres NFSPROC_READLINK(fhandle) = 5;
    readres NFSPROC_READ(readargs) = 6;
    void NFSPROC_WRITECACHE(void) = 7;
    attrstat NFSPROC_WRITE(writeargs) = 8;
    diropres NFSPROC_CREATE(createargs) = 9;
    stat NFSPROC_REMOVE(diropargs) = 10;
    stat NFSPROC_RENAME(renameargs) = 11;
    stat NFSPROC_LINK(linkargs) = 12;
    stat NFSPROC_SYMLINK(symlinkargs) = 13;
    diropres NFSPROC_MKDIR(createargs) = 14;
    stat NFSPROC_RMDIR(diropargs) = 15;
    readdirres NFSPROC_READDIR(readdirargs) = 16;
    statfsres NFSPROC_STATFS(fhandle) = 17;
  } = 2;
} = 100003;
```

NFS: Implementation



File System Tools

- `df`: display the mounted and unmounted disk with the amount of occupied space
- `nfsstat`: nfs statistics, describes the remote procedures and the number of calls. Options `-c` (client), `-s` (server)
- `rpcinfo`, information on remote procedure calls running on a machine `-m` (rpcbind), `-s` (concise list), `-p machine` (rpcbind on `machine`)

Remote Objects

In the last example, the program transferred primitive types from a client to a server

As a natural part of Java, RMI can also transfer objects: data and code

The transferred code is executed by the server virtual machine

It is then possible to implement "mobile code" using RMIs

This resemble applets but in a more general architecture

The older RPC model does not contain this feature

Mobile code also raises new security challenges

Sending Objects

Instead of transmitting primitive data types, let us send objects:

```
public class Operands2 {
  private double a;
  private double b;
  Operands2(double a, double b) {
    this.a = a;
    this.b = b;
  }
  public double getFirst() {return a;}
  public double getSecond() {return b;}
}
```

The SimpleArith2 Interface

```
import java.rmi.*;

public interface SimpleArith2 extends Remote {
  public double add(Operands2 o) throws RemoteException;
  public double subtract(Operands2 o) throws RemoteException;
  public double sqrt(double a) throws RemoteException;
}
```

The Server

```
import java.rmi.*;
public class SimpleArithServer2 {
  public static void main(String [] args) {
    try {
      SimpleArith2 simple = new SimpleArith2Impl();
      Naming.rebind("SimpleArith2", simple);
      System.out.println("RMI server running");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The New Client

And the client is:

```
Operands2 o = new Operands2(1.0, 2.0);
System.out.println(simple.add(o));
System.out.println(simple.subtract(o));
System.out.println(simple.sqrt(2.0));
```

Will this work?
(The files are in the RMI2 folder)

More on Naming

- `rebind(String name, Remote obj)` rebinds a name to a new remote object
- `bind(String name, Remote obj)` binds a name to a remote object. It can't replace an old object as `rebind()`
- `unbind(String name)` unbinds a name
- `String [] list(String name)` returns a list of names in the name service from the URL

The server program normally does not return. In addition to the main thread, `rebind()` launches a second thread that blocks in the registry.

Moving Code

In this example, in addition to moving data, we will move code
This is a simplified example from a Java tutorial by Sun:
`http://java.sun.com/docs/books/tutorial/rmi/`
Let's use a `compute()` remote method that will designate some sort of generic computation
This `compute()` method will take an addition or a subtraction as parameter and move the code corresponding to two different classes
This is a toy example but it can be generalized to operations that are more realistic
The server will not know of the real classes. It will download them from the client and load them in its VM at runtime

The Declaration of the Remote Methods

An interface contains the declaration of the remote methods:

