DISTRIBUTED OBJECTS AND REMOTE METHOD INVOCATION (RMI)

Network Distributed Systems

Dr. Christina Thorpe

OUTLINE

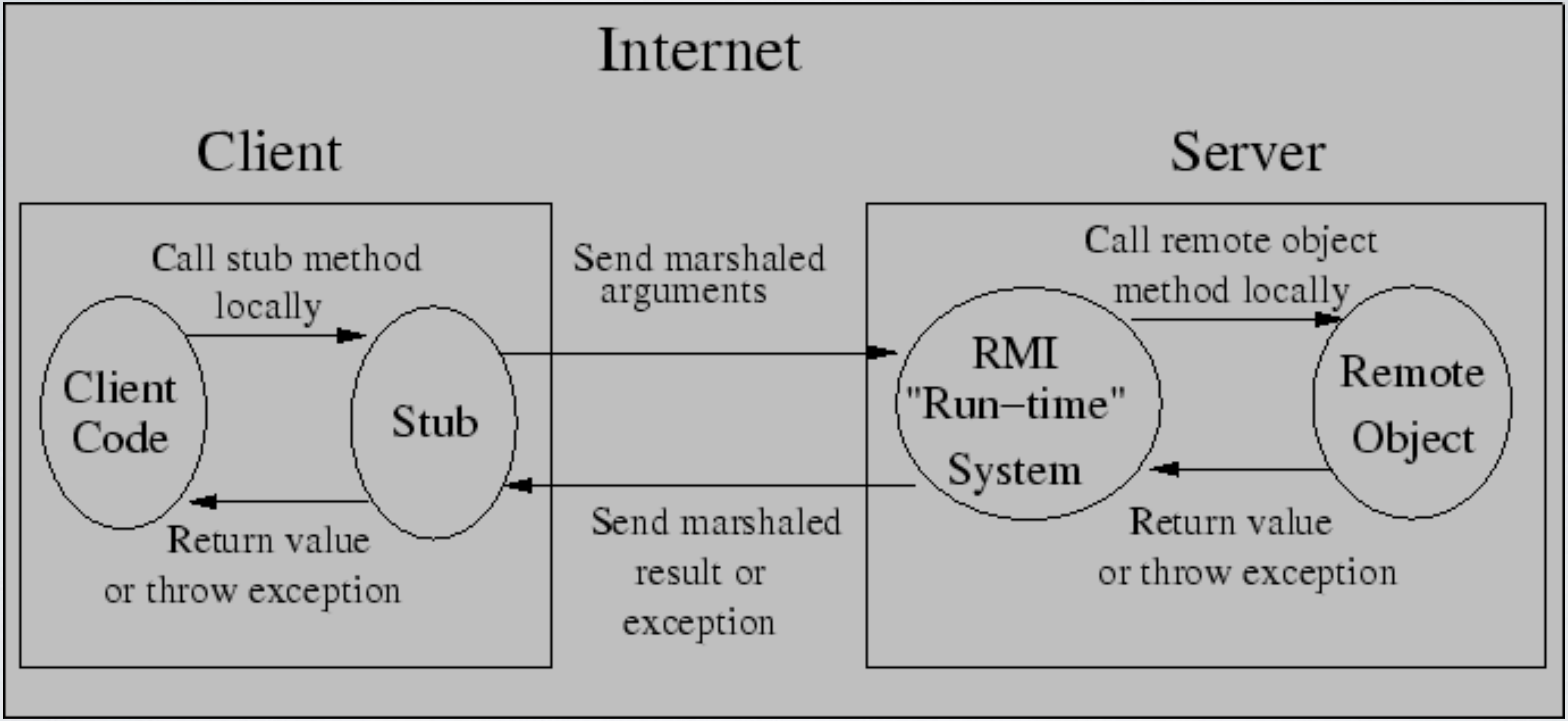
- Introduction to Distributed Objects and RMI
- RMI Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
- A more advanced RMI application
  - File Server

DISTRIBUTED OBJECTS

- A programming model based on Object-Oriented principles for distributed programming.
- Enables reuse of well-known programming abstractions (Objects, Interfaces, methods…), familiar languages (Java, C#...), and design principles and tools (design patterns, UML…)
- Each process contains a collection of objects, some of which can receive both remote and local invocations:
  - Method invocations between objects in different processes are known as remote method invocation, regardless the processes run in the same or different machines.
- Distributed objects may adopt a client-server architecture, but other architectural models can be applied as well.

