



# Chapters 4 and 10: COMMUNICATION Part 2b

## Communications in Distributed Systems Distributed Objects and **RMI**



Distributed Computing,  
M. L. Liu



Thanks to the authors of the textbook **[TS]** and **[MLL]** for providing the base slides. I made several changes/additions. These slides may incorporate materials kindly provided by Prof. Dakai Zhu. So I would like to thank him, too.

**Turgay Korkmaz**

korkmaz@cs.utsa.edu

# Chapters 4 and 10: Communications

---

## ■ FUNDAMENTALS

- Layered Protocols
- Types of communications

## ■ REMOTE PROCEDURE CALL

- Basic RPC Operation
- Parameter Passing
- RPC operation
- RPC Examples
- Asynchronous RPC
- RMI (some from chapter 10, but most from web and [MLL])
- CORBA (some from chapter 10, but most from web)

## ■ MESSAGE-ORIENTED COMMUNICATION

- Transient and Persistent Communication

## ■ STREAM-ORIENTED COMMUNICATION

- Support for Continuous Media and Quality of Service
- Stream Synchronization

## ■ MULTICAST COMMUNICATION

- Application-Level Multicasting
- Gossip-Based Data Dissemination

# Objectives

---

- To understand how processes communicate (the heart of distributed systems)
- To understand computer networks and their layers
- To understand client-server paradigm and low-level message passing using **sockets**
- To learn higher-level communication mechanisms
  - RPC, RMI, CORBA
- To understand various forms of communications and their issues
  - Msg-, Stream-oriented communication, multicast, etc.

# Background

## Object-oriented programming (OOP)

---

- Object-oriented programming (OOP) **encapsulates data and operations** into **objects**
- Operations are implemented as **methods** that are grouped into **interfaces** (the **signature** of a set of methods)
- Actions are performed in OOP by having objects invoke methods of other objects, the invoker is called a “**client**” of the object
- *Invocation can cause:*
  - *the state of the receiver to be changed (modifier methods)*
  - *additional invocations of methods on other objects*

# Background (cont'd)

## Object-oriented programming (OOP)

---

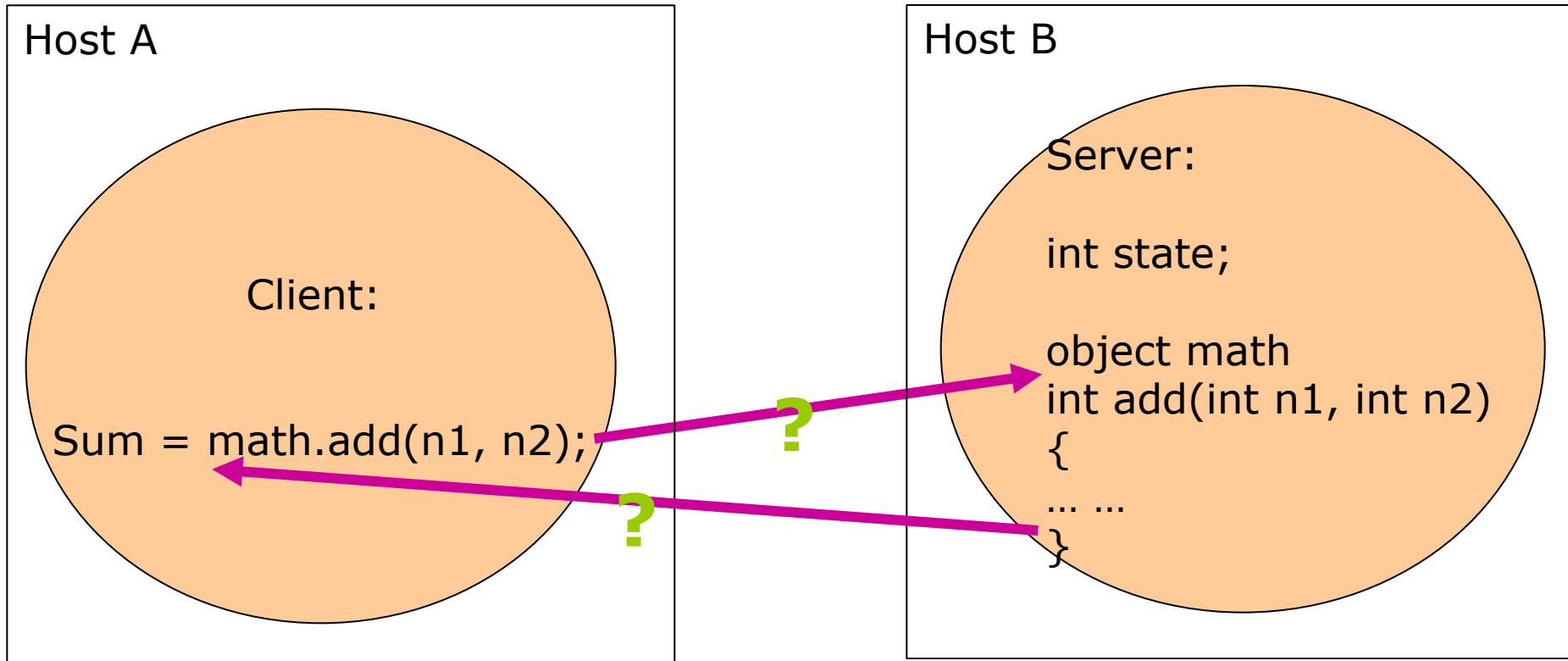
- **Exceptions** are thrown when an error occurs.
  - If object doesn't "catch" the exception, the exception is delivered to the caller
  - similar to signals, but at the programming language level
- **Additional concepts in OOP (wiki):**
  - **Object reference**
  - **Abstraction**
  - **Classes** of objects
  - **Instances** of classes
  - **Inheritance** Passes "Knowledge" Down
  - **Polymorphism** Takes any Shape
  - Dynamic dispatch
  - Message passing

# Procedural Programming vs. OOP

---

- Components (global variables + procedures)  
⇔ objects (data attributes + methods)
- Visible component state (global variables)  
⇔ object data attributes.
- Usable component services (procedures)  
⇔ object operations/methods.
- Component interactions (procedure call)  
⇔ operation execution requests.
- Component service failures  
⇔ exceptions

