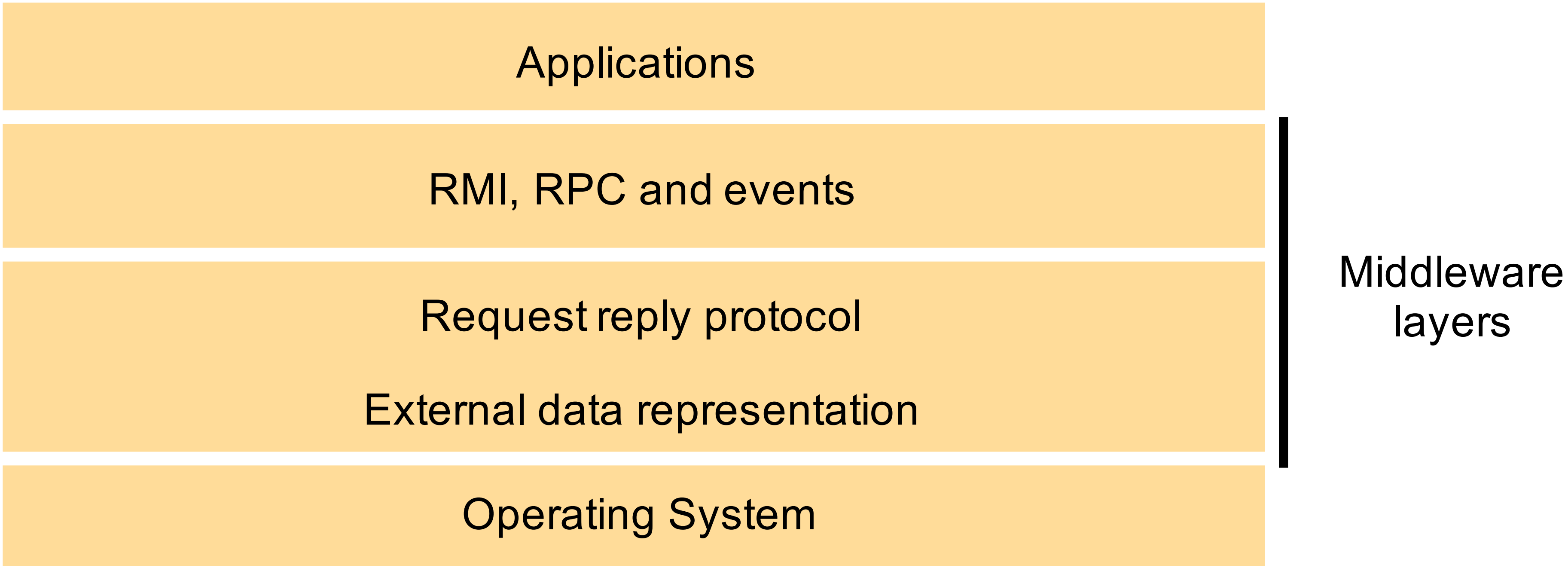
95-702 Distributed Systems

Distributed Objects and Java RMI

Learning Objectives:

- Understand distributed objects (Chapter 5 of Coulouris)
- Understand Java RMI (Chapter 5 Case study)
- Be able to describe the functional and non-functional characteristics of a distributed object system
- Be able to compare and contrast distributed objects with web services (which is faster?, which is more interoperable?, which is more popular in the industry?)
- Understand Martin Fowler’s First Law of Distributed Objects

Review: What is Middleware?



Applications

RMI, RPC and events

Request reply protocol

External data representation

Operating System

Middleware layers

- Applications (clients and services) speak to the middleware.
- TCP/IP is provided by the operating system.
- UDP/IP is provided by the operating system.

Review: external data representation

- *External data representation* – an agreed standard for the representation of data structures and primitive values
- *Marshalling* – the process of taking a collection of data items and assembling them into a form suitable for transmission in a message
- *Unmarshalling* – is the process of disassembling them on arrival into an equivalent representation at the destination
- The marshalling and unmarshalling are intended to be carried out by the middleware layer

Review: Examples of external data representation:

- CORBA's CDR binary data may be used by different programming languages when CORBA middleware is present

- Java RMI and .Net Remoting use object serialization. These are platform specific (that is, Java on both sides or .Net on both sides) and binary. For Java RMI, we have a JVM running on both sides.

- XML and JSON are primarily textual formats, verbose when compared to binary but interoperable. Why? These are not tied to a particular language or machine architecture.