JAVA RMI

- Java Remote Method Invocation (Java RMI) is an extension of the Java object model to support distributed objects
    - methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts
- Single-language system with a proprietary transport protocol (JRMP)
- RMI uses object serialization to marshal and unmarshal
    - Any serializable object can be used as parameter or method return


Internet
Client
Server
Call stub method locally
Client Code
Stub
Return value or throw exception
Send marshaled arguments
Send marshaled result or exception
Call remote object method locally
RMI "Run–time" System
Remote Object
Return value or throw exception

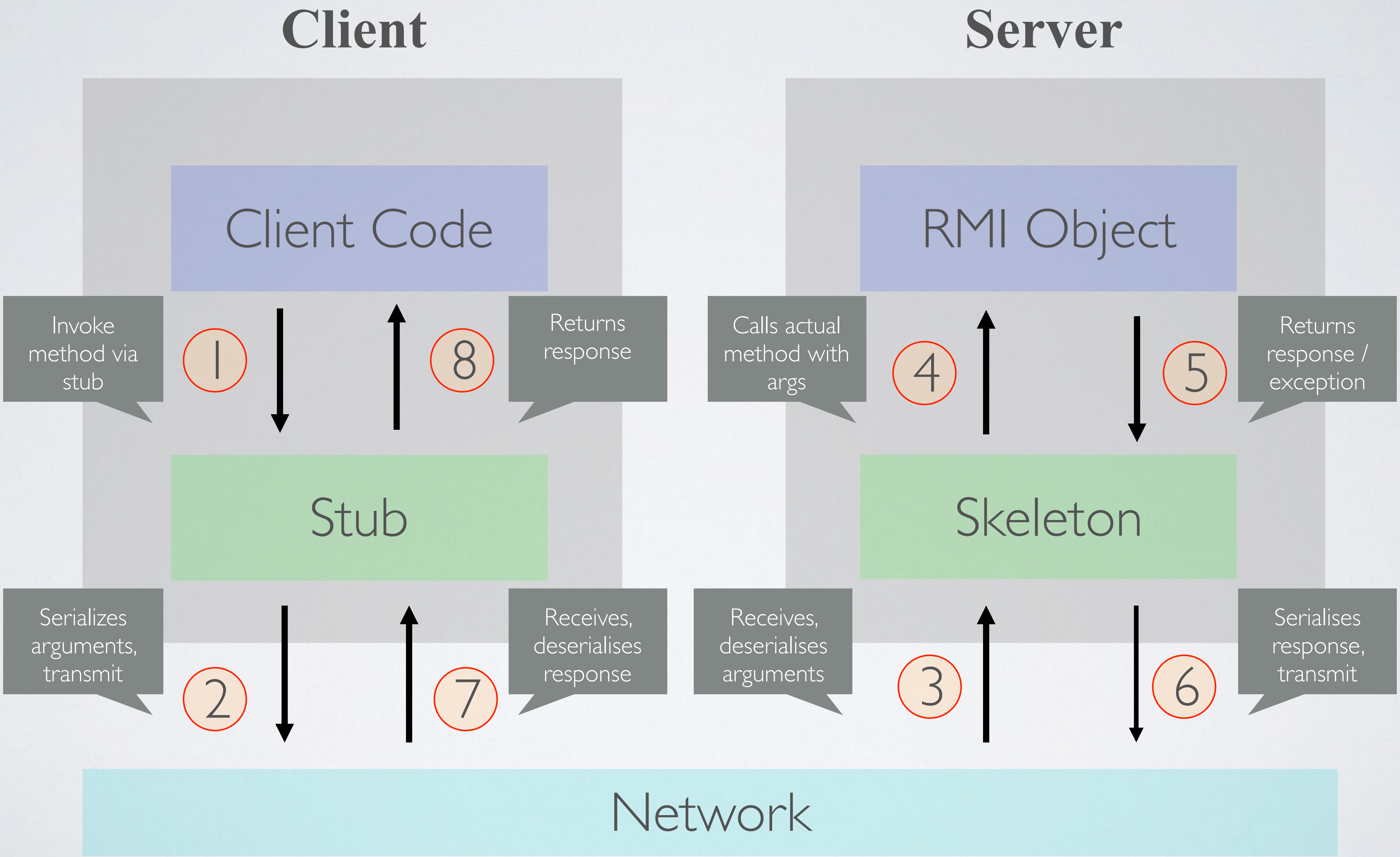
STUB OBJECT

- The tasks of stub object:
  - Building a block of information which consists of
    - an identifier of the remote object to be used,
    - an operation number describing the method to be called and
    - the marshalled parameters (method parameters have to be encoded into a format suitable for transporting them across the net).
- Sending the above information to the server

SKELETON

- The tasks of skeleton object:
  - Unmarshalling the parameters
  - Invoking the real object’s required method which is on the server
  - Capturing the value returned or exception returned by the invoked call on the server
  - Marshalling this value
  - Sending the package along with the value in the form of marshalled back to the stub on the client

INVOCATION LIFECYCLE



Client
Server
Client Code
Invoke method via stub
1
8
Returns response
Stub
Serializes arguments, transmit
2
7
Receives, deserialises response
RMI Object
Calls actual method with args
4
5
Returns response / exception
Skeleton
Receives, deserialises arguments
3
6
Serialises response, transmit
Network

STEPS OF IMPLEMENTING AN RMI APPLICATION

- Design and implement the components of your distributed application
  - Remote interface
  - Servant program
  - Server program
  - Client program
- Compile source and generate stubs
  - Client proxy stub
  - Server dispatcher and skeleton
- Make classes network accessible
  - Distribute the application on server side
- Start the application