# Distributed Objects vs. Message Passing



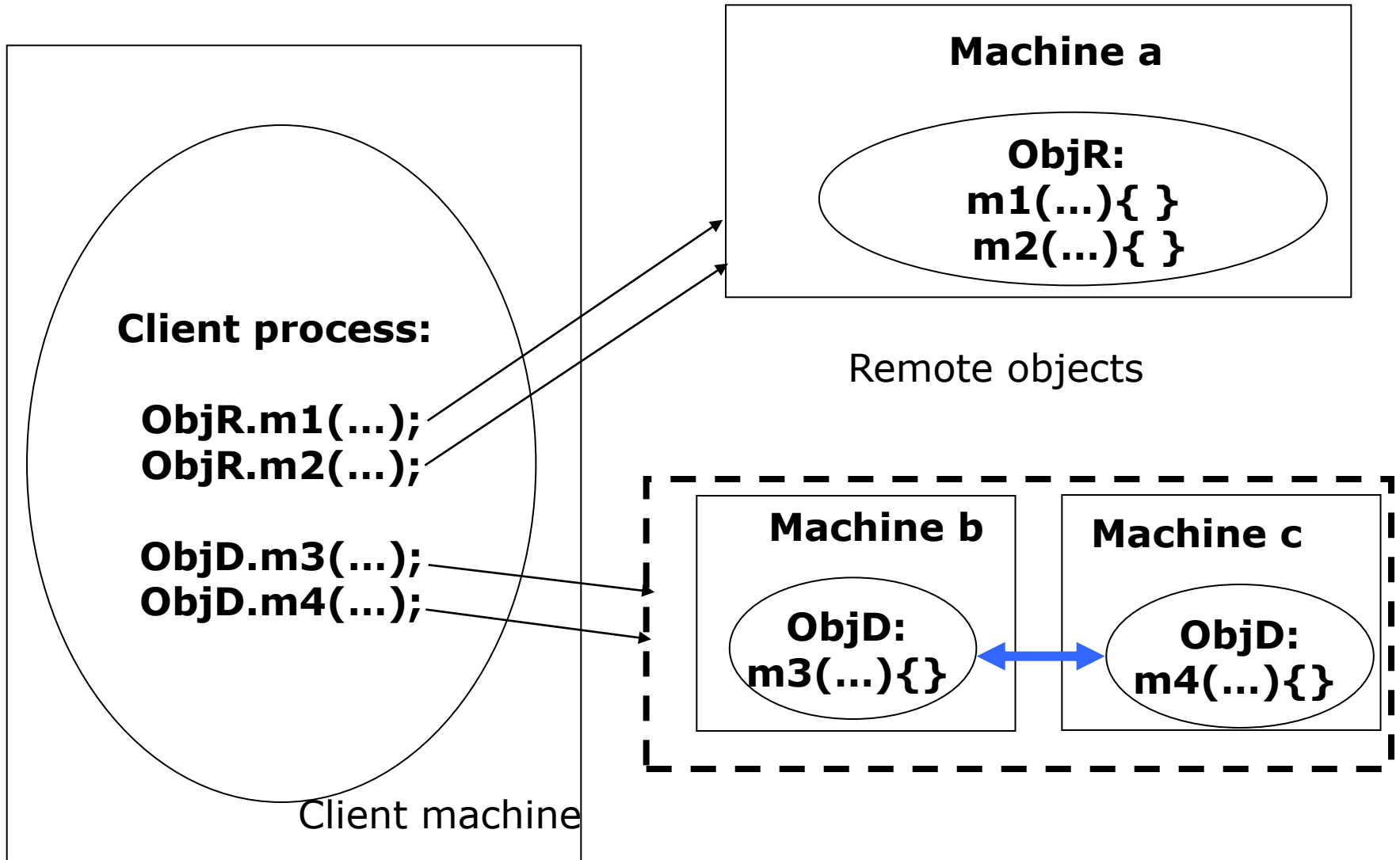
Create a socket  
Connect it to server

Put n1, n2 in a msg  
Send msg to server  
Read/wait reply msg  
Extract result from the msg

Create a socket  
Bind it to a port  
Accept a connection

Read/wait for a msg  
Extract n1, n2 from the msg  
Compute result  
Put it in a reply msg  
Send reply msg to client

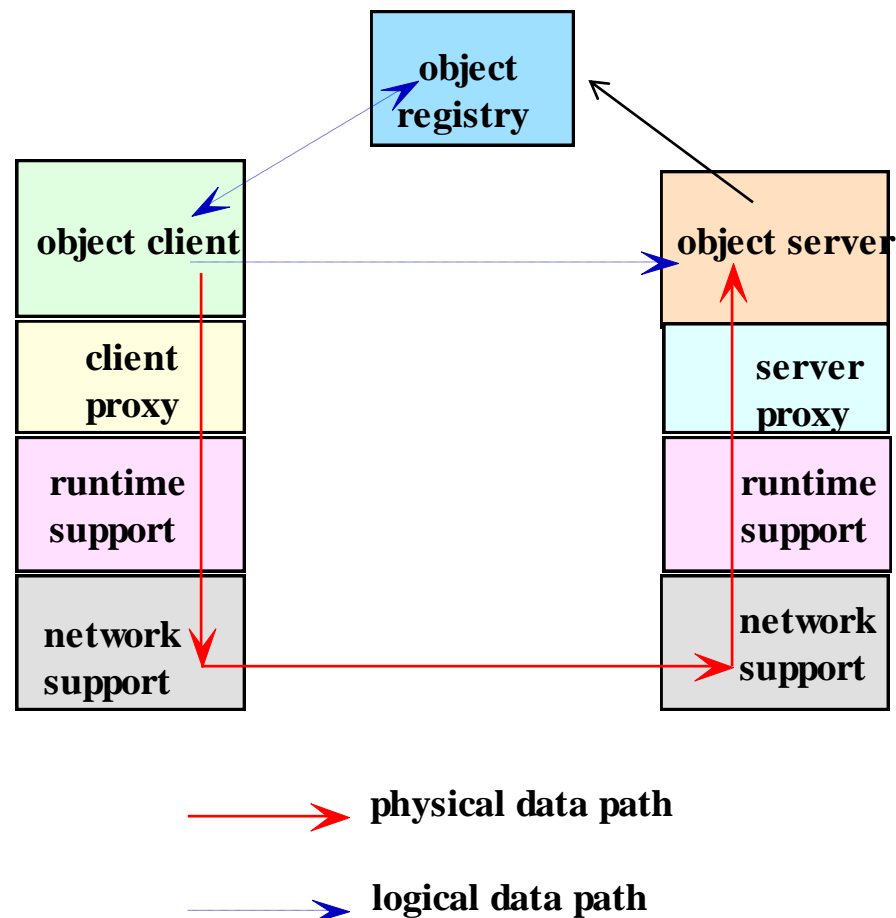
# Distributed Objects





# Distributed Object Systems

- Java RMI,
- CORBA (Common Object Request Broker Architecture)
- .NET and its predecessor, the Distributed Component Object Model (DCOM)
- Simple Object Access Protocol (SOAP) – web service



---

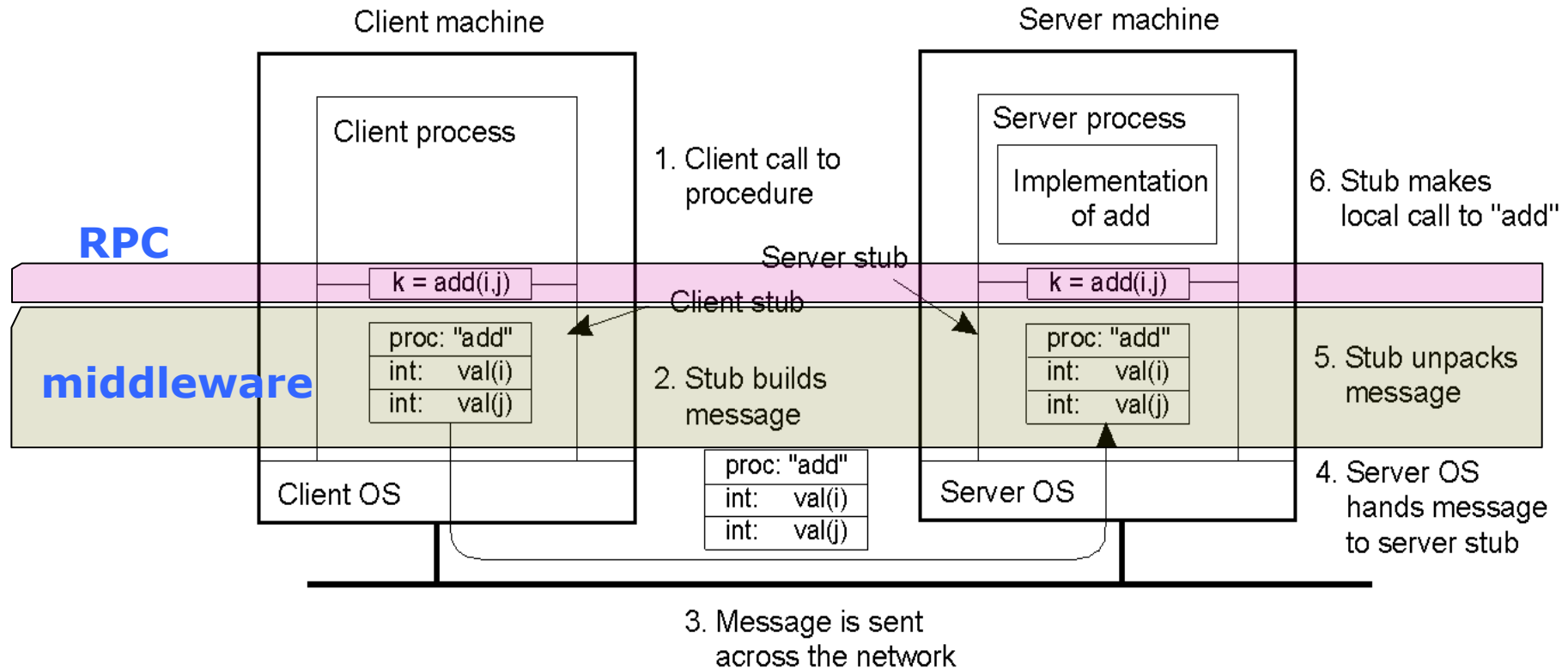
Recall that RPC allows to call a procedure on a different host!

RMI applies the same idea to **objects** and allows to invoke a method of an object that resides on a different host

In contrast to RPC, RMI allows parameters to be object references...

