Lab Notes
Distributed Systems

Java Remote Method Invocation

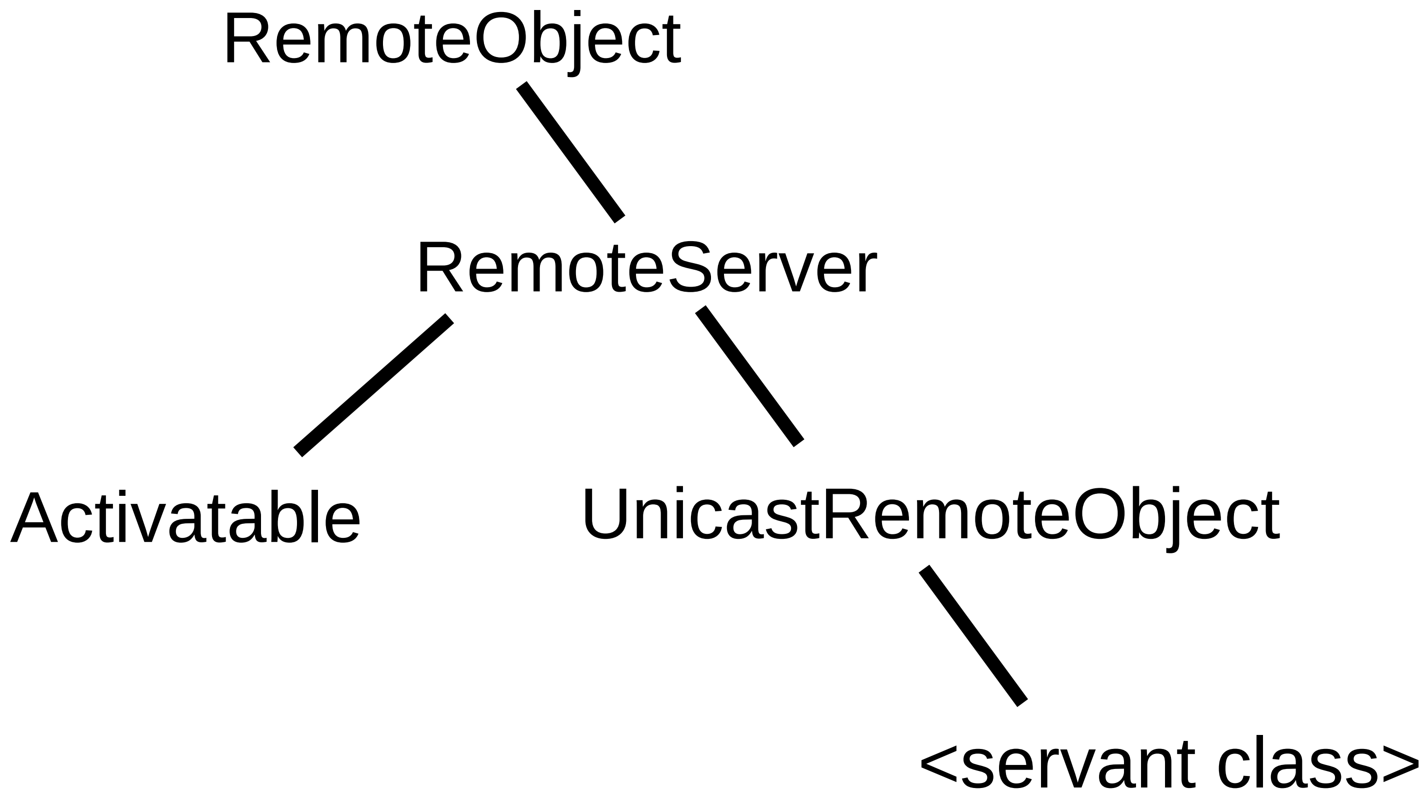
RMI

- RMI – a Java mechanism for calling methods of objects which do not run on the same Java Virtual Machine.

- Integral part of the Java language
- Always
  - One service provider – the *server;*
  - One service receiver – the *client*.

Java RMI

- Remote interfaces are defined in the Java language.
- Allows objects to invoke methods on remote objects using the same syntax as for local invocations.
- Type checking applies equally to remote invocations as to local ones.
- An object making a remote invocation is aware that its target is remote because it must handle *RemoteExceptions*.
- Behaviour in a concurrent environment must be considered.


RemoteObject
RemoteServer
Activatable
UnicastRemoteObject
<servant class>

Remote Method Invocation

- Remote Method Invocation (RMI) is an object-oriented implementation of the Remote Procedure Call model.
  - It is an API for Java programs only.

- Using RMI, an object server exports a remote object and registers it with a directory service. The object provides remote methods, which can be invoked in client programs.

- Syntactically:
  - A remote object is declared with a *remote interface*, an extension of the Java interface.
  - The remote interface is implemented by the *object server*.
  - An *object client* accesses the object by invoking the remote methods associated with the objects using syntax provided for remote method invocations.