Review: Generic Request-Reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

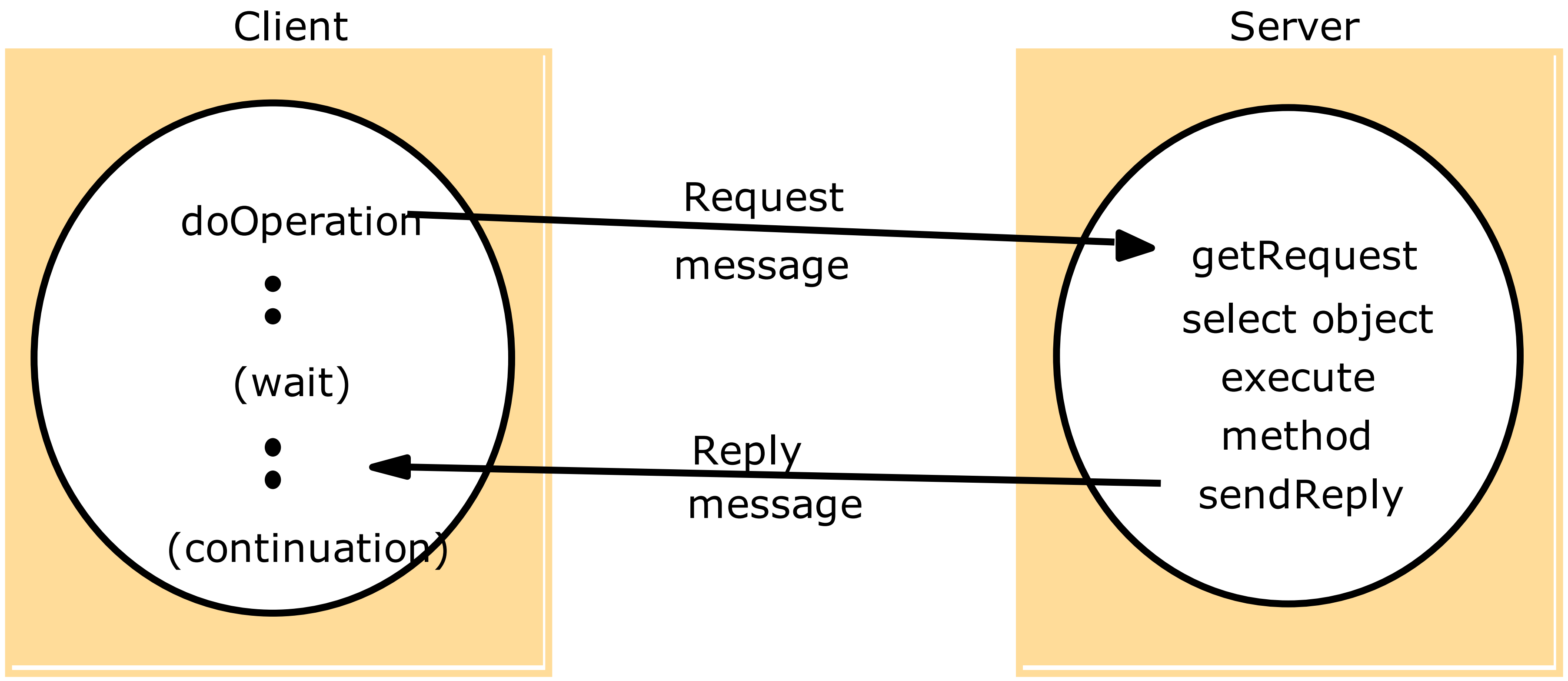
Review: generic remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

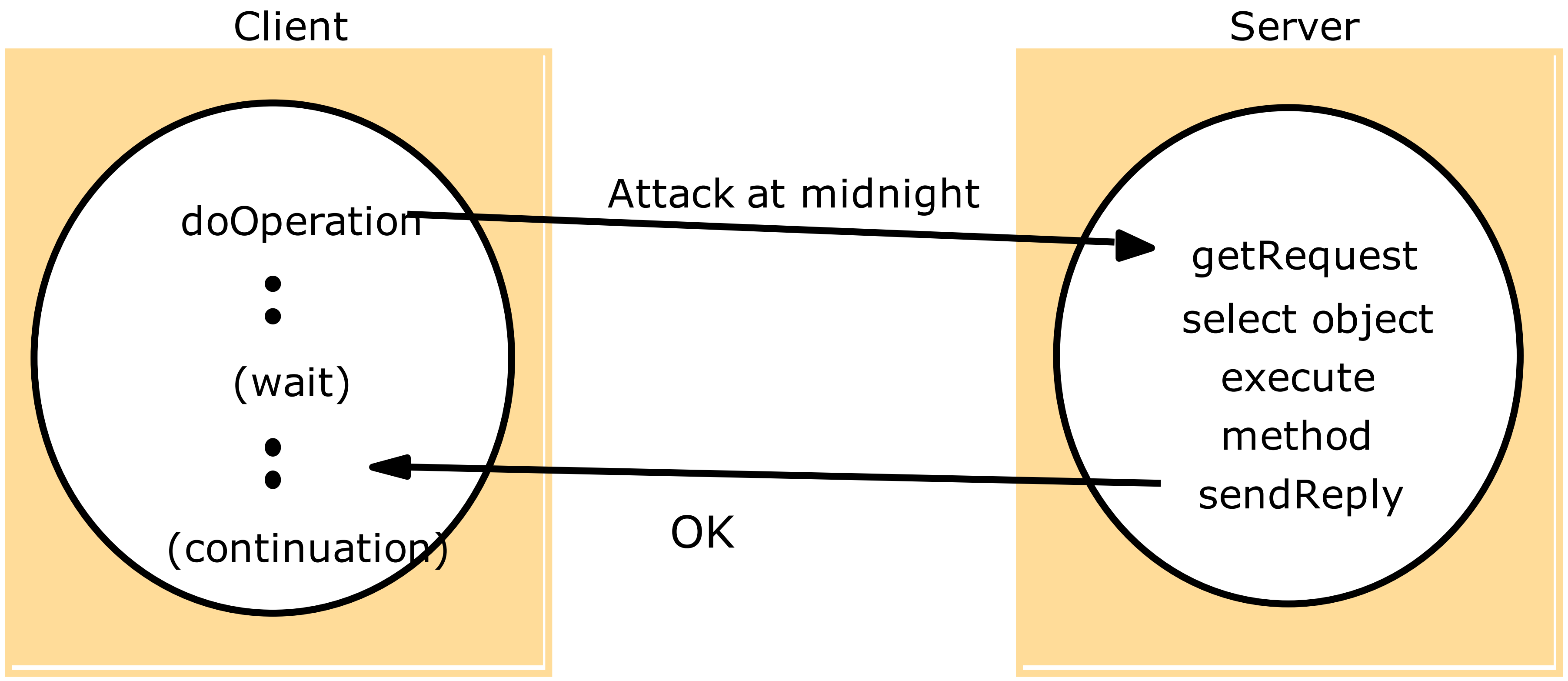
A remote object reference is an identifier for a remote object.
May be returned by or passed to a remote method.