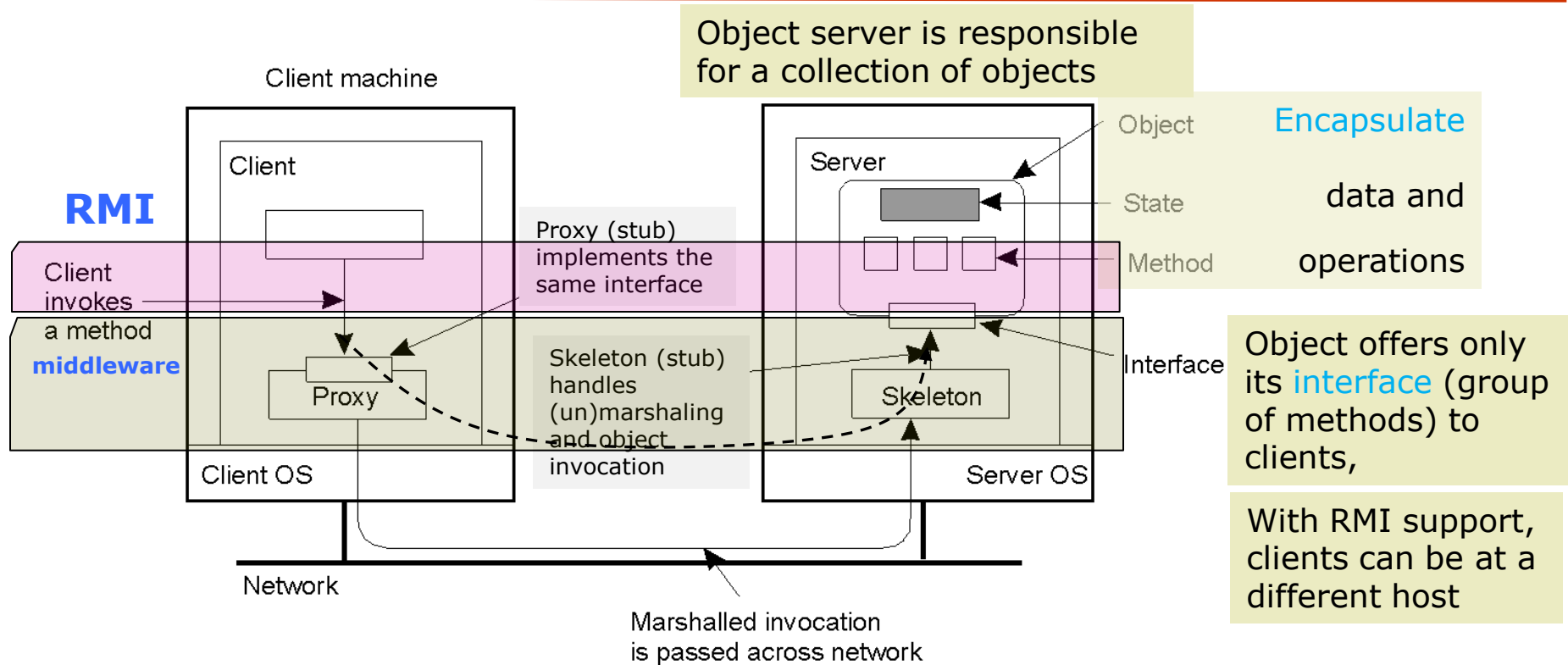
# RMI

# Recall RPC (Review)



- Middleware generates **stubs** on both sides

# RMI Overview

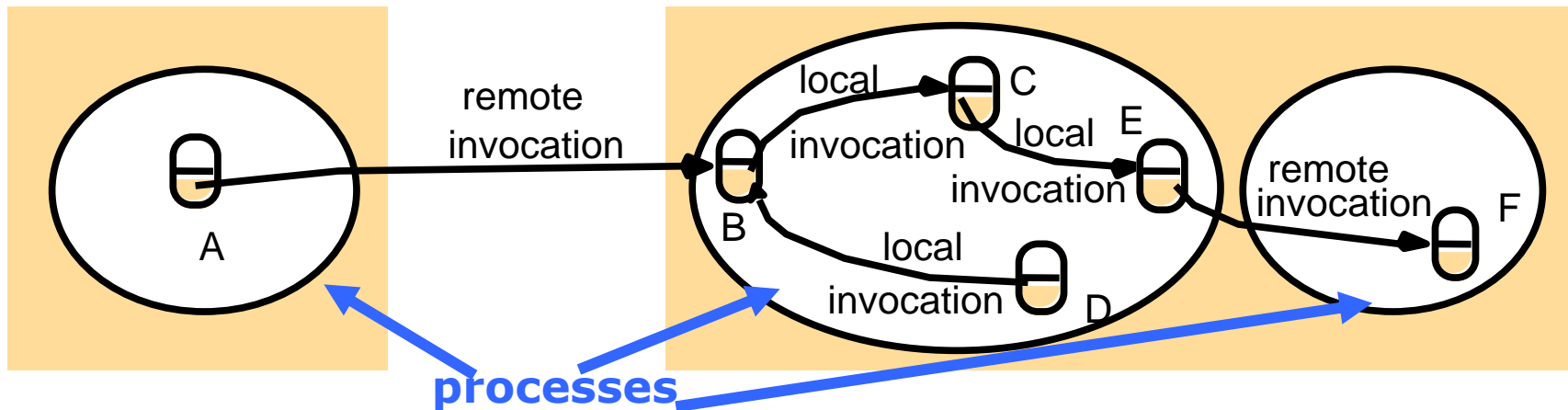


■ How do clients know where the remote objects are?

■ Binding...

To make a remote object accessible to other virtual machines, a program typically **registers it with the RMI registry**. The program **supplies to the registry the string name of the remote object as well as the remote object itself**. When a program wants to access a remote object, it **supplies the object's string name to the registry that is on the same machine as the remote object**. The **registry returns to the caller a reference (called stub) to the remote object**. When the program receives the stub for the remote object, it can invoke methods on the object (through the stub).

# Distributed, Remote Object Model



- Process contains one or more objects (local/remote)
  - Local objects (e.g., C, E): accept only local invocations
  - Remote object (e.g., B, F): accept both local/remote invocations
  - Remote invocation – different processes (same or different hosts)
- **Object references** are required for invocation
  - C must have E's reference (local) or
  - A must have B's reference (remote)

***What are the local and remote object references?***

# Remote Object Reference (ROR)

---

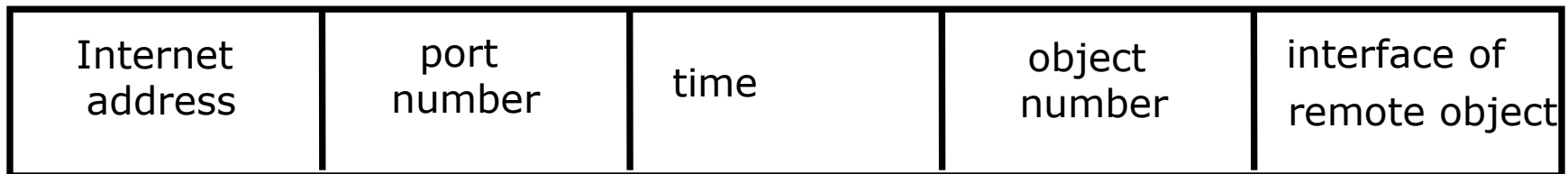
- Local object reference could be a **memory location**
- ROR must **uniquely** identify remote objects in distributed systems (so what additional information is needed then?)

*32 bits*

*32 bits*

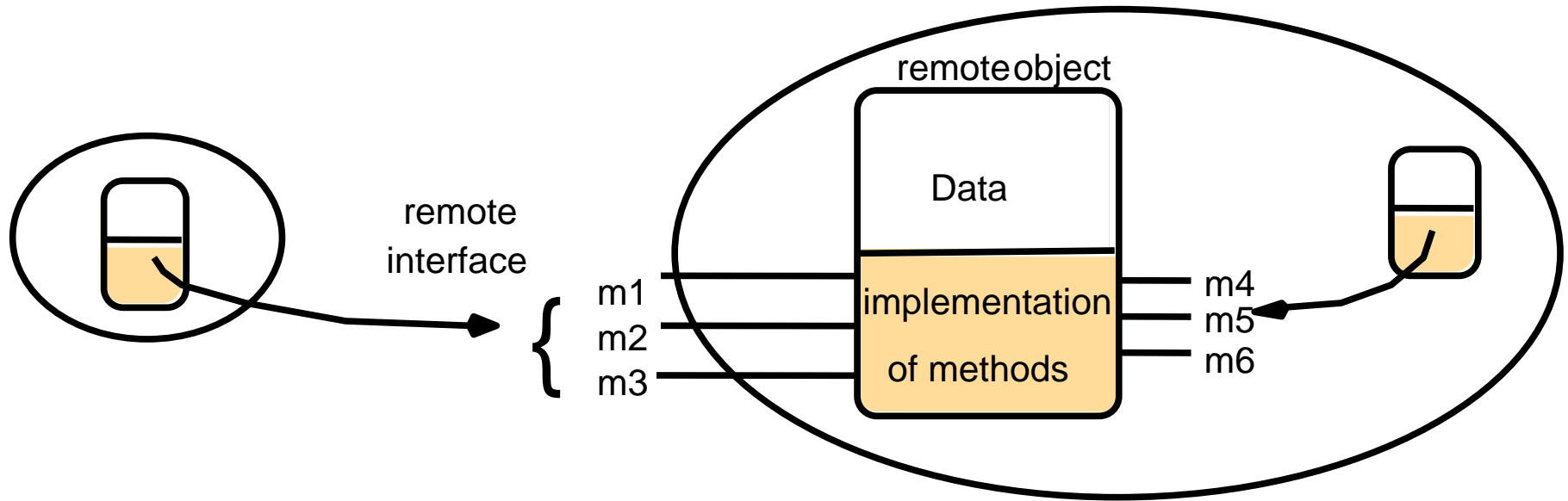
*32 bits*

*32 bits*



- Needed to invoke a method of a remote object
- Remote object references may be passed as input arguments or returned as output arguments.