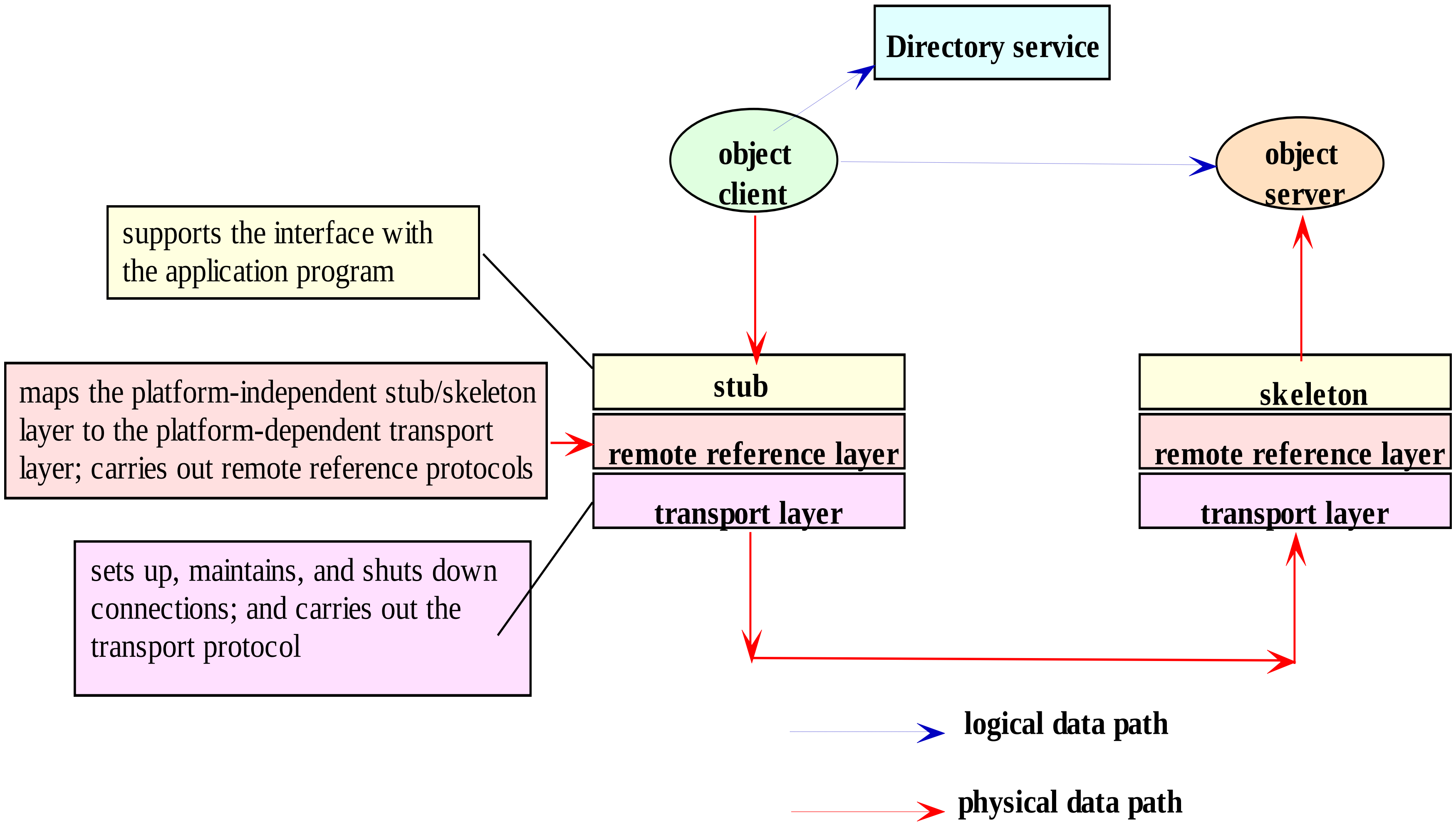
The Java RMI Architecture: Client-Side Architecture

1. The **Stub layer**
   - A client process's remote method invocation is directed to a proxy object, known as a **stub**. The stub layer lies beneath the application layer and serves to intercept remote method invocations made by the client program.
   - Then it forwards them to the next layer below, the Remote Reference Layer.
2. The **Remote Reference Layer**
   - Interprets and manages references made from clients to the remote service objects and issues the IPC operations to the next layer, the transport layer, to transmit the method calls to the remote host.
3. The **Transport layer**
   - TCP based – connection-oriented
   - Carries out the inter-process communication (IPC), transmitting the data representing the method call to the remote host.

The Java RMI Architecture: Server-side Architecture

1. The Skeleton layer
   – Lies just below the application layer and serves to interact with the stub layer on the client side.
2. The Remote Reference Layer
   – Manages and transforms the remote reference originating from the client to local references that are understandable to the Skeleton layer.
3. The Transport layer
   – Connection-oriented transport layer, i.e. TCP – in the TCP/IP architecture.

The Java RMI Architecture


Directory service
object client
object server
supports the interface with the application program
stub
skeleton
maps the platform-independent stub/skeleton layer to the platform-dependent transport layer; carries out remote reference protocols
remote reference layer
remote reference layer
transport layer
transport layer
sets up, maintains, and shuts down connections; and carries out the transport protocol
logical data path
physical data path

The Java RMI Architecture :Object Registry – Directory Service

- The RMI API allows a number of directory services to be used for registering a distributed object.
  - One such service is the Java Naming and Directory Interface (JNDI), which is more general than the RMI registry, in the sense that it can be used by applications that do not use the RMI API.
- We will use a simple directory service called the RMI registry, `rmiregistry`, which is provided with the Java Software Development Kit (SDK).
  - The RMI Registry is a service whose server, when active, runs on the object server's host machine, by convention and by default on the TCP port 1099.

The basic RMI process

- The server program controlling the remote objects registers an *interface* with a naming service – this interface then becomes accessible by client programs.
- The interface contains the signatures for those methods of the object that the server wishes to make publicly available. A client program can then use the same naming service to obtain a reference to this interface in the form of what is called a *stub*.
- *Stub*
  - A *local* surrogate for the remote object
- *Skeleton*
  - A surogate on the *remote* system

The basic RMI process (cont.)

- When the client program invokes a method on the remote object, it appears to the client as though the method is being invoked directly on the object.
- In reality – an equivalent method is being called in the stub. The stub then forwards the call and any parameters to the skeleton on the remote machine.
  - Only primitive types and those reference types that implement the `Serializable` interface may be used as parameters.
  - The serializing of these parameters - *marshalling*.
- The skeleton converts received byte stream into the original method call and associated parameters.
  - The deserialization of parameters - *unmarshalling*
- Skeleton then calls the implementation of the method.
- If the method has a return value, then the above process is reversed, with the return value being serialized on the server (by the skeleton) and deserialized on the client (by the stub)

The API for the Java RMI

- The Remote Interface
- The Server-side Software
  - The Remote Interface Implementation
  - Stub and Skeleton Generations
  - The Object Server
- The Client-side Software

The Remote Interface

- A Java interface is a class that serves as a template for other classes
  - it contains declarations or signatures of methods whose implementations are to be supplied by classes that implements the interface.

- A java remote interface is an interface that inherits from the Java `Remote` class, which allows the interface to be implemented using RMI syntax.
  - Other than the `Remote` extension and the `Remote` exception that must be specified with each method signature, a remote interface has the same syntax as a regular or local Java interface.

The Server-side Software

- An object server is an object that provides the methods of and the interface to a distributed object. Each object server must
  - implement each of the remote methods specified in the interface,
  - register an object which contains the implementation with a directory service.
- It is recommended that the two parts be provided as separate classes.