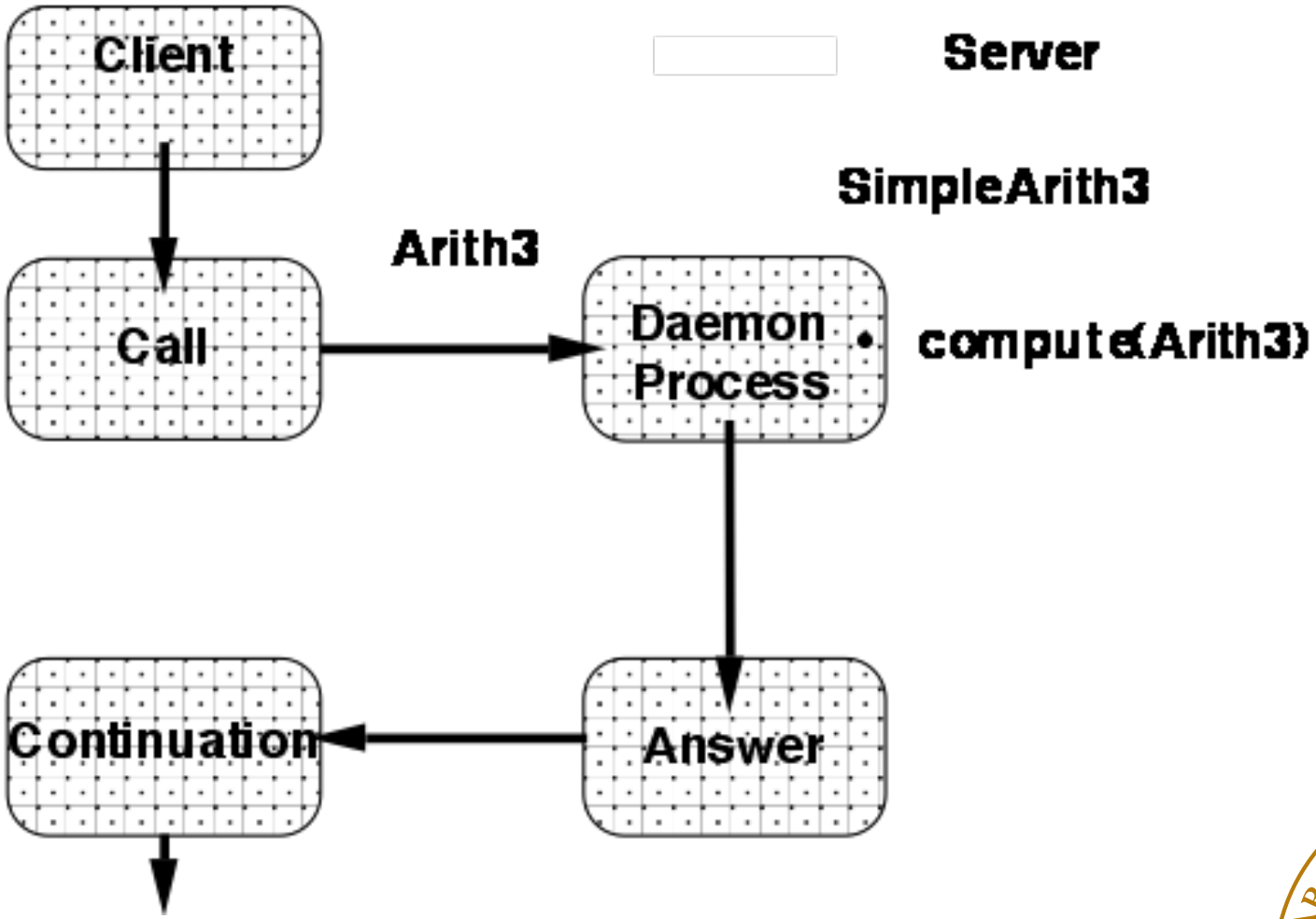
```
import java.rmi.*;

public interface SimpleArith3 extends Remote {
  public double compute(Arith3 o) throws RemoteException;
}
```

The interface uses a generic arithmetic operation `Arith3` that it passes to the server