# An Example: Remote Object



- An object may implement both remote and local **interfaces** (see next slide)
  - Other processes use ROR and invoke only methods in remote interface
  - Same process uses local object reference and invoke only methods in local interface

# Interfaces

---

## ■ Interface for local objects

- Specify methods and/or data that can be accessed
- Do not specify an implementation

## ■ Interface for remote objects

- Specifies methods for **remote invocation**
- **Input** and **output** parameters are also specified and parameters may be objects or ROR
- CORBA – uses IDL to specify remote interfaces
- JAVA – uses ordinary interfaces that are extended by the keyword `remote`.



# Parameter Passing in RMI

Less restrictive than RPCs

## ■ **Primitive** types (int, double etc.)

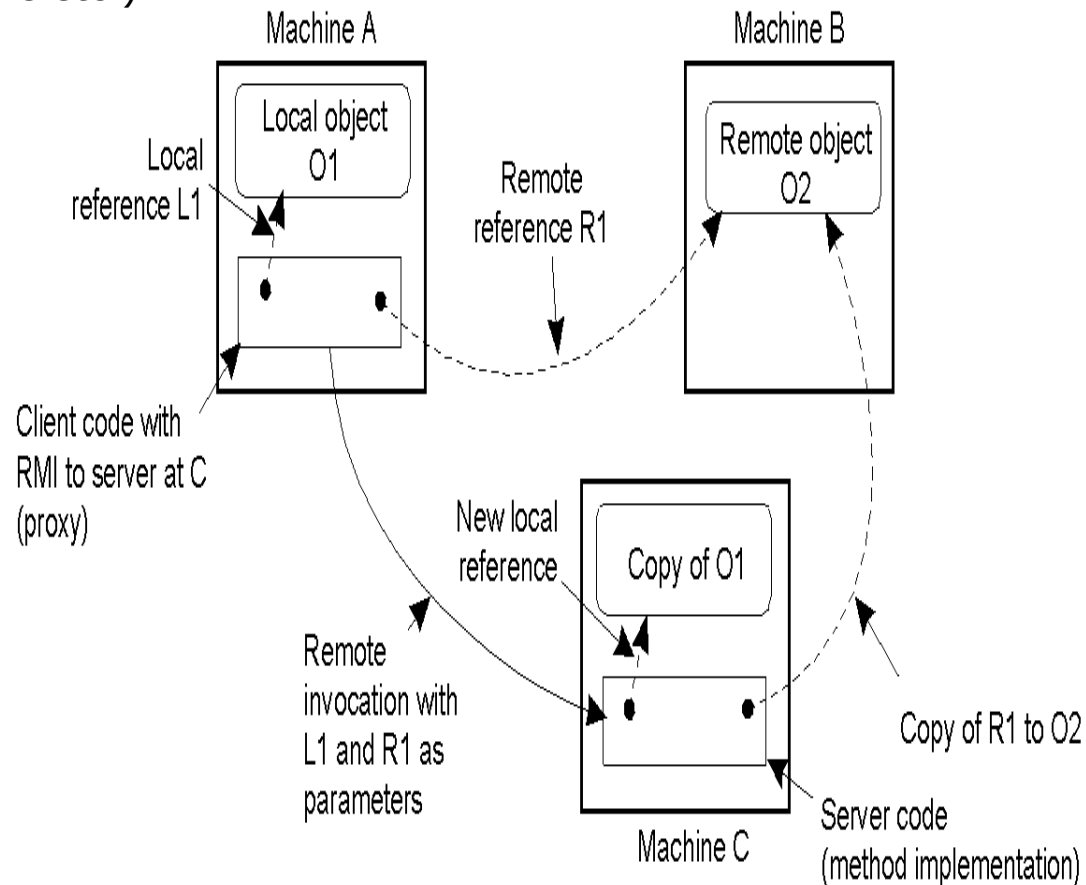
- pass by value

## ■ **Ordinary** objects

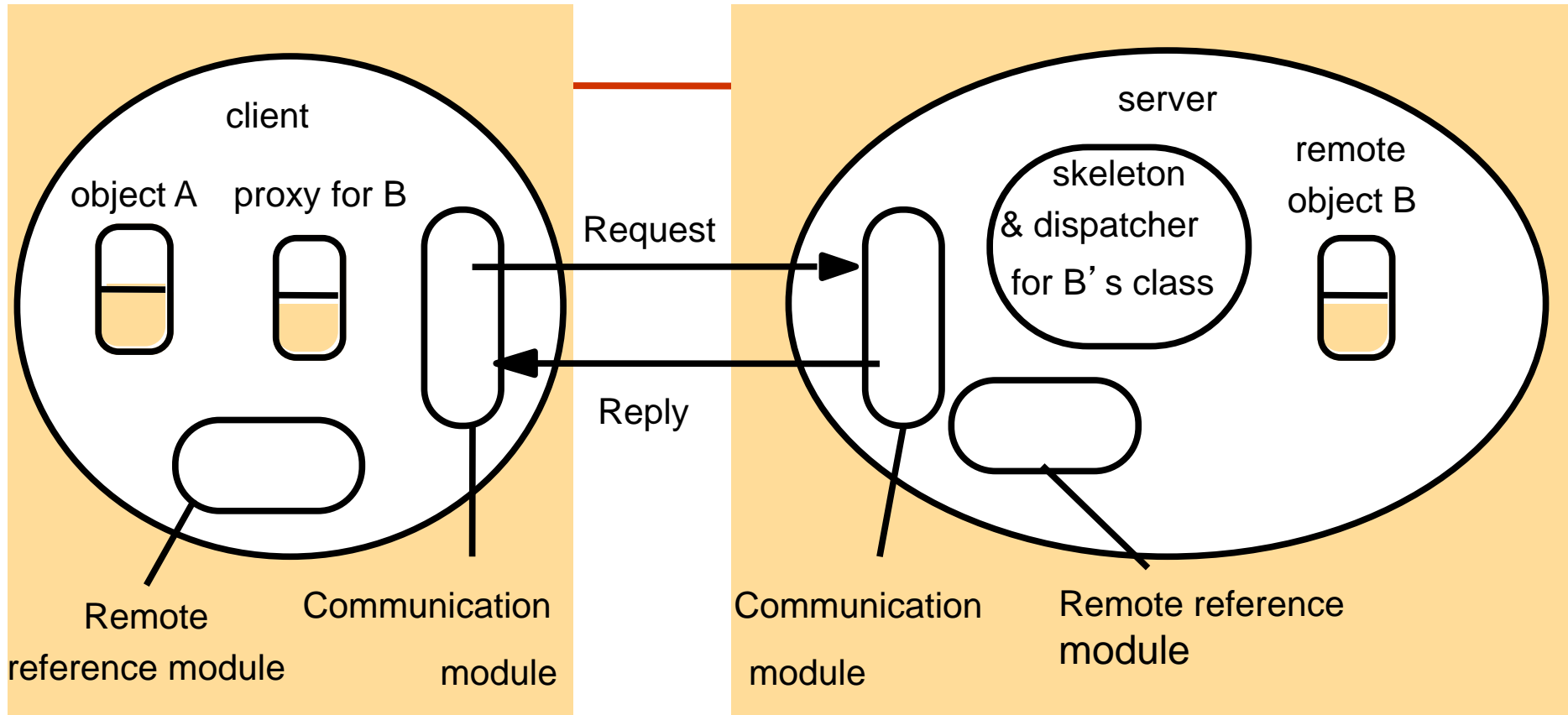
- passed by copy (e.g. using Java serialization; the object must implement the [java.io.Serializable](#) Interface).

## ■ **Remote** objects

- extends `UnicastRemoteObject` implements `ObjInterface`
- pass the remote object reference (ROR)
- Supports system-wide object references



Why not pass a copy of the remote object?