The Client-side Software

- The program for the client class is like any other Java class.
- The syntax needed for RMI involves
  - locating the RMI Registry in the server host, and
  - looking up the remote reference for the server object; the reference can then be cast to the remote interface class and the remote methods invoked.

Implementation details

Packages

- `java.rmi`
- `java.rmi.server`
- `java.rmi.registry`

The steps:

1. Create the interface.
2. Define a class that implements this interface.
3. Create the server process.
4. Create the client process.

Example application - Hello

- Displays a greeting to any client that uses the appropriate interface registered with the naming service to invoke the associated method implementation on the server.

Create the interface

- Should import package `java.rmi`
- Must extend interface `Remote` – an interface that contains no methods.
- `Hello` example:
  - Must specify the signature for method *`getGreeting`*, that is to be made available to clients – this method must declare that it throws a `RemoteException`.

```
import java.rmi.*;

public interface Hello extends Remote {
   public String getGreeting() throws RemoteException;
}
```

Define a class that implements this interface.

- Import `java.rmi` and `java.rmi.server` packages.
- Extend class `RemoteObject` or one of `RemoteObject`’s subclasses.
  - Most implementation extend subclass `UnicastRemoteObject` – as this class supports point-to-point communication using TCP streams.
- Implement our interface `Hello`, by providing an executable body for the single interface method `getGreeting.`
- Must provide a constructor for the implementation object.
  - The constructor must declare that it throws a *RemoteException*.
- Common convention:
  - Append `Impl` onto the name of the interface to form the name of the implementing class.

Implementation of RMI interface.

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

public class HelloImpl extends UnicastRemoteObject implements Hello
{

   public HelloImpl() throws RemoteException {
   //No action needed here (but must specify constructor).
   }

   public String getGreeting() throws RemoteException {
        return ("Hello there!");
   }
}
```

Create the server process

- The server creates object(s) of the implementation class and registers them with a naming service called the *registry*.
  - Uses method `rebind` of class `Naming(java.rmi)`: this method establishes a connection between the object's name and its reference; clients can then use the remote object's name to retreive a reference to that object via the registry.
- `rebind` method arguments:
  - A `String` that holds the name of the remote object as a URL with protocol *rmi;*
    - Apart from specifying a protocol of RMI and a name for the object, the URL string specifies the name of the remote object's host machine; RMI assumes *localhost* by default; RMI default port is 1099 – which can be changed.
  - A reference to the remote object (as an argument of type `Remote`)

Server code

```
import java.rmi.*;

public class HelloServer {

   private static final String HOST = "localhost";

   public static void main(String[] args) throws Exception
   {
       HelloImpl temp = new HelloImpl();

       String rmiObjectName = "rmi://" + HOST + "/Hello";
       //Could omit host name, since 'localhost' would be
       //assumed by default.
       Naming.rebind(rmiObjectName,temp);

       System.out.println("Binding complete...\n");
   }
}
```

Create the client process

- Client obtains a reference to the remote object from the registry.
- It uses method `lookup` of class `Naming`, Supplying as an argument the same URL that the server did when binding the object reference to the object's name in the registry.
- As lookup returns a `Remote` reference, this reference must be typecast into a *Hello* reference.
- Once the Hello reference has been obtained, it can be used to call the solitary method that was made available in the interface.

Client code

```
import java.rmi.*;
public class HelloClient
{
   private static final String HOST = "localhost";

   public static void main(String[] args)
   {
       try{
         Hello greeting = (Hello)Naming.lookup("rmi://" + HOST + "/Hello");

         //Simply retrieve and display greeting...
         System.out.println("Message received: " + greeting.getGreeting());
        }
       catch(ConnectException conEx){
         System.out.println("Unable to connect to server!");
         System.exit(1);
       }
       catch(Exception e){
         e.printStackTrace();
          System.exit(1);
       }
   }
}
```

Compilation and execution

1. Compile all files with `javac`.

Note: Before Java SE 5 it was necessary to ompile the implementation class with the `rmic` compiler:

- Supplied with the JDK
- Operates on the .class file generated in the step 1.
- Used without any command line option, it will generate both a stub file and a skeleton file.

➢ `rmic HelloImpl`

Compilation and execution (cont.)

2. Start the RMI registry

```
> rmiregistry
```

3. Open a new window and run the server. From the new window, invoke the Java interpreter:

```
> java HelloServer
```

4. Open a third window and run the client.

```
> java HelloClient
```

Using RMI meaningfully

- Multiple methods and multiple objects normally employed. Two possible strategies may be adopted:
    1. Use a single instance of the implementation class to hold instance()s) of a class whose methods are to be called remotely. Pass instance(s) of the latter class as argument(s) of the constructor for the implementation class.
    2. Use the implementation class directly for storing required data and methods, creating instances of *this* class, rather than using separate class(es).

Method 2 approach

1. Interface - Bank.java – methods declared, each throws a *RemoteException*

```
import java.rmi.*;
public interface Bank extends Remote {

     public int getAcctNum() throws RemoteException;

     public String getName() throws RemoteException;

     public double getBalance() throws RemoteException;

     public double withdraw(double amount) throws
     RemoteException;

     public void deposit(double amount) throws RemoteException;
}
```

Method 2 - Bank example

2. Implementation class: (BankImpl.java):
    - As well as holding the data and method implementations associated with an individual account, this class defines a constructor for implemention objects.