How do these references differ from local references?

Review: UDP Style Request-Reply Communication



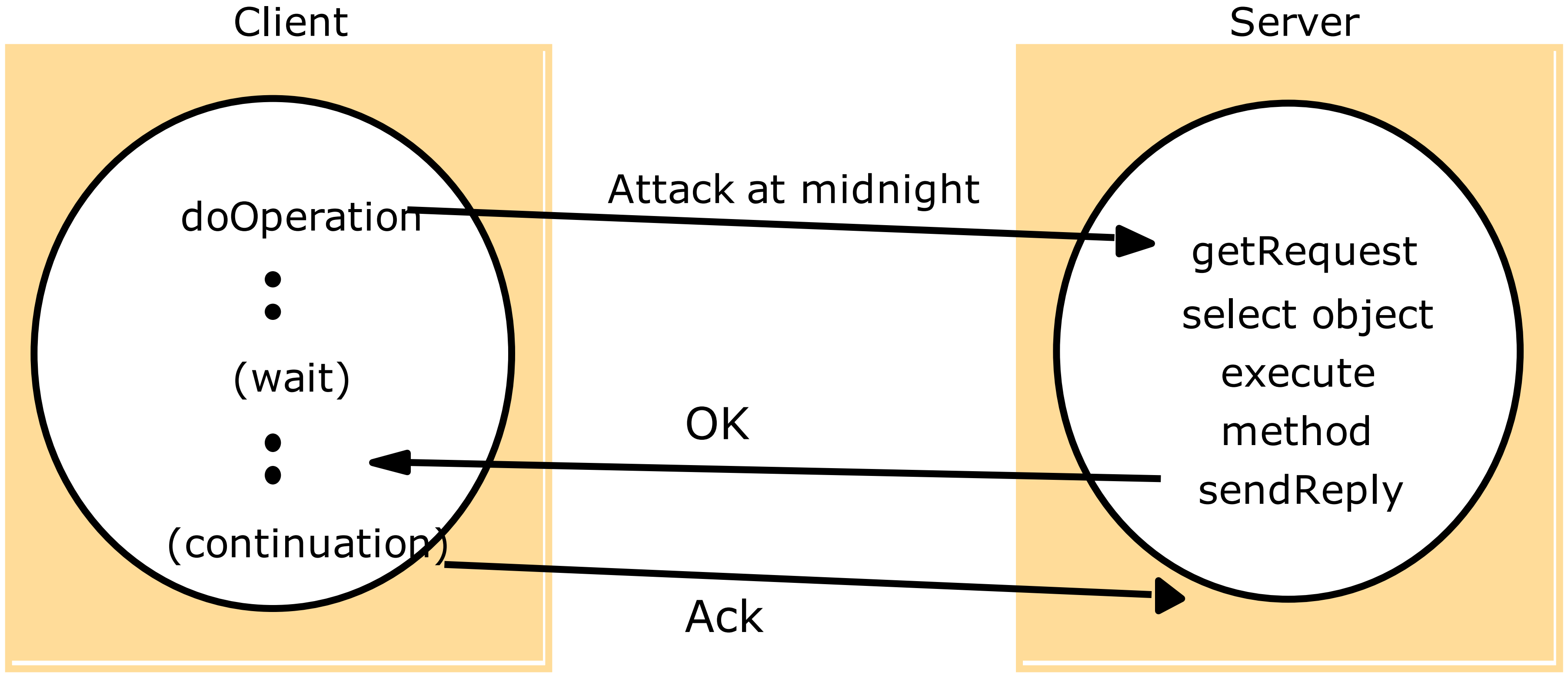
UDP Style Request-Reply Communication and the Two Army Problem (1)



Why does this not work?

The server is not sure the client received the message.

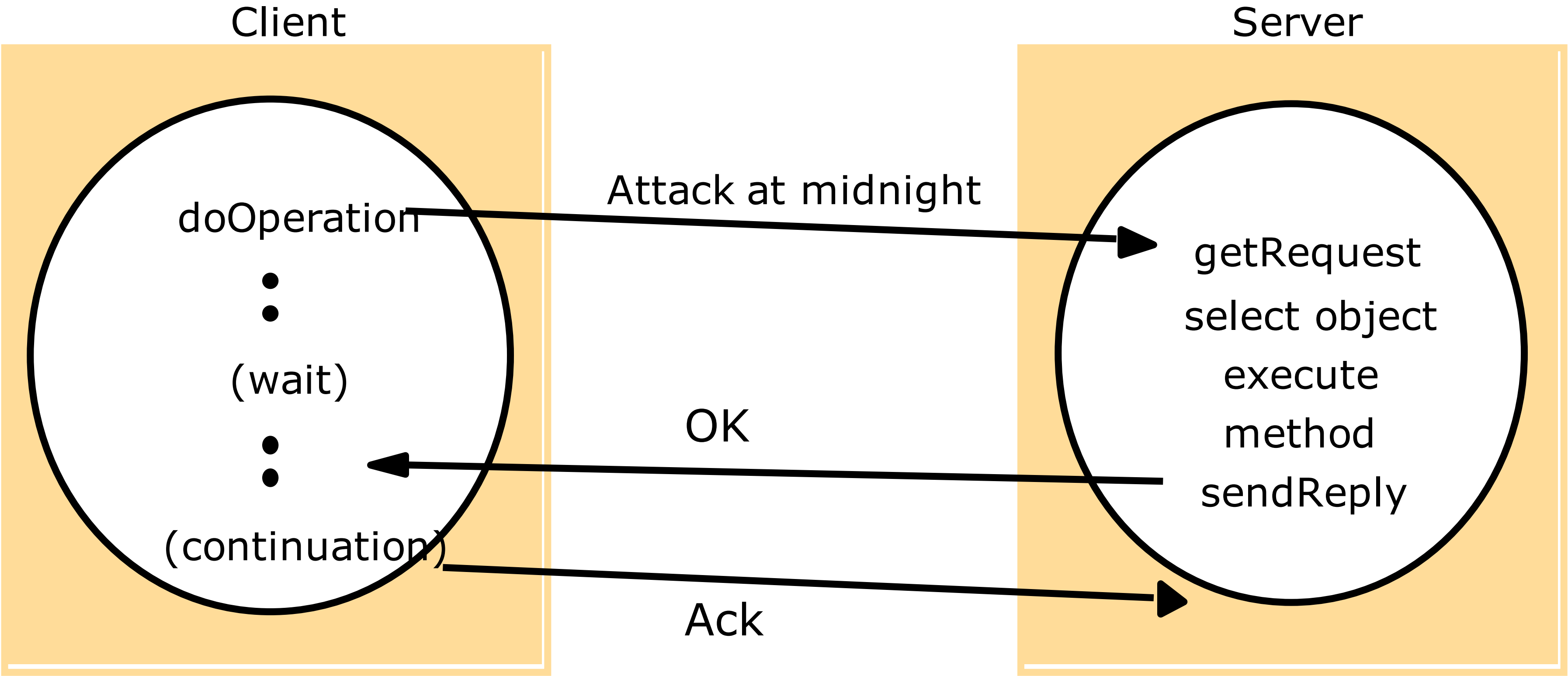
UDP Style Request-Reply Communication and the Two Army Problem (2)