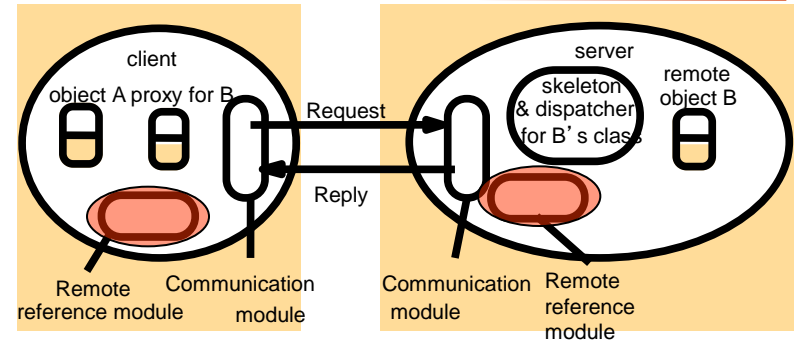


# ARCHITECTURE TO SUPPORT REMOTE OBJECTS

# Remote Reference Module (RRM)

## ■ Server side:

- Create remote object reference (ROR)
- Maintain remote object reference table
  - ▶ One entry per remote object
- **Map between remote reference and local reference**



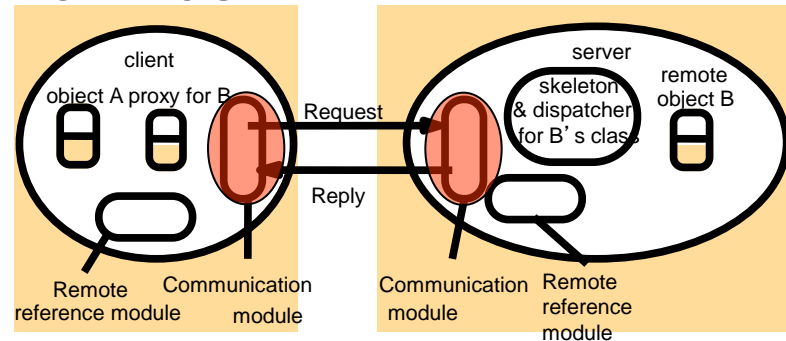
## ■ Client side:

- Create proxy object when first get ROR
- Maintain remote object reference table
- Entry for local proxy (client side)
- **Map between remote reference and local reference**

# Communication Module

- Carry out request-reply protocol
- Provide certain invocation semantics

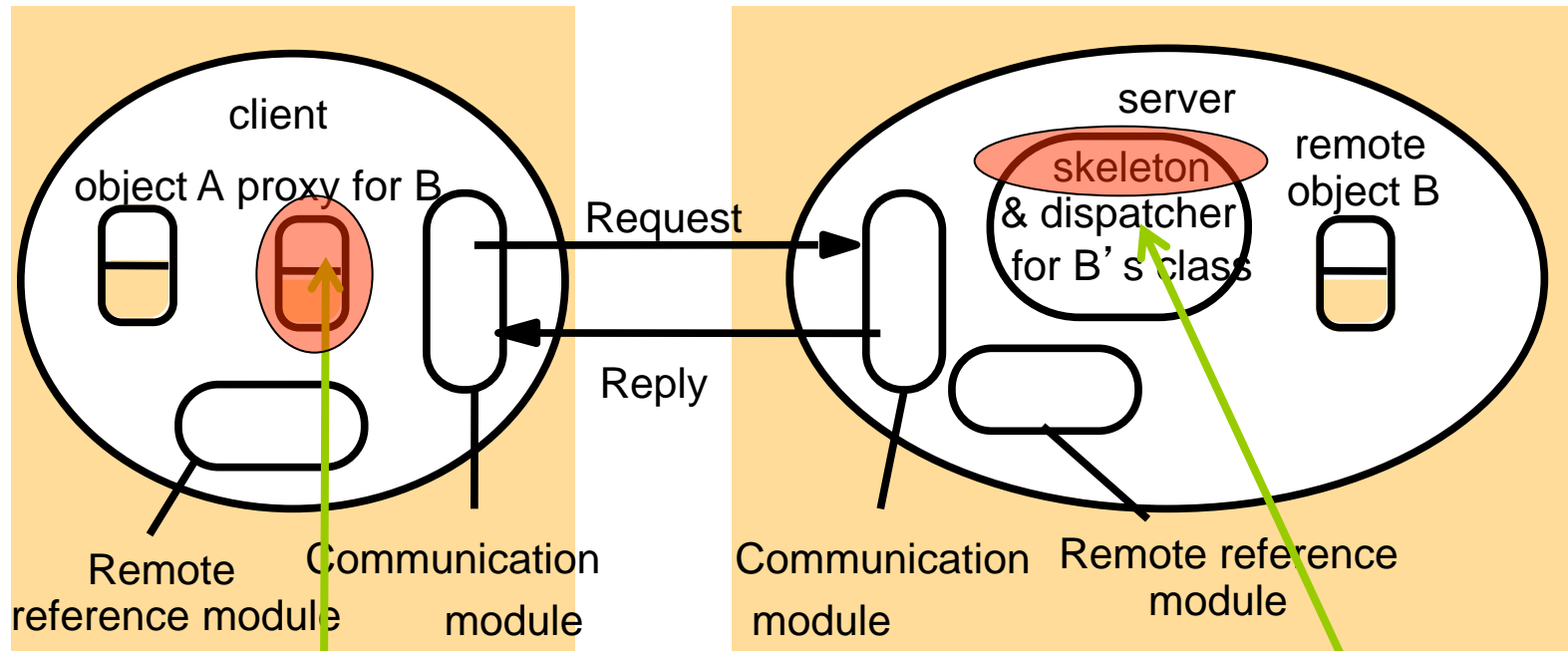
- Retry request
- Duplication message filtering
- Reply message history cache



- Interact with remote reference module

- Get remote object's local reference
- Pass message and local reference to appropriate dispatcher on server side

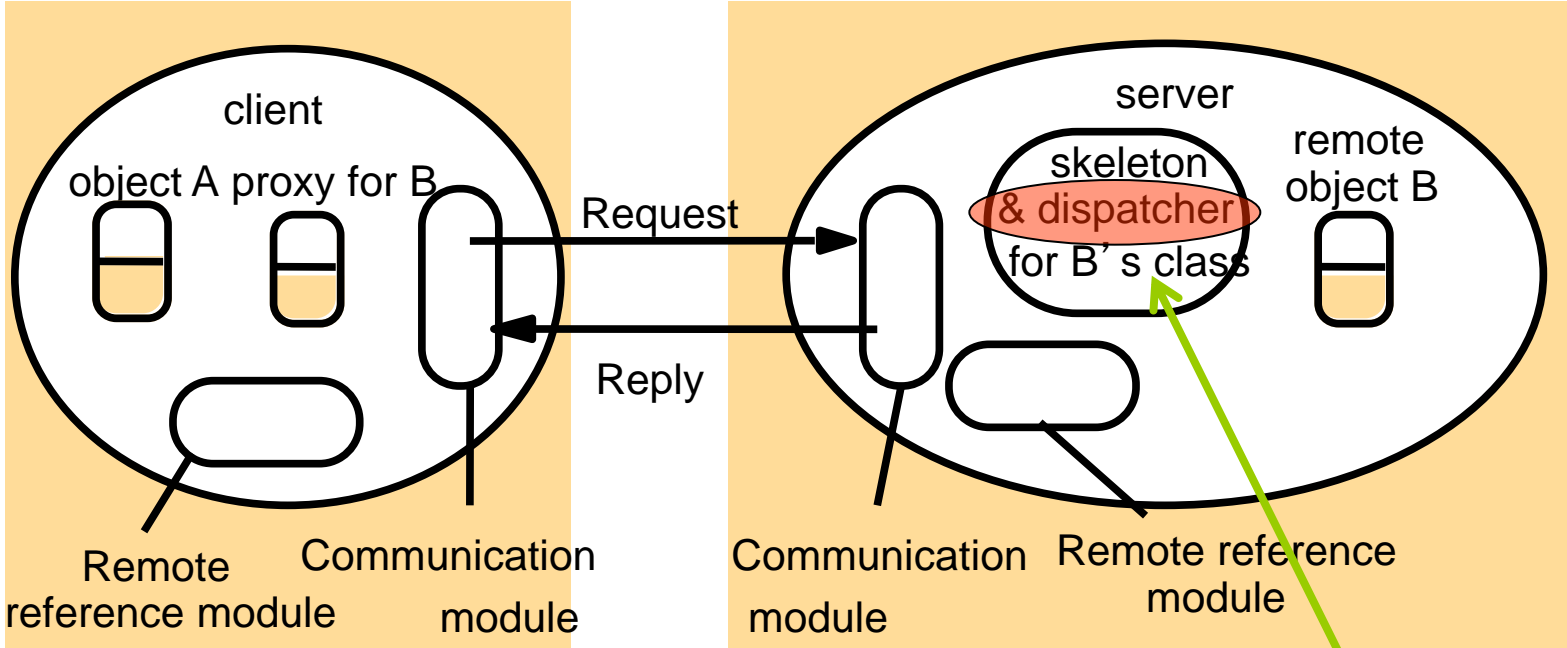
# Proxy and Skeleton



*Proxy* (client stub)- When a client binds to a distributed object, load the interface (“proxy”) into client address space on the fly. It makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

*Skeleton* - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.