```
import java.rmi.*;
import java.rmi.server.*;
public class BankImpl extends UnicastRemoteObject implements Bank {
      private int acctNum;
      private String surname;
      private String firstNames;
      private double balance;
      public BankImpl (int acctNo, String sname, String fnames, double bal) throws RemoteException {
                acctNum = acctNo;
                surname = sname;
                firstNames = fnames;
                balance = bal;
      }
      public int getAcctNum() throws RemoteException {
                return acctNum;
      }
      public String getName() throws RemoteException {
                return (firstNames + " " + surname);
      }
      public double getBalance() throws RemoteException {
                return balance;
      }
      public double withdraw(double amount) throws RemoteException {
                if (amount <= balance) return amount;
                else return 0;
      }
      public void deposit(double amount) throws RemoteException {
                if (amount > 0) balance += amount;
      }
}
```

Method 2 – Bank example

3. Create the server process (BankServer.java):

   - The server class creates an array of implementation objects and binds each one individually to the registry. The name used for each object will be formed from concatenating the associated account number onto the word ‘Account’ (forming ‘Account111111’, etc..

BankServer.java

```
import java.rmi.*;
public class BankServer {

   private static final String HOST = "localhost";

   public static void main(String[] args) throws Exception {
   BankImpl[] account =
        {new BankImpl(111111, "Smith", "Fred James", 112.58),
        new BankImpl(222222, "Jones", "Sally", 507.85),
        new BankImpl(234567, "White", "Mary Jane", 2345.00),
        new BankImpl(666666, "Satan", "Beelzebub", 666.00)};

    for (int i=0; i<account.length; i++) {
      int acctNum = account[i].getAcctNum();
   Naming.rebind("//" + HOST + "/account" + acctNum, account[i]);
   }
   System.out.println("Binding complete...\n");
   }
}
```

Method 2 – Bank example

4. Create the client process (BankClient.java)

   - The client again uses method lookup, this time to obtain references to individual accounts (held in separate implementation objects)

BankClient.java

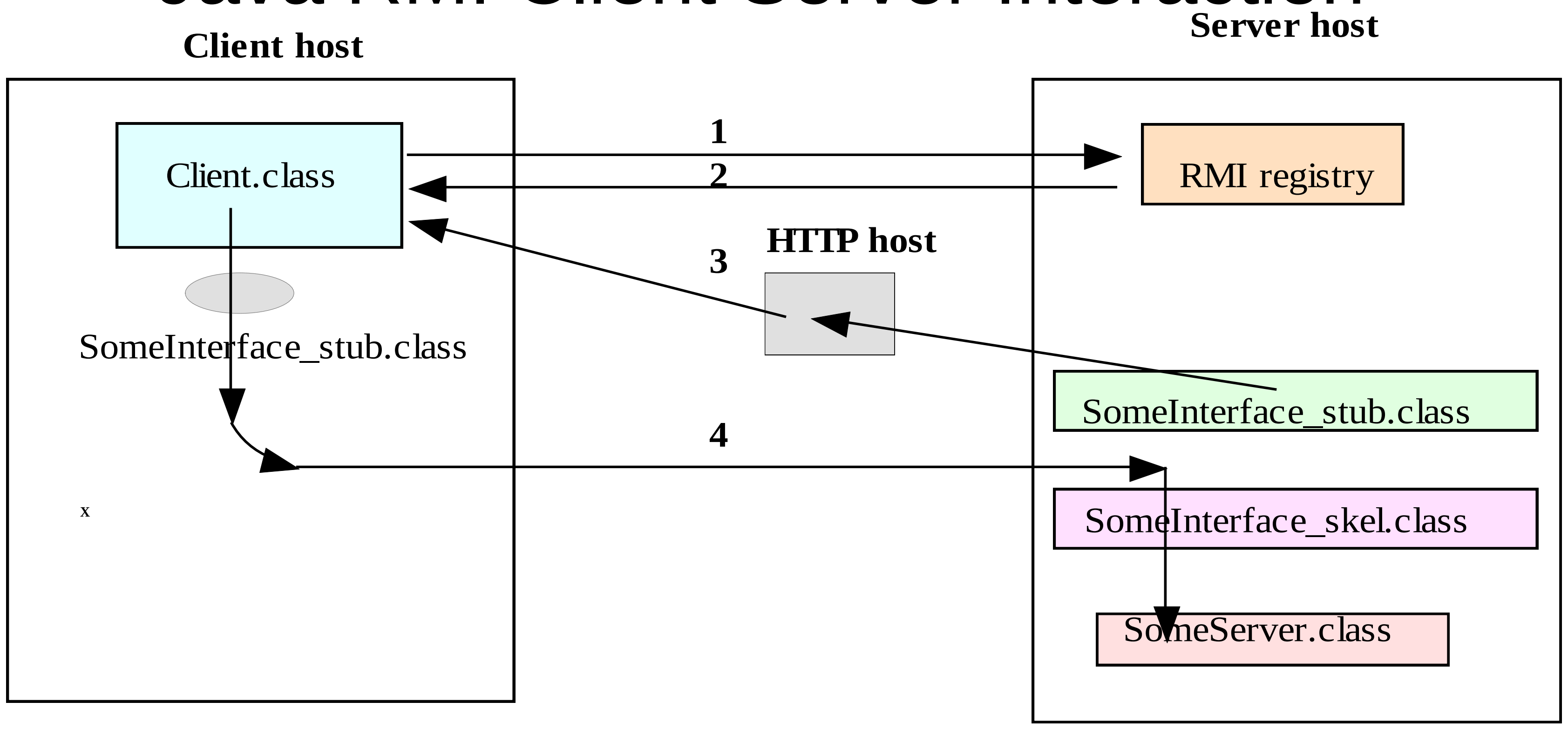
```
import java.rmi.*;

public class BankClient {

   private static final String HOST = "localhost";
   private static final int[] acctNum = {111111, 222222, 234567, 666666};

   public static void main(String[] args) {
     try {
      for (int i=0; i<acctNum.length; i++) {
        Bank temp=(Bank)Naming.lookup("rmi://"+HOST + "/account"+ acctNum[i]);
        System.out.println("\nAccount number: " + temp.getAcctNum());
        System.out.println("Name: " + temp.getName());
        System.out.println("Balance: " + temp.getBalance());
       }
      } catch(ConnectException conEx) {
        System.out.println("Unable to connect to server!");
        System.exit(1);
      } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
      }
   }
}
```

Java RMI Client Server Interaction



1. **Client looks up the interface object in the RMIregistry on the server host.**
2. **The RMIRegistry returns a remote reference to the interface object.**
3. **If the interface object's stub is not on the client host and if it is so arranged by the server, the stub is downloaded from an HTTP server.**
4. **Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the server object.**

RMI Stub Downloading

- RMI is designed to allow stubs to be made available to the client dynamically. Doing so allows changes to be made in the remote methods without affecting the client program.
- The stub can be filed with an web server and be downloaded using HTTP.
- Security measures are needed to prevent both the client side and the server side:
  - A java security policy file needs to be set on the server host and also on the client host.
  - A Java Security Manager should be instantiated in both the client and server programs.

RMI Security

- An application receiving an object for which it does *not* have the corresponding class file can try to load that class file from a remote location and instantiate the object in its JVM.
- An object passed as an RMI argument from such a remote source can attempt to initiate execution on the client's machine immediately upon deserialization – without the user/programmer doing anything with it. Such a breach of security is not permitted to occur.
- The loading of this file is handled by an object of class `SecureClassLoader`, which `must` have security restrictions defined for it.
  - `java.policy` defines security restrictions,
  - `java.security` defines the security properties.
- Implementation of the security policy is controlled by an object of class `RMISecurityManager` (a subclass of `SecurityManager`)

RMI Security (Cont)

- A default RMISecurityManager relies on the system’s default security policy – far too restrictive to permit the downloading of class files from a remote site.
- Therefore, we must create our own security manager that extends `RMISecurityManager`. This security manager must provide a definition for method `checkPermission`

Security manager sample

- Allows everything