RMI PROGRAMMING AND EXAMPLES

APPLICATION DESIGN

- Remote Interface
    - Exposes the set of methods and properties available
    - Defines the contract between the client and the server
    - Constitutes the root for both stub and skeleton
- Servant component
    - Represents the remote object (skeleton)
    - Implements the remote interface
- Server component
    - Main driver that makes available the servant
    - It usually registers with the naming service
- Client component

EXAMPLE APPLICATION – HELLO WORLD

- Server side
  - Create a HelloWorld interface
  - Implement HelloWorld interface with methods
  - Create a main method to register the HelloWorld service in the RMI Name Registry
  - Generate Stubs and Start RMI registry
  - Start Server
- Client side
  - Write a simple Client with main to lookup HelloWorld Service and invoke the methods

EXPLANATION: SERVER SIDE

- **Restrictions on the Remote interface**
  - User defined interface must extend java.rmi.Remote interface
  - Methods inside the remote interface must throw java.rmi.RemoteException
- **Servant class is recommended to extend** java.rmi.server.UnicastRemoteObject
  - Servant that does not extend UnicastRemoteObject need to export explicitly
    - UnicastRemoteObject.export(Remote remoteObj);
- **Name Service: RMI Registry**
  - Bind, unbind and rebind remote object to specified name
  - All the classes required need to be on rmiregistry classpath
  - java.rmi.Naming is an interface for binding object name (URL) to the registry

EXPLANATION: SERVER SIDE

- **Format of object URL**
    - //host:port/objectname (implicit declaration of RMI protocol for name)
        - Eg. //localhost/HelloWorldService or rmi://localhost/HelloWorldService
    - Default port is 1099
- For security reason, a server program can bind, unbind and rebind only on its same host
    - Prevent any binding from the remote clients
    - Lookup can be invoked everywhere
- After the binding, the remote object will not be reclaimed in garbage collection until the server unbinds it
    - Because it is remotely referred by the registry (running in a separate JVM)