# Dispatcher



*Dispatcher* - gets request from communication module and invokes method in skeleton (using *methodID* in message).

# Middleware for Remote Objects

---

- Middleware is a layer between application and communication/remote reference modules
- Automatically create proxy, skeleton and dispatcher from **remote interface** definition
- Client side: one proxy for each remote object
  - Call the methods in remote objects
- Server side: one dispatcher / skeleton per class
  - Dispatcher accepts message and select appropriate method in the skeleton: methodID
  - Skeleton: Marshall / unmarshall messages and invokes corresponding method in the remote object

# Server/Client Programs and Binder

---

## ■ Server program

- Dispatcher, skeleton → middleware generated
- **Servant class**: implement methods for remote objects

## ■ Client program

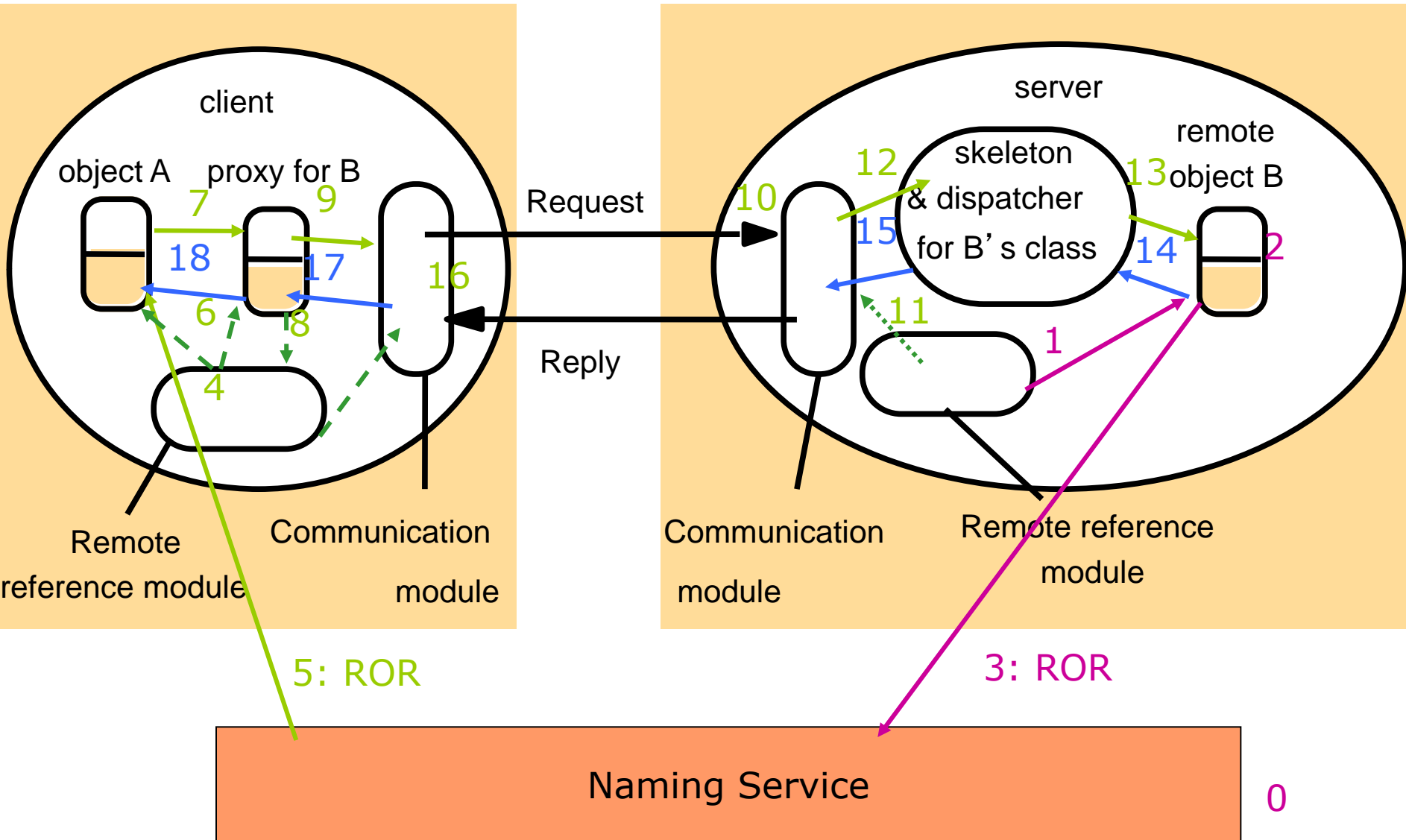
- Proxy → middleware generated
- Use binder to locate remote objects → obtain reference to local proxy

## ■ **Binder**: kind of name service

- Mapping between text name and remote object reference
- System wide register/look up service



# Steps in RMI



# How to Use Remote Objects: Server Side

---

- Step 0: start binder
- Step 1: start communication and remote reference module
- Step 2: server create remote object, add it to remote object table, and obtain remote object reference from remote reference module
- Step 3: publish the remote object to name service → bind the remote object reference with a **name**; wait for invocation requests

# How to Use Remote Objects: Client Side

---

- Step 4: start communication and remote reference module
- Step 5: contact name service for desired remote object reference
- Step 6: remote reference module create proxy
- Step 7: call methods in proxy → mashall parameters
- Step 8: locate remote object through remote reference module
- Step 9: Send method invocation request (contain remote object reference) through communication module

# How to Use Remote Objects: Server Side

---

- Step 10: get invocation requests (contains remote object reference) through communication module
- Step 11: consult with remote reference module and get local reference for the remote object
- Step 12: hand the request to the dispatcher/skeleton of the remote object's class → which method
- Step 13: call method in skeleton → unmarshall parameters in request and invoke real-method in remote object
- Step 14: perform the operation in remote object and return results to skeleton
- Step 15: marshall results in skeleton and send out message (remote object reference) through communication module

# How to Use Remote Objects: Client Side

---

- Step 16: get result message (contain remote object reference) in communication module
- Step 17: obtain proxy reference from remote object reference and hand the result message to the proxy
- Step 18: unmarshall the results in proxy and return results to the caller

*Complete a Remote Method Invocation!!!* 😊

Then, why do people say RMI is simpler than sockets?

# Comparison of RMI and Sockets

---

- In RMI, there is no need to design a protocol, which is an error-prone task. (recall slide 7 and the homework)
- In RMI, the developer thinks he is calling a local method from a local class file. The arguments are shipped to the remote target and interpreted, and the results are sent back to the callers.
- But, Sockets have less overhead than RMI. For applications which require **high performance**, this may be a consideration.
- So which one you will select for your application?

---

Java RMI

CORBA (Common Object Request Broker Architecture)

Distributed Component Object Model (DCOM)

Simple Object Access Protocol (SOAP) – web service

# **CASE STUDIES: APPLICATION DEVELOPMENT**

# Steps involved in developing Java RMI applications

---

## ■ Define a remote interface

HelloInterface.java

## ■ For the server

- Implement the interface

HelloImp.java

- Develop the server

HelloServer.java

## ■ For the Client

- Develop a client

HelloClient.java

## ■ Run the RMI registry, the server, and the client

```
//HelloInterface.java
import java.rmi.*;

public interface HelloInterface
extends Remote {

    public String add(String s)
        throws RemoteException;

    public String say( )
        throws RemoteException;

}
```



# HelloImp.java

## Servant Class Implement Remote Interface

---

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

public class HelloImp extends UnicastRemoteObject
    implements HelloInterface {
    private String message;

    public Hello (String msg) throws RemoteException {
        message = msg;
    }

    public String add(String s) throws RemoteException {
        message = new String (message + s);
        return message;
    }

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

# HelloServer.java

## Server: Create Servant Object and Bind

---

```
//HelloServer.java
import java.rmi.*;
public class HelloServer{
    public static void main(String args[]) {
        try {
            Naming.rebind("Hello",
                new Hello("Hello, world!"));
            System.out.println ("Server is ready");
        } catch (Exception e) {
            System.out.println ("Server failed:"+e);
        }
    }
}
```

If your program defines Serializable classes that need to be downloaded to another machine, then insert the statement `System.setSecurityManager(new RMISecurityManager());` as the first statement in the main

# HelloClient.java

---

```
//HelloClient.java
import java.rmi.*;
import java.rmi.server.*;
public class HelloClient{
    public static void main (String[] argv) {
        try {
            HelloInterface hello = (HelloInterface)
                Naming.lookup("//localhost/Hello");
            System.out.println(hello.say( ));
            System.out.println(hello.add("Here is added
                information!!!"));
        } catch (Exception e) {
            System.out.println ("HelloClient exception:"+e);
        }
    }
}
```

# Java RMI: Compile and Run the Server

(current host X)

---

## ■ Compile the interface and remote class

```
javac HelloInterface.java HelloImp.java
```

## ■ Compile server

```
javac HelloServer.java
```

## ■ Start RMI registry # (default at port 1099)

- `rmiregistry [port] &`

## ■ Start Hello server

- `java HelloServer &`

## ■ Generate skeletons & stubs (old Java compiler)

- `rmic Hello --> Hello_Skel.class & Hello_Stub.class`

# Java RMI: Compile and Run the Client

(Suppose host X is localhost)

## ■ Compile the interface class

```
javac HelloInterface.java
```

## ■ Compile client

```
javac HelloClient.java
```

## ■ Start Hello client

```
java HelloClient [X]
```

```
> java HelloClient
```

Hello, world!