Why does this not work?

The client is not sure the server received the Ack.

UDP Style Request-Reply Communication and the Two Army Problem (3)

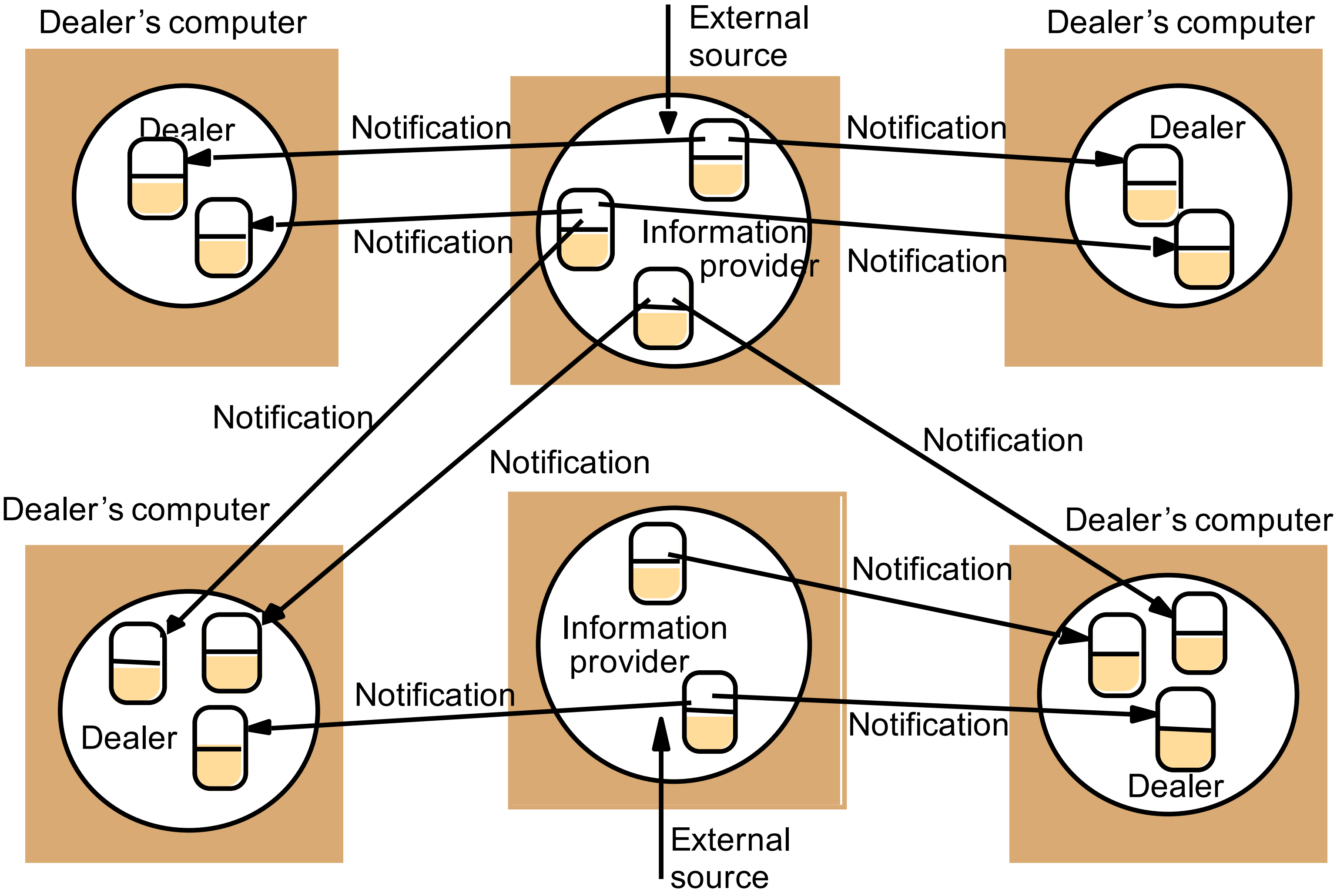


Consider a minimum sized protocol that solves this problem. If its last message is lost then agreement has not been achieved or (if it has) this must not be a minimum sized protocol.