SERVER SIDE : DEFINE AN INTERFACE (HELLOWORLD.JAVA)

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

public interface HelloWorld extends Remote {
    public String sayHello(String who) throws RemoteException;
}
```

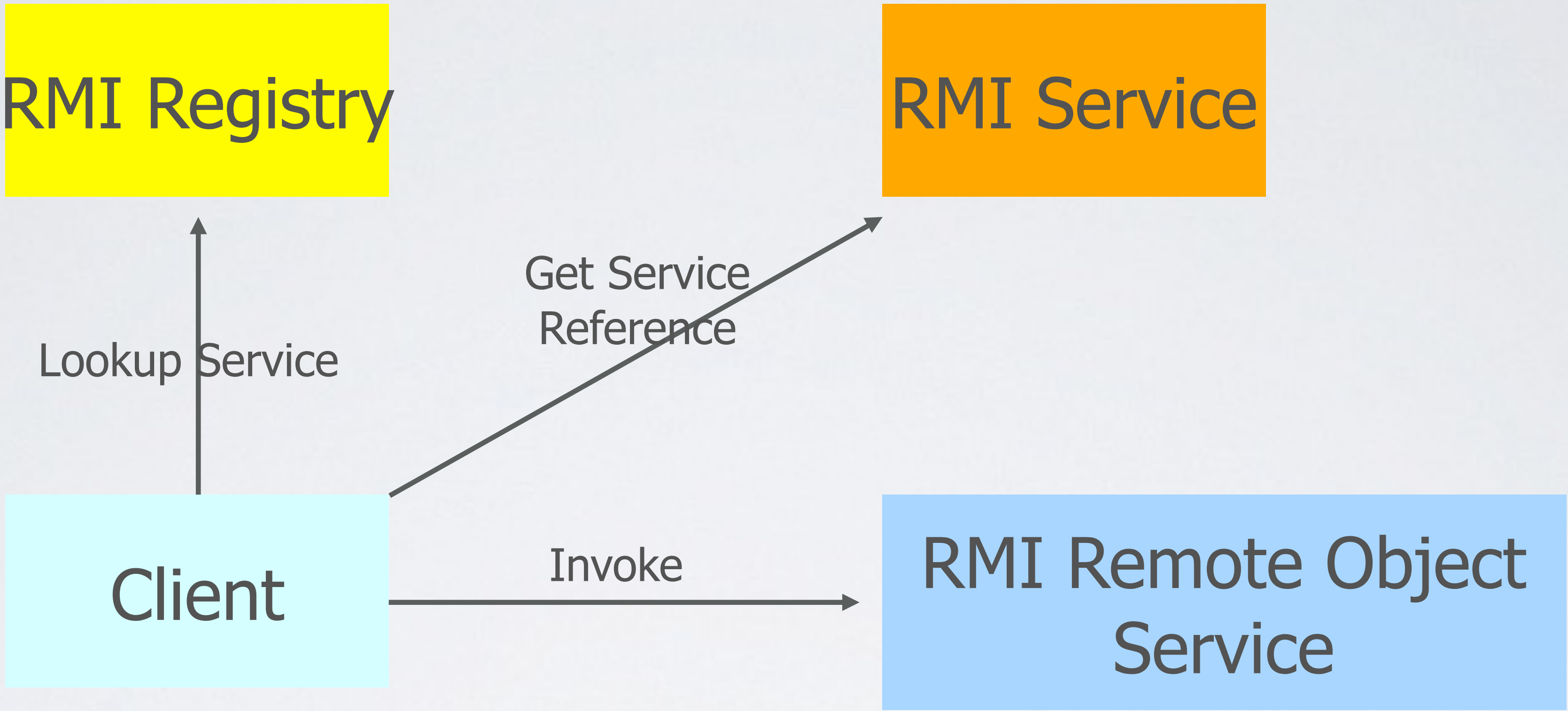
IMPLEMENTING A REMOTE INTERFACE(HELLOWORLDIMPL.JAVA)