Hello, world!Here is added information!!!

```
> java HelloClient
```

Hello, world!Here is added information!!!

Hello, world!Here is added information!!!Here is added information!!!

```
> java HelloClient
```

Hello, world!Here is added information!!!Here is added information!!!

Hello, world!Here is added information!!!Here is added information!!!Here is added information!!!

```
> ps
> kill -9 1611 #rmiregistry pid
> java HelloClient
HelloClient exception:
java.rmi.ConnectException: Connection
refused to host: localhost; nested exception
is:
    java.net.ConnectException:
Connection refused
```

# Basic Java RMI Summary

---

■ Design the remote interfaces `AbcInterface.java`

■ Server side

● `AbcImp.java`

- ▶ Implement remote methods in the remote interfaces
- ▶ Define the constructor for the remote object and implement local interface

● `AbCServer.java`

- ▶ Create server object and register it with “remote object” registry (so, `rmiregistry` must be executed before the server)

■ Client side

● `AbcClient.java`

- ▶ Looks up server in remote object registry and gets ROR
- ▶ Uses normal method call syntax for remote methods

# The binder: RMI Registry

---

## ■ For the server

- void **rebind** (String name, Remote obj)
  - ▶ Register object by name
  - ▶ Override previous registration
- void bind (String name, Remote obj)
  - ▶ Register an object by name
  - ▶ If existent throw exception
- void unbind (String name, remote obj)

//host:port/name



## ■ For the Client

- Remote **lookup** (String name)
  - ▶ Used by client
  - ▶ Remote object reference is returned
- String[] list()
  - ▶ Show all names bound in this registry

---

## **Factory classes and methods**

### **Client Callback**

Java RMI and synchronization

Distributed Garbage collection

Transient vs. Permanent objects

Stub Downloading

RMI security Manger

# **OTHER ISSUES AND ADVANCED JAVA RMI**



---

The **constructor** method of a remote object is **NOT** included in the interface. So, it cannot be called by the client

Where the remote objects come from?

**First remote object:** initiated by the server at startup and registered

**Factory method (remote method)** → create remote objects, and return remote object reference (ROR) based on client's requests

# REMOTE OBJECT FACTORY

# Factory Classes

---

*How to create remote objects, and who is responsible for what?*

- When a remote object reference (ROR) to a remote object is obtained through the RMI registry and then used to request additional RORs, the *registered* remote object is referred to as a *factory class*.
- Using RORs obtained through method calls on factory objects, client applications can dynamically request the creation of new remote objects, without the objects being registered individually with the server registry.

# Factory Class Example

---

- Consider a remote banking system using the *Account* object.
- The server provides services to remote clients running on PCs, embedded in ATMs etc.
- On the server, we run an RMI registry, create an *Account* object for every account we have on record, and register each one with the RMI registry using the account name.

```
Registry local = LocateRegistry.getRegistry();
```

```
local.bind("Abrams, John", new AccountImpl("John Abrams"));
```

```
local.bind("Adams, John", new AccountImpl("John Adams"));
```

```
:
```

- This is unwieldy. Starting the server can take long, as thousands of accounts need to be registered, many of them unnecessarily, since many accounts may not see any activity before the next down time. Also, accounts that are created or closed during the server's lifetime need to be added or removed from the RMI registry, as well as from the bank's database of accounts.

# Factory Class Example (cont'd)

---

- So we define a factory class for *Account* objects, as in:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface AccountManager extends Remote {
    public Account getAccount(String name) throws RemoteException;
    public Boolean newAccount(Account s) throws RemoteException;
}
```

- The *AccountManager* lets a client ask for an account by name, using the *getAccount()* remote method. The method returns a reference to an *Account* object that corresponds to the account. Once the client has an *Account* reference, transactions against the account can be done through method calls on the *Account* object. The *AccountManager* also has a *newAccount()* method that allows clients to add new accounts to the underlying database.
- The server implementation of the *getAccount()* method simply needs to look up the named account in the database, create an *AccountImpl* object to represent the account, and return the object to the client as a remote reference. Since *Account* objects are *Remote* objects, the RMI remote reference layer automatically creates a remote reference for the *Account* object, and the client that called the *getAccount()* method receives a stub for the *Account* object (thus the *Account* object is created on the server **but is not registered** with the registry and nor does the client need to call *Naming.lookup()* to look this object up. This avoids keeping the RMI registry in sync with the database and an unnecessary shadow of the database. Only the *AccountManager* object is registered with the registry and the client would only need to call *Naming.lookup()* to get a remote reference to the *AccountManager* object.) The *AccountManager* object can access the bank's database directly to find accounts and create corresponding *Account* remote objects.

---

Enable the server (or other peers)  
to invoke the methods on the client.

# **RMI CALLBACK**

## **REMOTE OBJECTS ON CLIENTS**

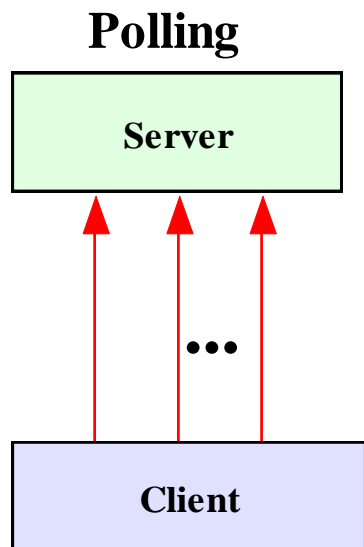
# RMI Callback

---

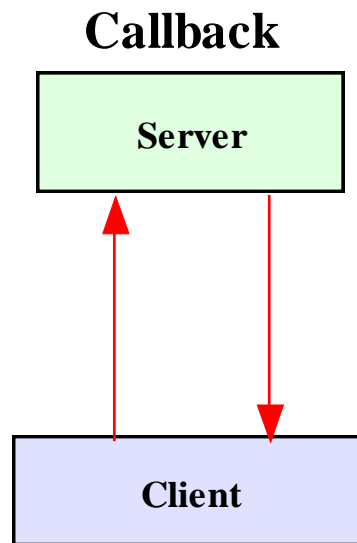
- In the client-server model,
  - the server is passive;
  - the IPC is initiated by the client;
  - the server waits for requests and provides responses
- Some applications require the server to initiate communication upon certain events such as
  - Auctioning: user submits bid, server inform if higher bit by others.
  - chat-room: user type message, server forwards messages from other users. message/bulletin board etc.
- With the RMI callback feature, client creates **remote objects** (callback objects) that implements an interface for server to call.
  - So we can now develop interactive distributed applications.

# RMI Callback vs. Polling

- In the absence of callback feature, how would a client be notified if it needs to know that a certain event has occurred at the server or not?



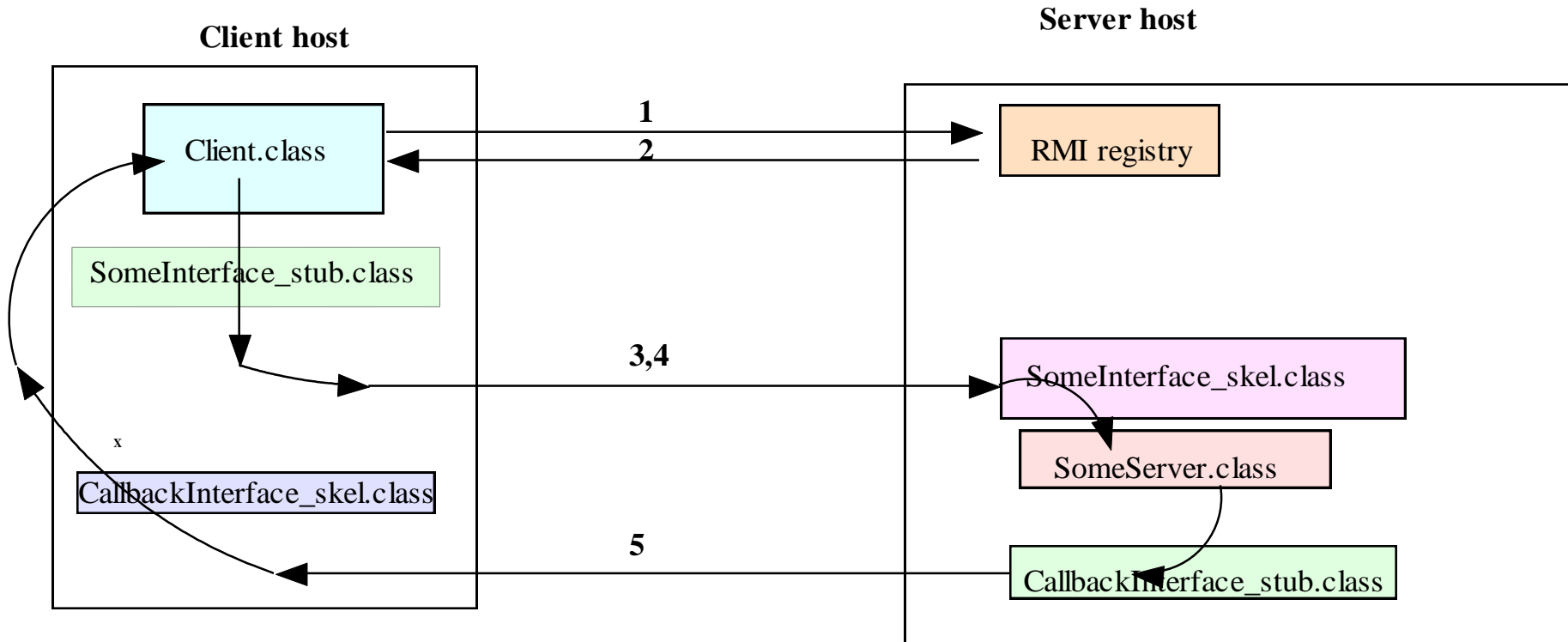
A client issues a request to the server repeatedly until the desired response is obtained.



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

→ a remote method call

# Callback Client-Server Interactions



1. Client looks up the interface object in the RMI registry on the server host.
2. The RMI Registry 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.



# Good and Bad about Callback

---

## ■ Advantages

- More efficient than polling
- More timely than polling
- Provides a way of server inquiring about client status

## ■ Disadvantages

- May leave server with inconsistent state if client crashes or exits without notifying the server
- Requires the server to make a series of synchronous RMI's

# RMI Callback example

<http://www2.cs.uic.edu/~i441/RMICallback/>

<http://searchdaily.net/tag/callback-pattern-rmi-example/>

[http://docs.oracle.com/cd/E13211\\_01/wle/rmi/callbak.htm](http://docs.oracle.com/cd/E13211_01/wle/rmi/callbak.htm)

<http://www.uwplatt.edu/csse/tools/java/samples/tictactoe/>

Just google ....