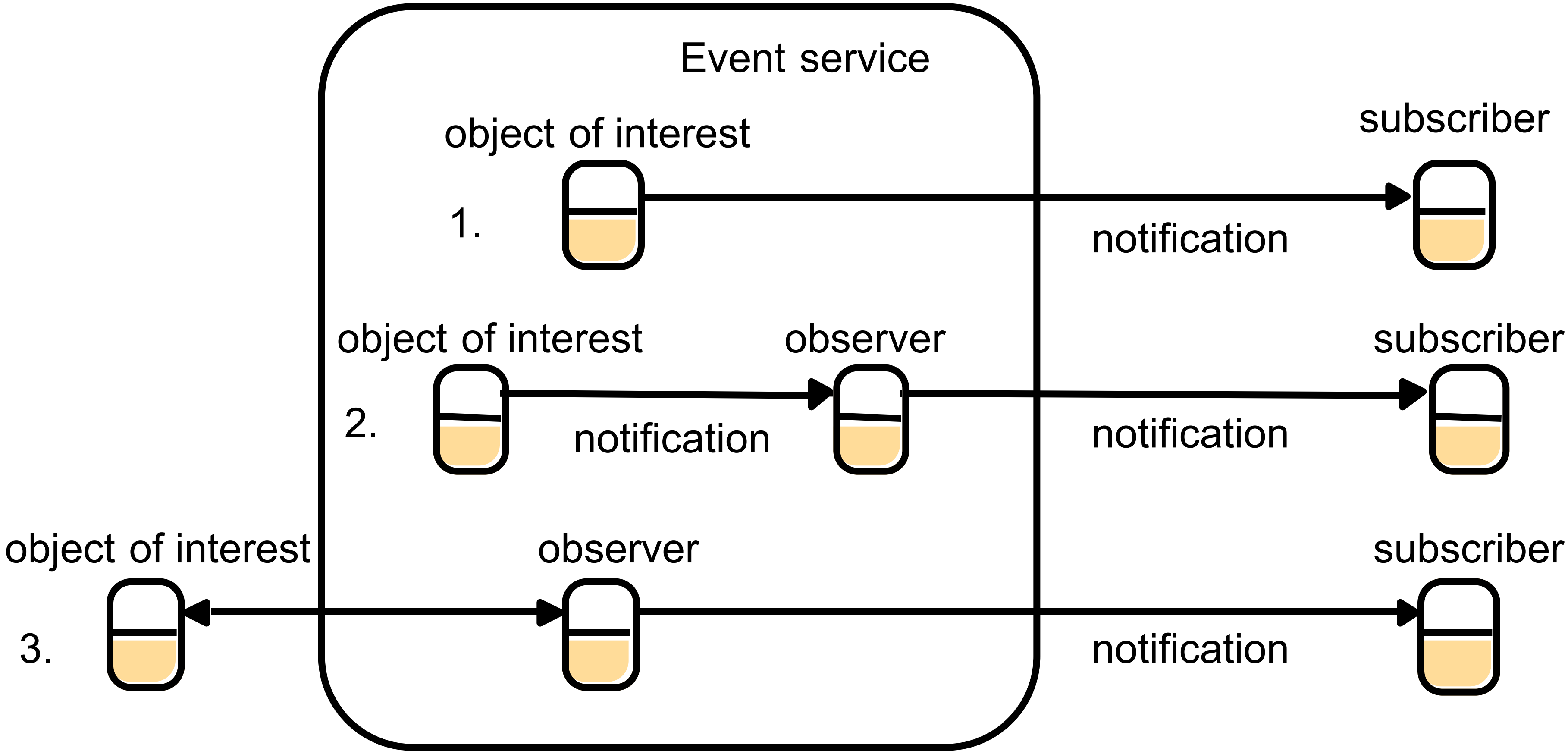
Review: What is an IDL?

- Definition: An interface definition language (IDL) provides a notation for defining interfaces in which each of the parameters of a method may be described as for input or output in addition to having its type specified.
- These may be used to allow objects written in different languages to invoke one another. Think Code Generation
- In Java RMI, we use a Java interface.
- See also WSDL, WADL, and AIDL
- Does REST need an IDL?

How would you build this?



Alternative architectures for distributed event notification



What are the goals of Java RMI?

- Distributed Java
- Almost the same syntax and semantics used by non-distributed applications
- Promote separation of concerns – the implementation is separate from the interface
- The transport layer is TCP/IP

The traditional object model (OOP 101)

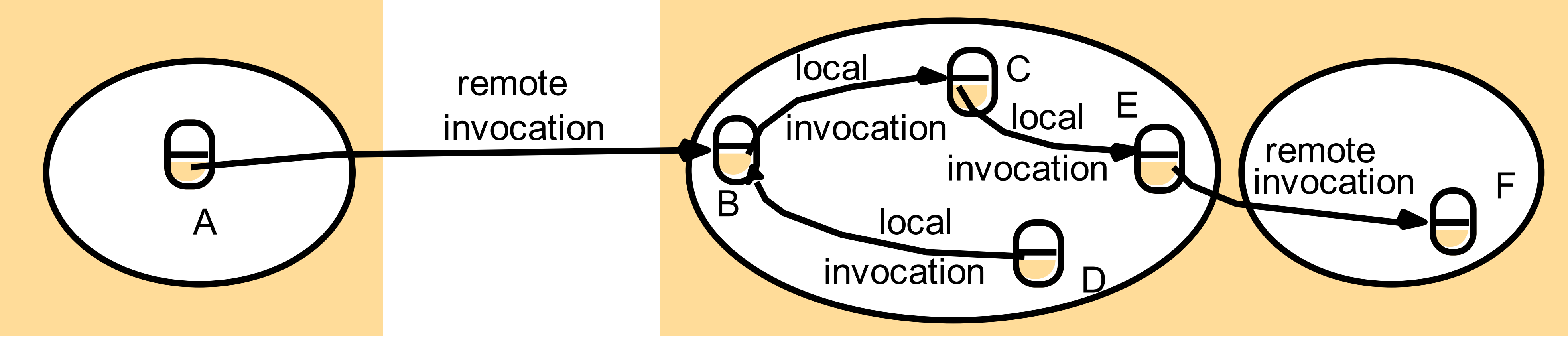
- Each object is a set of data and a set of methods.
- Object references are assigned to variables.
- Interfaces define an object’s methods.
- Actions are initiated by invoking methods.
- Exceptions may be thrown for unexpected or illegal conditions.
- Garbage collection may be handled by the developer (C++) or by the runtime (.NET and Java).
- We have inheritance and polymorphism.
- We want similar features in the distributed case