```
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class HelloWorldImpl extends UnicastRemoteObject  implements HelloWorld {
	public HelloWorldImpl() throws RemoteException {
		super();
	}
	public String sayHello(String who) throws RemoteException {
		return "Hello "+who+" from your friend RMI 433-652 :-)";
	}
	public static void main(String[] args) {
		String hostName = "localhost";
		String serviceName = "HelloWorldService";
		if(args.length == 2){
			hostName = args[0];
			serviceName = args[1];
		}
		try{
			HelloWorld hello = new HelloWorldImpl();
			Naming.rebind("rmi://"+hostName+"/"+serviceName, hello);
			System.out.println("HelloWorld RMI Server is running...");
		}catch(Exception e){}
	}
}
```

GENERATE STUBS AND START RMI REGISTRY

- Compile
  - javac HelloWorld.java
  - javac HelloWorldImpl.java
- Generate Stubs (before Java 5)
  - rmic HelloWorldImpl
- Start RMI registry
  - start rmiregistry (windows)
  - rmiregistry 10000 & (unix)
- Start HelloWorld Server
  - java HelloWorldImpl
  - java HelloWorldImpl localhost:10000 HelloWorldService

EXPLANATION: CLIENT SIDE



- Look up the server registry
  - Based on the targeting object URL
- Get the remote object reference from the Registry
- Invoke the methods on the remote object reference

CLIENT SIDE : A SIMPLE CLIENT (RMICLIENT.JAVA)

```
import java.rmi.Naming;
public class RMIClient {
    public static void main(String[] args) {
      String hostName = "localhost";
      String serviceName = "HelloWorldService";
      String who = "Raj";
      if(args.length == 3){
          hostName = args[0];
          serviceName = args[1];
          who = args[2];
      }else if(args.length == 1){
          who = args[0];
      }
      try{
          HelloWorld hello = (HelloWorld)Naming.lookup("rmi://"+hostName+"/"+serviceName);
          System.out.println(hello.sayHello(who));
      }catch(Exception e){
          e.printStackTrace();
      }
    }
}
```

COMPILE AND RUN

- Compile
  - javac RMIClient.java
- Run it
  - java RMIClient
  - java RMIClient localhost HelloWorldService Raj
  - java RMIClient holly:10000 HelloWorldService Raj

SECURITY MANAGER

- Java's security framework
  - *java.security.-*
  - Permissions, Principle, Domain etc.
  - Security manager, for access control (file, socket, class load, remote code etc)
  - *$JAVA_HOME/jre/lib/security/java.policy*
- Use security manager in RMI
  - RMI recommends to install a security manager, or RMI may not work properly while encountering security constraints.
  - A security manager ensures that the operations performed by downloaded code go through a set of security checks.
    - Eg. Connect and accept ports for RMI socket and allowing code downloading

SECURITY MANAGER (CONT.)

- Two ways to declare security manager
    - Use System property java.security.manager
        - java –D java.security.manager HelloWorldImpl
    - Explicit declare in the source code

```
public static void main(String[]args){
   //check current security manager
   if(System.getSecurityManager()==null){
          System.setSecurityManager(new RMISecurityManager());
   }
   …
   //lookup remote object and invoke methods.
}
```

- Use customized policy file instead of java.policy
    - Usage
        - java -D java.security.manager -D java.security.policy=local.policy HelloWorldImpl

FILE: "LOCAL.POLICY" CONTENTS

Specific permissions:

```
grant {
    permission java.net.SocketPermission      "*:
   1024-65535","connect,accept";
    permission java.io.FilePermission  "/home/globus/RMITutorial/-", "read";
};
```

Grant all permissions:

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

EXCEPTIONS

- The only exception that could be thrown out is RemoteException
- All RMI remote methods have to throw this exception
- The embedded exceptions could be:
  - java.net.UnknownHostException or java.net.ConnectException: if the client can’t connect to the server using the given hostname. Server may not be running at the moment
  - java.rmi.UnmarshalException: if some classes not found. This may because the codebase has not been properly set
  - Java.security.AccessControlException: if the security policy file java.policy has not been properly configured

PASSING OBJECTS

- Restrictions on exchanging objects
    - Implementing java.io.serializable
    - All the fields in a serializable object must be also serializable
    - Primitives are serializable
    - System related features (e.g.Thread, File) are non-serializable
- How about the socket programming issues?
    - Where are sockets and corresponding input, output streams?
    - How to handle object passing?
    - Who does all the magic?

SAMPLE SCENARIO : FILE SERVER

Basic Requirements (File Server)

- Get the files from the File Server on specific directory
- Create new files on the File Server
- Create new directories on the File Server
- Delete files or directory on the File Server

Client

- A File Browser to explore the file system on the File Server supporting view, create and delete functionalities.

PROGRAMMING FILE SERVER

Create RMI remote interface

```
public interface FileServer extends Remote {
 public Map getFiles(String baseDir) throws RemoteException;
 public Map getFiles(String baseDir,boolean upper) throws RemoteException;
 public void createFile(String filename,boolean isDir) throws RemoteException;
 public void deleteFile(String filename) throws RemoteException;
}
```

Implement the Servant

FileServerImpl.java [FileServerImpl implements FileServer]

Implement the Server

```
 public class RMIFileServer{
public static void main(String [] args){
     try{
         String host = "localhost";
         if(args.length == 1){
                   host = args[0];
         }
         Naming.rebind("//"+host+"/FileServer",new FileServerImpl());
    }catch(Exception e){}
             }
     }
```

Generate Stub

rmic –keep rmi.server.FileServerImpl (keep option is to keep the generated java file)

PROGRAMMING FILE BROWSER

- Create a Subclass of JFrame
  - RMIFileBrowser.java
- Locate the FileServer Object

```
private FileServer findFileServer(String serviceURI)throws Exception{
    return (FileServer)Naming.lookup("rmi://"+serviceURI);
}
```

- Invoke the appropriate methods