```
import java.rmi.*;
import java.security.*;

public class ZeroSecurityManager extends
  RMISecurityManager
{
   public void checkPermission(Permission permission)
   {
       System.out.println("checkPermission for : " +
                                    permission.toString());
   }
}
```

Security manager handling

- Must be compiled with `javac`
- The client program must install an object of this class by invoking method `setSecurityManager`, which is a static method of class System – that takes a single argument of class `SecurityManager`.
- Next slide: `HelloClient` program, incorporating a call to `setSecurityManager`.

Secure HelloClient

```
import java.rmi.*;
public class HelloClient
{
   private static final String HOST = "localhost";
   public static void main(String[] args){
   //Here's the new code...
   if (System.getSecurityManager() == null){
      System.setSecurityManager( new ZeroSecurityManager());
    }

    try{
      Hello greeting = (Hello)Naming.lookup("rmi://" + HOST + "/Hello");
      System.out.println("Message received: "greeting.getGreeting());
   }
   catch(ConnectException conEx){
      System.out.println("Unable to connect to server!");
      System.exit(1);
   }
   catch(Exception e){
      e.printStackTrace();
      System.exit(1);
   }
   }
}
```

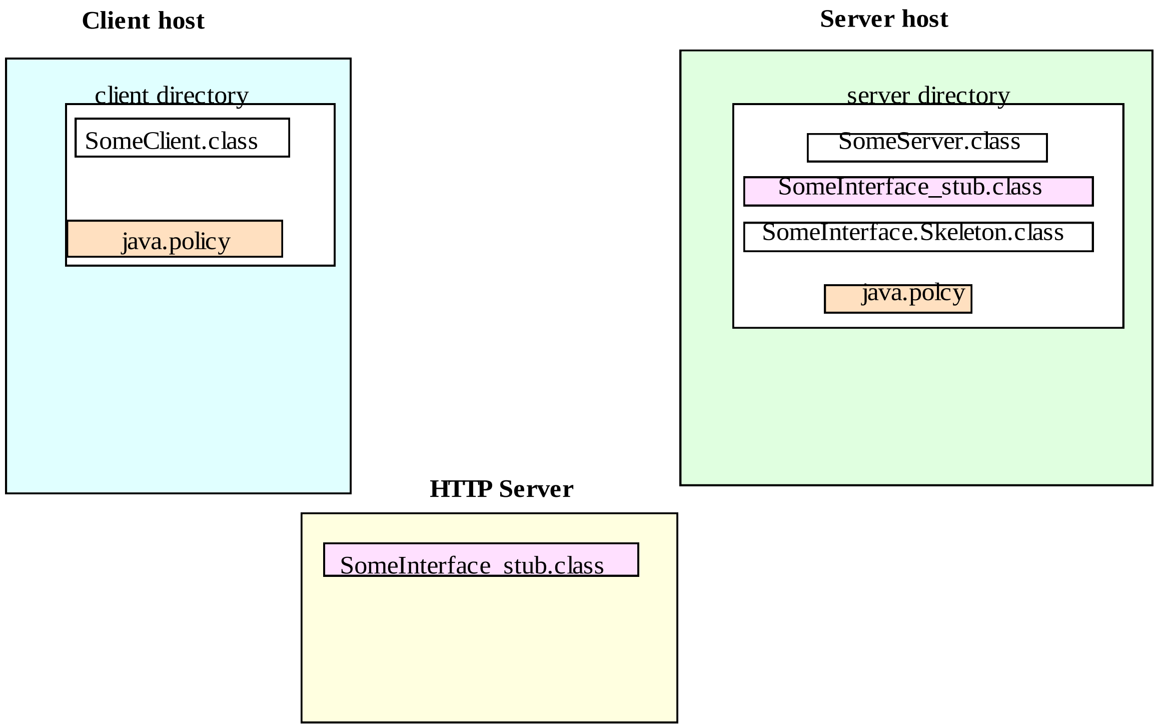
Server side options

- Need to specify where the required .class files are located, so that clients may download them:
  - At the same time that the server is started, set the `java.rmi.server.codebase` property to the URL of this location.
- E.g., if the URL of the file location is http://java.shu.ac.uk/rmi/, then the following line would set our `HelloServer` program running and would set the codebase property to the required location at the same time (in one line)

```
java -Djava.rmi.server.codebase
  =http://java.shu.ac.uk/rmi/HelloServer
```

Dynamic class loading

- Dynamic class loading is the facility that allows Java to look at runtime for a class that it requires - unlike other programming languages where all the code is linked into an executable following compilation.
- Java can load classes from anywhere - for example from any part of your class path on your local machine or your network. It can also load classes over the web.
- If the server implementation is changed and a new stub class is generated the stub class will have to be distributed to all clients. This is undesirable. A better approach would be to make the new stub class available online and whenever a client starts up it automatically loads the new class from the web server.



Advantages of Dynamic Code Loading

- RMI has ability to download the bytecodes (or simply code) of an object's class if the class is not defined in the receiver's virtual machine.
- The types and the behavior of an object, previously available only in a single virtual machine, can be transmitted to another, possibly remote, virtual machine.
- RMI passes objects by their true type, so the behavior of those objects is not changed when they are sent to another virtual machine. This allows new types to be introduced into a remote virtual machine, thus extending the behavior of an application dynamically.

Dynamic class loading from a web server

Steps:

1. Install and run a web server.
2. Remove the stub classes for the remote objects from the client and put them on the web server. Put the interfaces and any other classes used by the remote object on the web server as well.
3. Start the naming service (`rmiregistry`) from a location where it does not have access to the stubs.
4. When starting the server provide the URL of the location of the stubs as an argument so that it can inform the naming service of the location of the stubs – which is how the client will ultimately be able to find them. E.g.