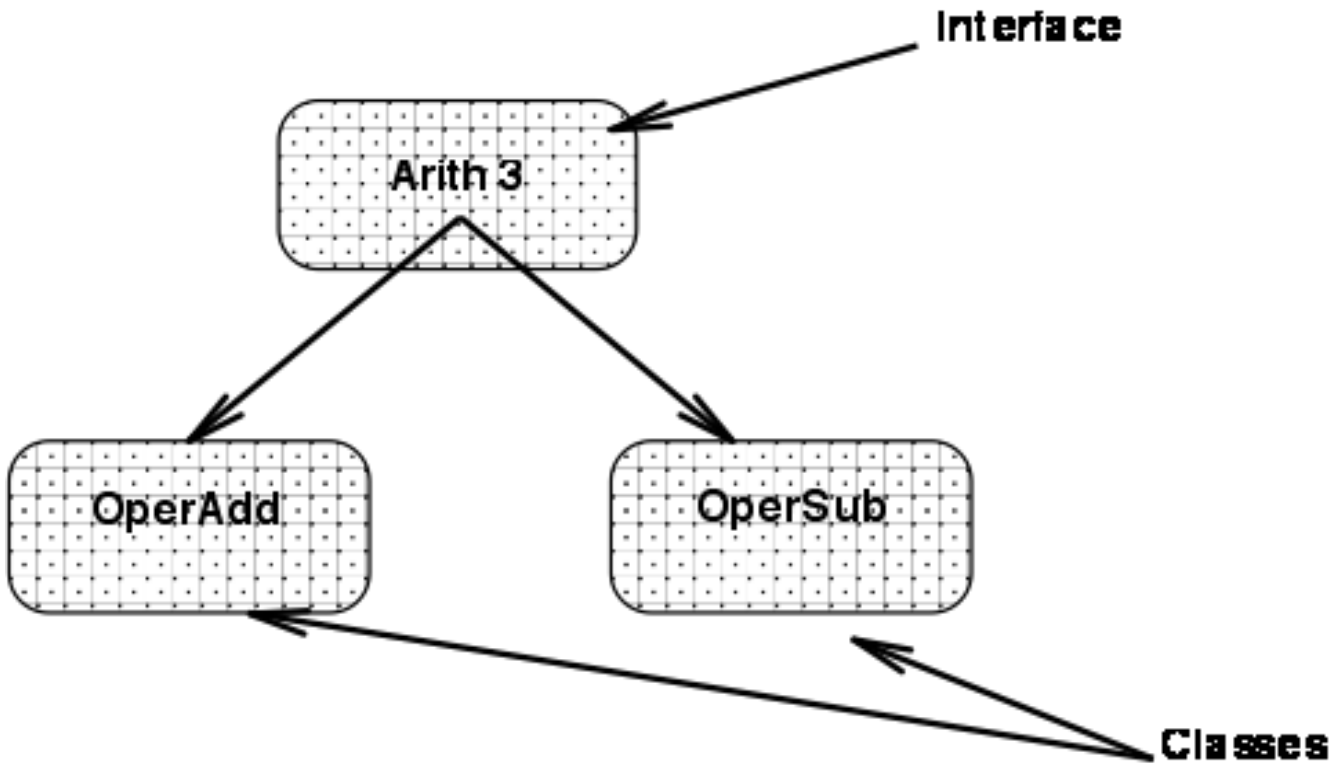
`Arith3` is the superclass of an addition or a subtraction

Data Exchange



The Real Objects

This object `Arith3` is the ancestor of real objects



The server only knows of the interface: `SimpleArith3`

The Arith3 Interface

The Arith3 interface describes the possible computations

```
import java.io.*;

public interface Arith3 extends Serializable {
  static final long serialVersionUID = 1L;
  double execute();
}
```

This execute() method could be anything.
It is implemented as an addition in the OperAdd class or a subtraction in the OperSub class

The OperAdd Class

```
import java.io.*;
public class OperAdd3 implements Arith3 {
  private double a;
  private double b;
  OperAdd3(double a, double b) {
    this.a = a;
    this.b = b;
  }
  public double getFirst() {return a;}
  public double getSecond() {return b;}
  public double execute() {
    return a + b;
  }
}
```