```
Get Files :   Map result = getFileServer().getFiles(dir,upper);
    Create File: getFileServer().createFile(absoluteFilePath,false/*isDir=false*/);
    Create Dir:  getFileServer().createFile(absoluteFilePath,true/*isDir=true*/);
    Delete File: getFileServer().deleteFile(absoluteFilePath);
```

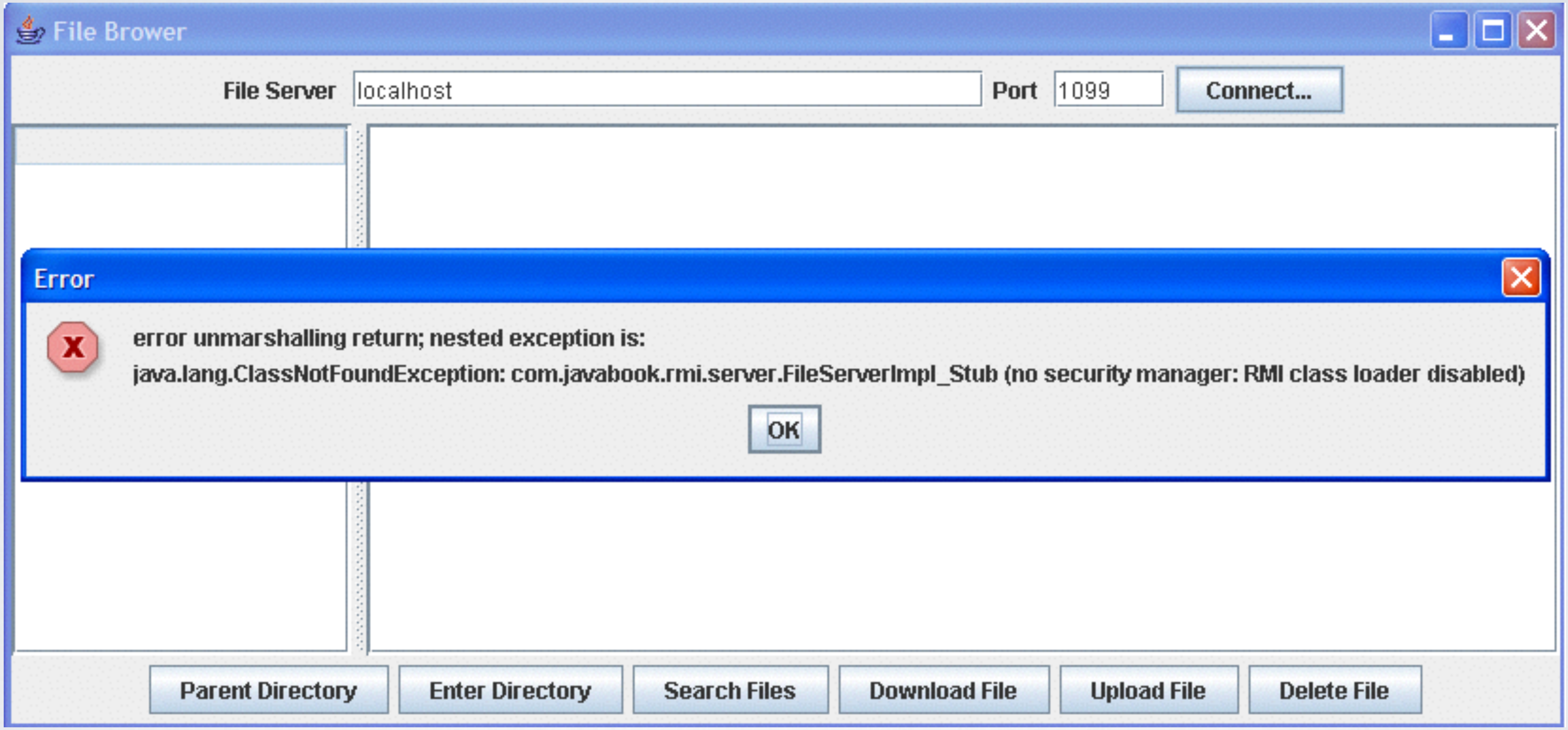
- Write a local policy file