```
java -Djava.rmi.server.codebase =http://localhost:8008/Server
```

RMI vs. Sockets

- RMI API can be used instead of the socket API in a network application.
- There are some tradeoffs:
  - The socket API is closely related to the operating system, and hence has less execution overhead.
- For applications which require high performance, this may be a consideration.
- The RMI API provides the abstraction which eases the task of software development.
  - Programs developed with a higher level of abstraction are more comprehensible and hence easier to debug.

RMI Callbacks

- In the client server model, the server is passive: the IPC is initiated by the client; the server waits for the arrival of requests and provides responses.
- Some applications require the server to initiate communication upon certain events. Examples applications are:
  - monitoring
  - games
  - auctioning
  - voting/polling
  - chat-room
  - message/bulletin board
  - groupware

RMI Callbacks

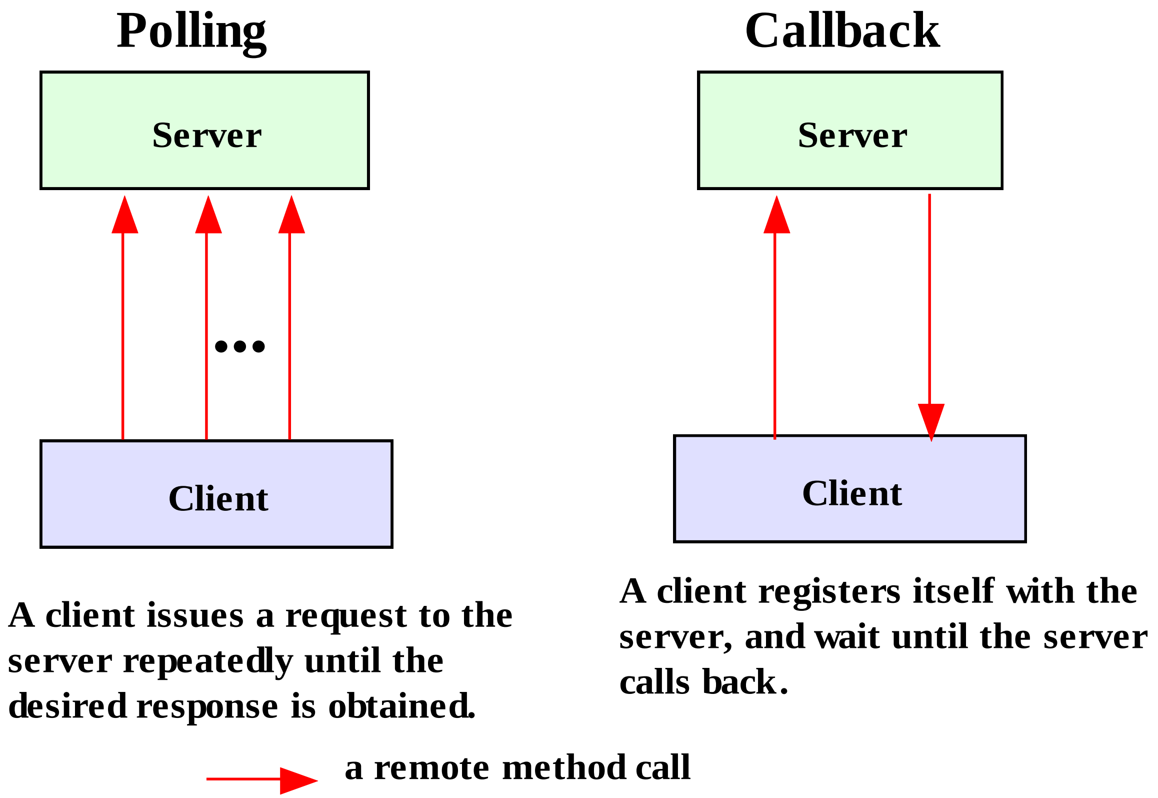
- In outline like any other callback
- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.

Callbacks

- Pass a reference to a remote object to another environment. A call to the stub of that object still call the original object i.e. a *call back*
- Call from server back to client.
- Remote reference to client must be available on server.
- Client object must be made remote.

Polling vs. Callback

In the absence of callback, a client will have to poll a passive server repeatedly if it needs to be notified that an event has occurred at the server end.

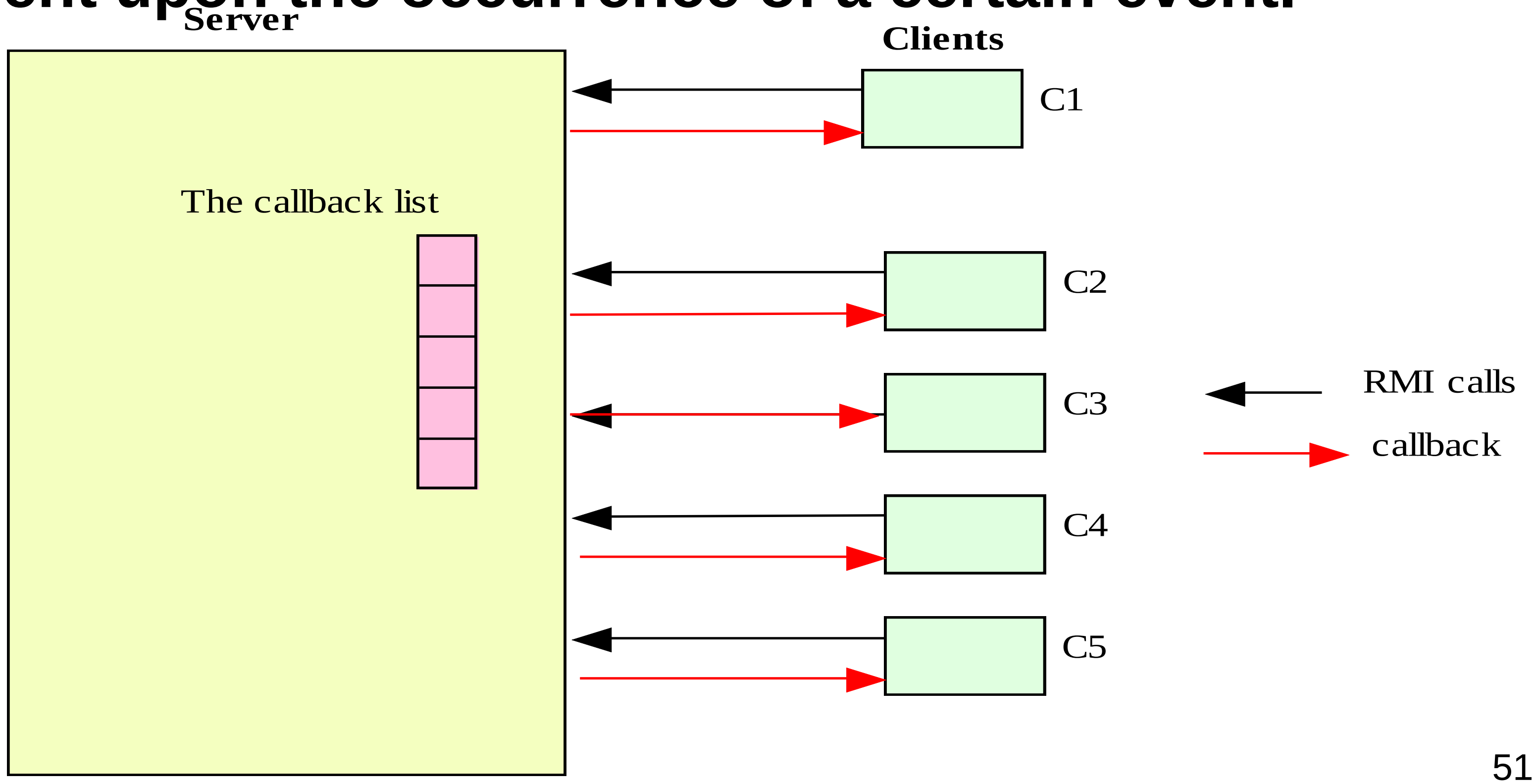


A client issues a request to the server repeatedly until the desired response is obtained.

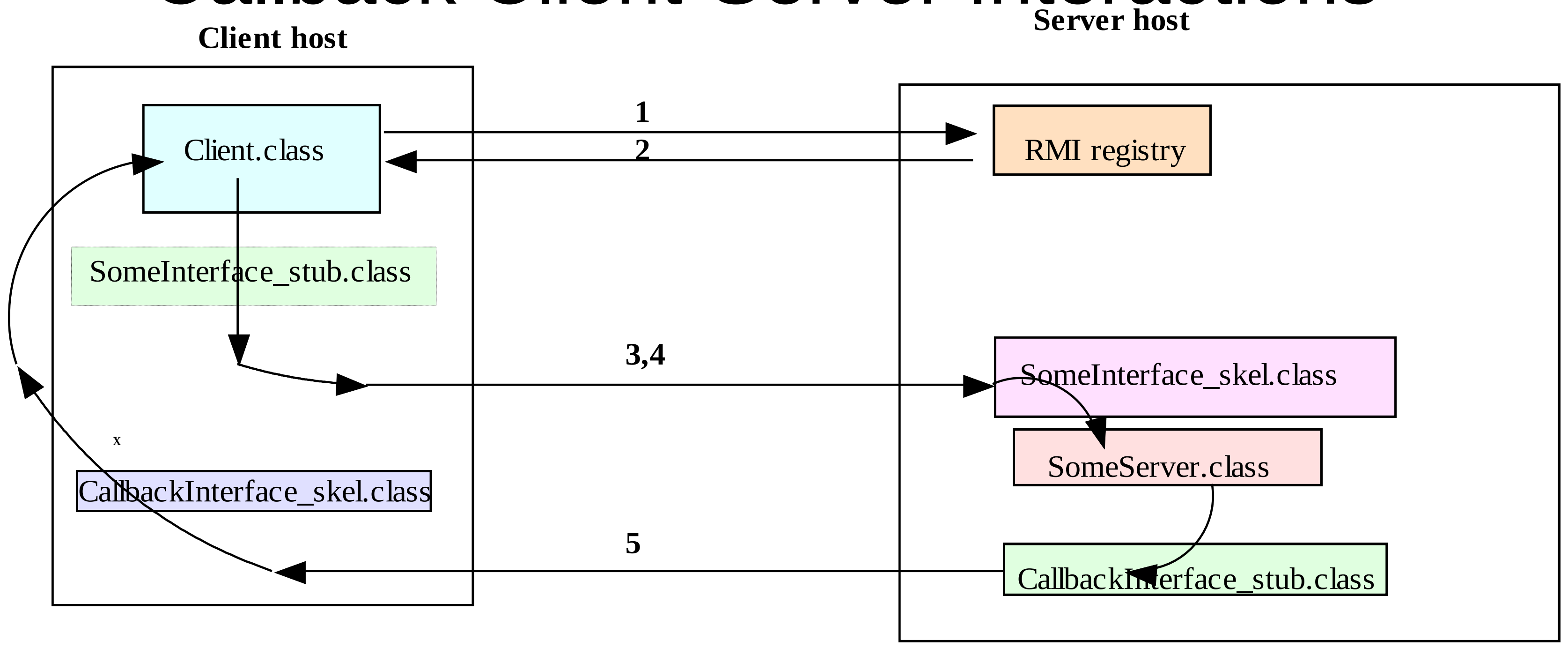
A client registers itself with the server, and wait until the server calls back.

RMI Callbacks

- **A callback client registers itself with an RMI server.**
- **The server makes a callback to each registered client upon the occurrence of a certain event.**

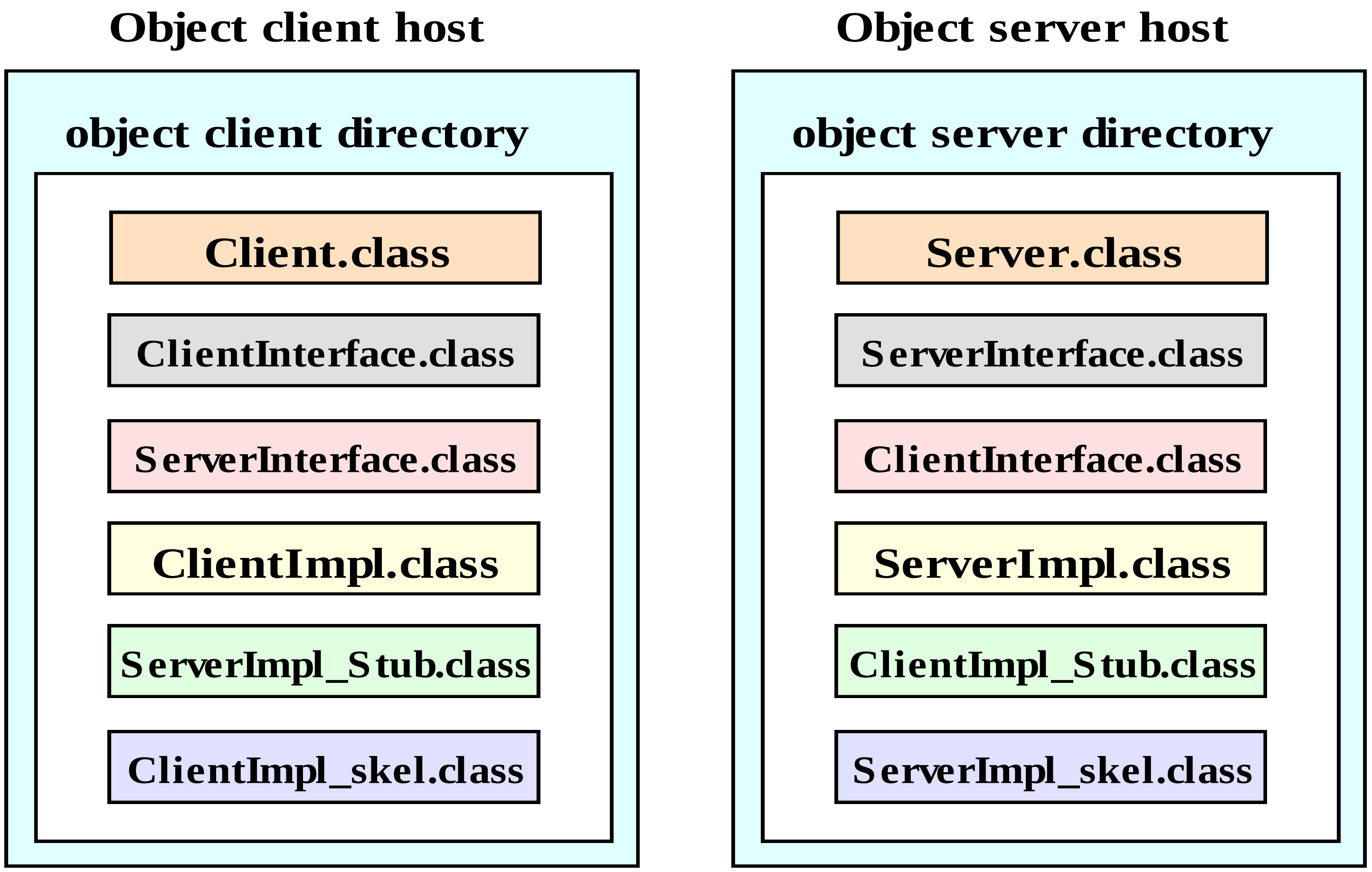