The distributed object model

- Having client and server objects in different processes enforces encapsulation. You must call a method to change its state.
- Methods may be synchronized to protect against conflicting access by multiple clients.
- Objects are accessed remotely through RMI or objects are copied to the local machine (if the object's class is available locally) and used locally.
- Remote object references are analogous to local ones in that:

  1. The invoker uses the remote object reference to identify the object and
  2. The remote object reference may be passed as an argument to or return value from a local or remote method.

We may use both remote and local method invocations

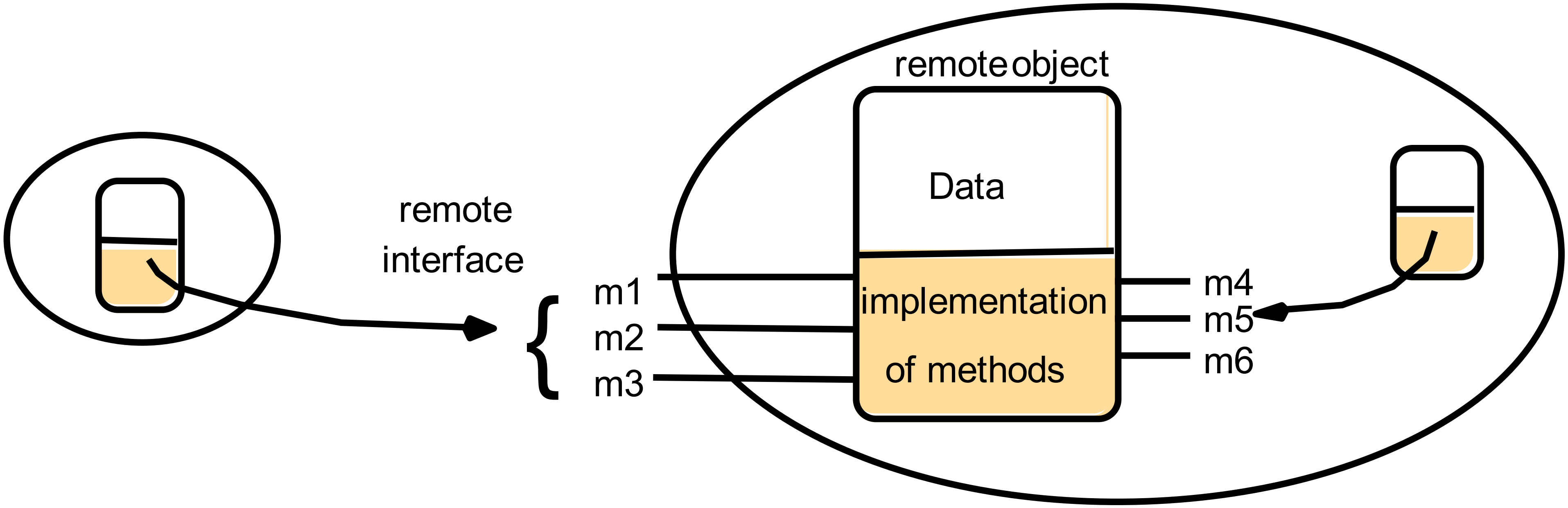


Level of transparency issues:

Remote calls should have a syntax that is close to local calls.

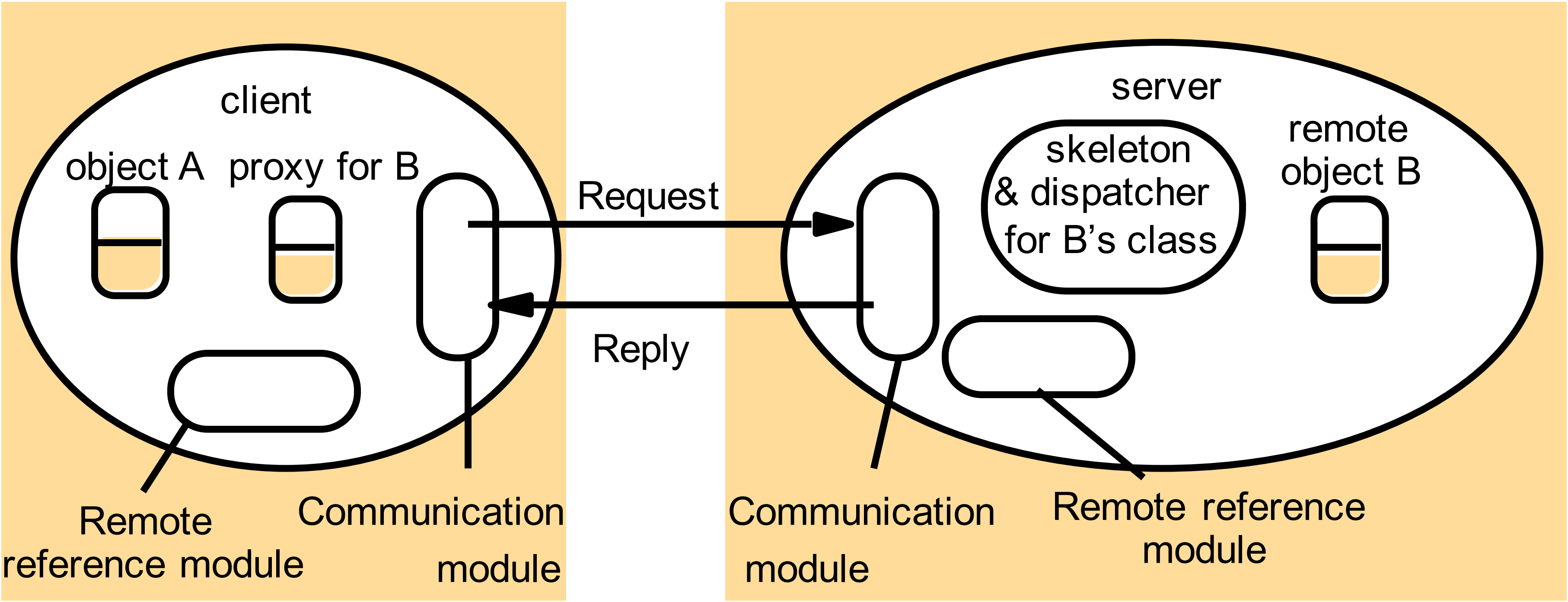
But it should probably be clear to the programmer that a remote call is being made. Remote calls are orders of magnitude slower.