The Implementation of the Remote Method

The `compute()` remote method runs the `execute()` method of the `Arith3` interface in the server implementation.

```
import java.rmi.*;
import java.rmi.server.*;
public class SimpleArith3Impl
    extends UnicastRemoteObject implements SimpleArith3 {
  public SimpleArith3Impl() throws RemoteException {}
  public double compute(Arith3 arith) throws
      RemoteException {
    return arith.execute();
  }
}
```

The Server

The server is similar to what we have already seen

```
public class SimpleArithServer3 {
  public static void main(String [] args) {
    try {
      SimpleArith3 simple = new SimpleArith3Impl();
      Naming.rebind("SimpleArith3", simple);
      System.out.println("RMI server running");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The Client

The client invokes the compute() method in two different classes

```
public class SimpleArithClient3 {
  public static void main(String [] args) {
    try {
      SimpleArith3 simple = (SimpleArith3)
      Naming.lookup("rmi://pierre.cs.lth.se/SimpleArith3");
      OperAdd3 operAdd = new OperAdd3(1.0, 2.0);
      OperSub3 operSub = new OperSub3(1.0, 2.0);
      System.out.println(simple.compute(operAdd));
      System.out.println(simple.compute(operSub));
    } catch (Exception e) { e.printStackTrace(); }
  }
} //(The files are in the RMI3 folder)
```

Security

The client and the server share the code in our example: It is loaded locally
In the case of a distributed application, both the client and the server need to protect themselves against malicious code
If we remove the `OperAdd3` and `OperSub3` classes from the server, the code does not run anymore
Security of RMI is similar to that of applets
However, the RMI model is more flexible and also much more complex to tune

The Security Manager

Protection is enforced through a "Security Manager"
It controls the loading of external code
The next lines create and install a security manager

```
if (System.getSecurityManager() == null) {
  System.setSecurityManager(new RMISecurityManager());
  System.out.println("Security Manager installed");
}
```

By default, the protection is strict.
Be aware that the security model is different across versions of Java
(The files are in the RMI4 folder)

The Security Policy

The security model requires code being granted permissions
The protection policy is described in a file named `.java.policy`
You can create and edit such files using the `policytool` command
See `http://java.sun.com/javase/6/docs/technotes/tools/solaris/policytool.html`
Examples:

```
grant codeBase "file://Users/pierre/classes/-" {
  permission java.io.FilePermission "/Users/pierre/files/*",
    "read, write";
};

grant {
  permission java.net.SocketPermission "*:1024-65535",
    connect,accept";
};
```

The Security Files

The security files are specific to each machine
There is a default one:

```
${java.home}/jre/lib/security/.java.policy
```

A user one:

```
${user.home}/.java.policy
```

Otherwise, you must specify it when you launch Java

```
-Djava.security.policy=path/file
```

In our example, we will grant all the permissions:

```
grant {
  permission java.security.AllPermission;
};
```

Deploying and Running the Client and the Server

Deploying a RMI application can be much more difficult. It sometimes requires black magic skills.
A key point is to define properly where the client and the server load their Java code.
The code is loaded from a http or an ftp server. Let's start a http server on localhost and load the code (files in the `dynamic/client` folder from Pierre's homepage)
Let's run the server (files in `dynamic/engine`) and the client (files in `dynamic/client`) from Pierre's machine.
We need also the interface definitions in `dynamic/compute` on both the client and server

Parameters

We specify parameters for the client and the server using the `-D` option of the Java virtual machine

- `java.security.manager` tells to use a security manager if this has not been done in the program
- `java.security.policy` tells the files where the policy is described
- `rmi.server.codebase` tells where is the code. It must be a web or ftp server

Examples

Server:

```
java -Djava.rmi.server.codebase=http://localhost/~pierre/classes/
-Djava.rmi.server.hostname=localhost
dynamic.engine.SimpleArithServer4
```

Client:

```
java dynamic.client.SimpleArithClient4 localhost
```

A Word of Caution

Code base parameters can be difficult to adjust and the bugs are often hard to find

Tools can help deployment as WebLogic from Oracle (formerly BEA)

The page `http://java.sun.com/docs/books/tutorial/rmi/` contains a chapter on RMI deployment that details RMI parameter setting.