```
Grant{
 java.security.AllPermission;
};
```

RUN IT!

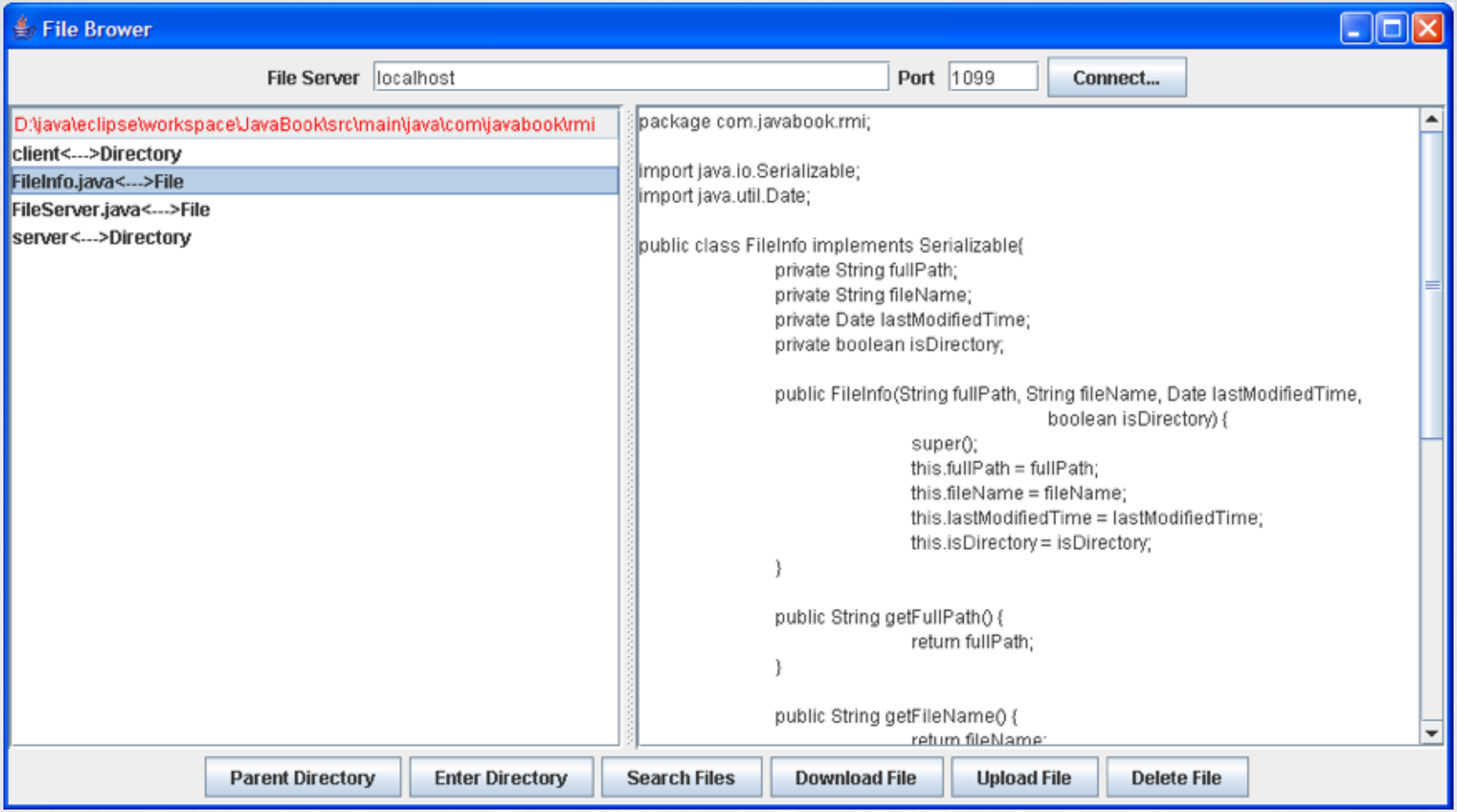
- Two steps
  - Start the server
    - start rmiregistry (be aware of your classpath)
    - java rmi.server.RMIFileServer
  - Start the client
    - java -D java.security.manager -D java.security.policy=local.policy demo.RMIFileBrowser
    - **Note: Required classes including the generated stubs must be visible by java classloader.**

POTENTIAL ERROR - NO SECURITY MANAGER!



error unmarshalling return; nested exception is:
java.lang.ClassNotFoundException: com.javabook.rmi.server.FileServerImpl_Stub (no security manager: RMI class loader disabled)

SUCCESSFUL EXECUTION



File Server localhost Port 1099

D:\java\eclipse\workspace\JavaBook\src\main\java\com\javabook\rmi

```
client<--->Directory
FileInfo.java<--->File
FileServer.java<--->File
server<--->Directory
```

```
package com.javabook.rmi;

import java.io.Serializable;
import java.util.Date;

public class FileInfo implements Serializable{
        private String fullPath;
        private String fileName;
        private Date lastModifiedTime;
        private boolean isDirectory;

        public FileInfo(String fullPath, String fileName, Date lastModifiedTime,
                        boolean isDirectory) {
                super();
                this.fullPath = fullPath;
                this.fileName = fileName;
                this.lastModifiedTime = lastModifiedTime;
                this.isDirectory = isDirectory;
        }

        public String getFullPath() {
                return fullPath;
        }

        public String getFileName() {
                return fileName;
```

SUMMARY: RMI PROGRAMMING

- RMI greatly simplifies creation of distributed applications (e.g., compare RMI code with socket-based apps)
- Server Side
  - Define interface that extend java.rmi.Remote
  - Servant class both implements the interface and extends java.rmi.server.UnicastRemoteObject
  - Register the remote object into RMI registry
  - Ensure both rmiregistry and the server is running
- Client Side
  - No restriction on client implementation, both thin and rich client can be used. (Console, Swing, or Web client such as servlet and JSP)