```
// Remote Interface for Server
public interface HelloInterface
    extends Remote {
    // remote method
    public String sayHello() throws
        java.rmi.RemoteException;
    // method to be invoked by a client to
    // add itself to the callback list
    public void addCallback(  
        HelloCallbackInterface  
        CallbackObject)
        throws java.rmi.RemoteException;
}
```

```
// Remote Interface for Callback Client
public interface HelloCallbackInterface
    extends java.rmi.Remote
{
    // method to be called by the server on
    // callback
    public void callMe (  
        String message  
    ) throws java.rmi.RemoteException;
}
```

# RMI Callback example (server)

```
public class HelloServer extends
    UnicastRemoteObject implements
    HelloInterface {
    static int RMIPort;
    // vector for store list of callback objects
    private static Vector callbackObjects;

    public HelloServer() throws RemoteException {
        super();
        // instantiate a Vector object for storing
        // callback objects
        callbackObjects = new Vector();
    }
    // method for client to call to add itself to its
    // callback
    public void addCallback(
        HelloCallbackInterface CallbackObject) {
        // store the callback object into the vector
        System.out.println("Server got an
        'addCallback' call.");
        callbackObjects.addElement (CallbackObject);
    }
}
```

```
public static void main(String args[]) {
    ...
    registry = LocateRegistry.createRegistry(RMIPort);
    ...
    callback();
    ...
} // end main

private static void callback() {
    ...
    for (int i = 0; i < callbackObjects.size(); i++) {
        System.out.println("Now performing the "+ i
        + "th callback\n");
        // convert the vector object to a callback object
        HelloCallbackInterface client =
            (HelloCallbackInterface) callbackObjects.elementAt(i);
        ...
        client.callMe ( "Server calling back to client " + i);
        ...
    }
}
```

# RMI Callback example (client)

---

```
public class HelloCallbackImp extends UnicastRemoteObject implements HelloCallbackInterface {
    public HelloCallbackImp() throws RemoteException { super(); }
    // call back method - this displays the message sent by the server
    public void callMe (String message) {
        System.out.println( "Call back received: " + message );
    }
} // end HelloCallbackImp class
```

```
public class HelloCallbackClient { public static void main(String args[]) {
    public static void main(String args[]) { // ...

        HelloInterface h = (HelloInterface)Naming.lookup(".....");
        ....
        HelloCallback clientCallback = new HelloCallbackImp();
        h.addCallback(clientCallback);
        while (true){
            ;} // end while
        } // end main
    } // end HelloClient class
}
```

# Serializable Obj

send a copy of the whole obj to other end in RMI

---

## ■ Serialization

- a mechanism of writing the state of an object into a byte stream.
- The reverse operation of serialization is called deserialization.
- The String class and all the wrapper classes implements java.io.Serializable interface by default.

■ In RMI, serialization is used to marshal/unmarshal method arguments that are objects, but that are not remote objects.

■ Any object argument to a method on a remote object in RMI must implement the Serializable interface.

```
public class User implements java.io.Serializable{
    private double x, y; // location
    private String name;
    ...
    // getters and setters
}
```

---

EXTRAS.....

---

# JAVA RMI AND SYNCHRONIZATION

# Java RMI and Synchronization

---

- Java supports Monitors: synchronized objects
  - Serializes accesses to objects
  - How does this work for remote objects?

Two Options: block at the client or the server

- Block at server
  - Can synchronize across multiple proxies
  - Problem: what if the client crashes while blocked?
- Block at proxy
  - Need to synchronize clients at different machines
  - Explicit distributed locking necessary
- Java uses proxies for blocking
  - No protection for simultaneous access from different clients
  - Applications need to implement distributed locking



---

RMI extends the internal garbage-collection mechanisms of the standard JVM to provide distributed garbage collection of remotely exported objects.

This is an automatic process that the application developer does not have to worry about.

# DISTRIBUTED GARBAGE COLLECTION

# Distributed Garbage Collection

---

## ■ Algorithm outline:

- Server: list of processes that hold remote object reference
- Client: when first request remote object reference, remote reference module calls **addRef** on server and then creates the proxy
- Client: no longer need a proxy (detected by local GC), the remote reference module calls **removeRef** on the server
- Server: when the list is empty and there are no local references the remote object is removed

# Distributed Garbage Collection (cont.)

---

- Pair-wise request-reply between remote reference modules
- Only called when proxies are created/deleted
- Fault-tolerance has to be addressed
  - Idempotent addRef and removeRef
  - removeRef is correct whether addRef worked or not
  - **Leases** (max. time to live) in case removeRef gets lost (or client crashed)

# Transient vs. Permanent Remote Objects

---

## ■ Transient Remote Objects

- exist within the process that creates the object

## ■ Persistent Objects can survive when process dies and be later activated by new process

- Long run objects: sleep for efficient resource usage and activated whenever necessary
- **Persistent object store** provides a simple storage management (like a database system)

# States for Persistent Objects

---

- **Active:** Ready for method invocation
- **Passive:** not active, and cannot accept invocation
  - A passive object consists of two parts
    - ▶ Implementation of its methods
    - ▶ Internal state in marshalled form
- **Activator**
  - “Passivate”: name of server and location of passive object
  - Activate: start server and active object within it
- *When to store information about object in persistent storage?*
  - *At the time of passivated*
  - *At end of important operations (e.g. end of transaction)*

# Object Location

---

- Remote object reference contains IP address and port # of server process to guarantee uniqueness
  - Advantage: can be used as address
  - Disadvantage: object cannot be migrated to other server
  
- **Location service**
  - Database to map remote object reference to their current location
  - Location services can be replicated on each machine; information on machines is kept more or less consistent by some update propagation mechanism