Callback Client-Server Interactions



1. **Client looks up the interface object in the RMIregistry on the server host.**
2. **The RMIRegistry returns a remote reference to the interface object.**
3. **Via the server stub, the client process invokes a remote method to register itself for callback, passing a remote reference to itself to the server. The server saves the reference in its callback list.**
4. **Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the interface object.**
5. **When the anticipated event takes place, the server makes a callback to each registered client via the callback interface stub on the server side and the callback interface skeleton on the client side.**

Callback application files



Testing and Debugging an RMI Application

1. Build a template for a minimal RMI program. Start with a remote interface with a single signature, its implementation using a stub, a server program which exports the object, and a client program which invokes the remote method. Test the template programs on one host until the remote method can be made successfully.
2. Add one signature at a time to the interface. With each addition, modify the client program to invoke the added method.
3. Fill in the definition of each remote method, one at a time. Test and thoroughly debug each newly added method before proceeding with the next one.
4. After all remote methods have been thoroughly tested, develop the client application using an incremental approach. With each increment, test and debug the programs.

Shared whiteboard example

- The distributed program that allows a group of users to share a common view of a drawing surface containing graphical objects, each of which has been drawn by one of the users.
- The server maintains the current state of a drawing and it provides operations for clients to:
  – add a shape, retrieve a shape or retrieve all the shapes,
  – retrieve its version number or the version number of a shape

Remote interfaces in Java RMI

- Defined by extending an interface called *Remote (java.rmi* package*).*