A remote object and its remote interface



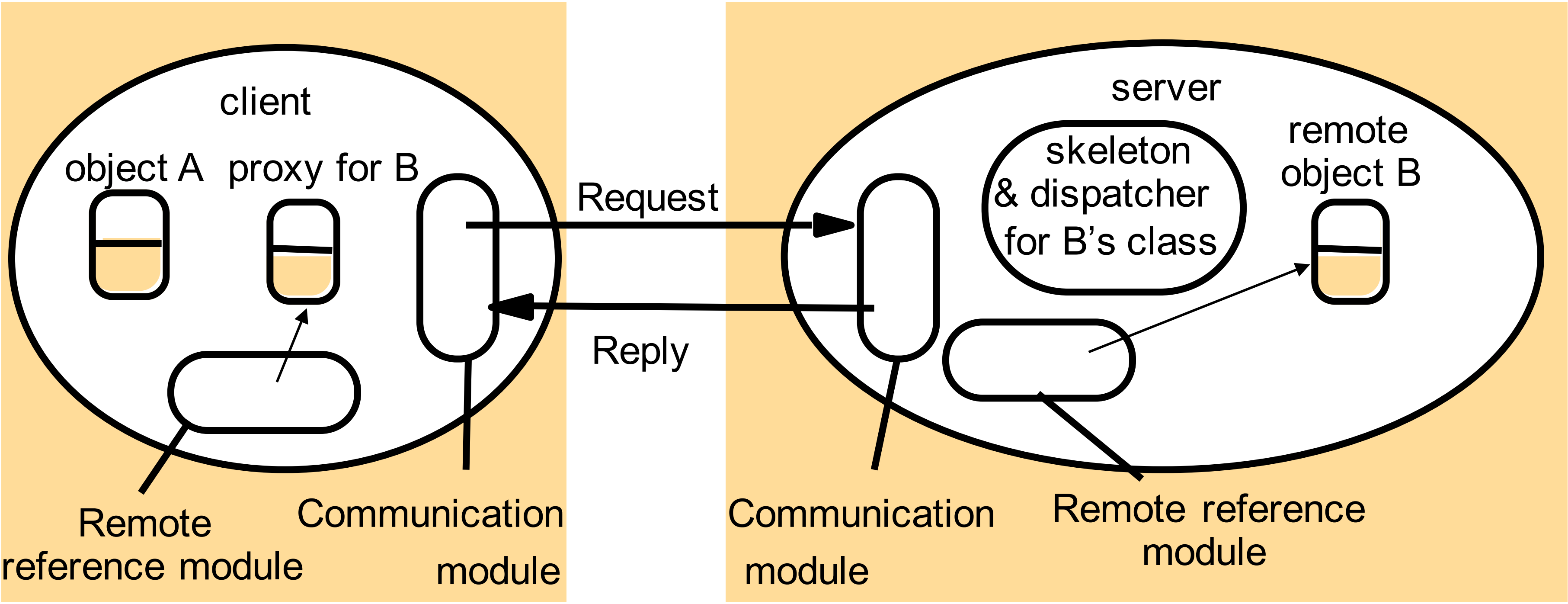
Enterprise Java Beans (EJB's) provide a remote and local interface.
EJB's are a component based middleware technology. EJB's live in a container. The container provides a managed server side hosting environment ensuring that non-functional properties are achieved.
Middleware supporting the container pattern is called an application server.
Quiz: Describe some non-functional concerns that would be handled.
Java RMI presents us with plain old distributed objects.
Fowler's First Law of Distributed Objects: "Don't distribute your objects".

Generic RMI


client
object A
proxy for B
Request
Reply
server
skeleton
& dispatcher
for B's class
remote
object B
Remote
reference module
Communication
module
Communication
module
Remote reference
module