- Both ordinary objects and remote objects can appear as arguments and results in a remote interface.
- *Shape* and *ShapeList* examples:
  - *GraphicalObject* is a class that holds the state of graphical object (type, position); it must implement the *Serializable* interface.

Java Remote interfaces *Shape* and *ShapeList*

- Note the interfaces and arguments
- *GraphicalObject* is a class that implements *Serializable*.

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
   int getVersion() throws RemoteException;
   GraphicalObject  getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
   Shape newShape(GraphicalObject g) throws RemoteException;
   Vector allShapes() throws RemoteException;
   int getVersion() throws RemoteException;
}
```

Parameter and result passing

- The parameters of a method are assumed to be *input* parameters and the result of a method is a single *output* parameter.
- *Passing remote objects*:
  - When the type of a parameter or result value is defined as a remote interface, the corresponding argument or result is always passed a remote object reference.
    - E.g. the return value of the method *newShape* is defined as a Shape – a remote interface.
- *Passing non-remote objects*:
  - All serializable non-remote objects are copied and passed by value

Summary

- Stub downloading allows a stub class to be loaded to an object client at runtime, thereby allowing a remote object’s implementation to be modified and its stub class regenerated without affecting the software on the client host.
- A security manager oversees access restrictions specified in a **Java security policy file**, which can be a system-wide policy file, or a policy file applied to an individual application only.

- For security protection, the use of security managers is recommended in **all** RMI applications, **regardless** of whether stub downloading is involved.

References

- Chapter 5: Introduction to Network Programming in Java by Graba

- Ch 7 Slides : Liu, M. L., Distributed Systems: Principles and Applications