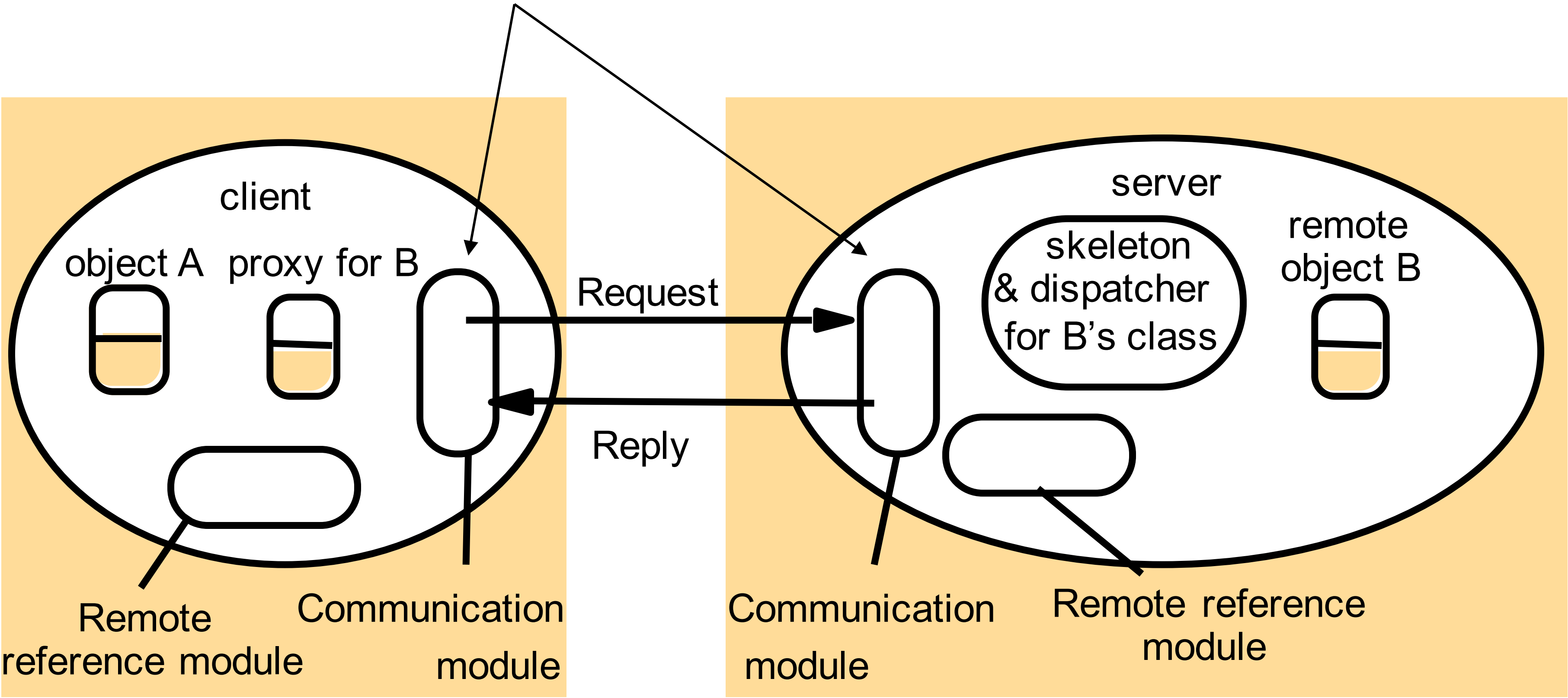
Generic remote reference module

The remote reference module holds a table that records the correspondence between local object references in that process and remote object references (which are system wide).



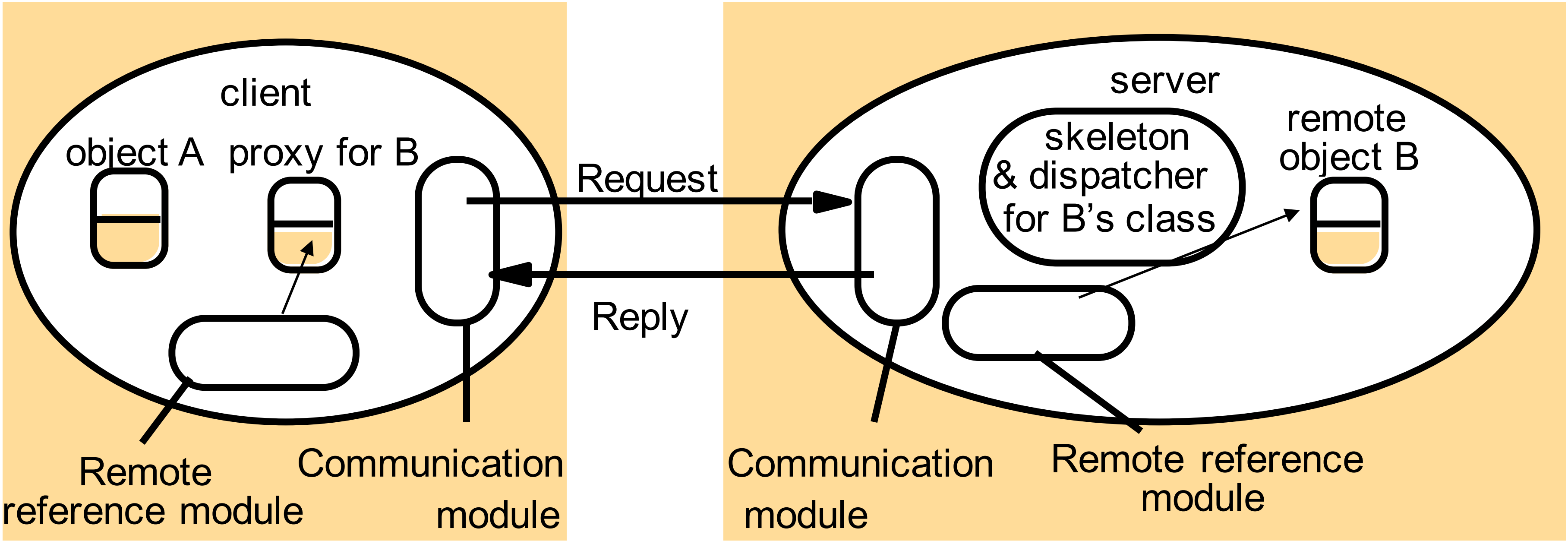
Generic communication module

Coordinate to provide a specified invocation semantics. On the server side, the communication module selects the dispatcher for the class of the object to be invoked, passing on the remote object' s local reference.



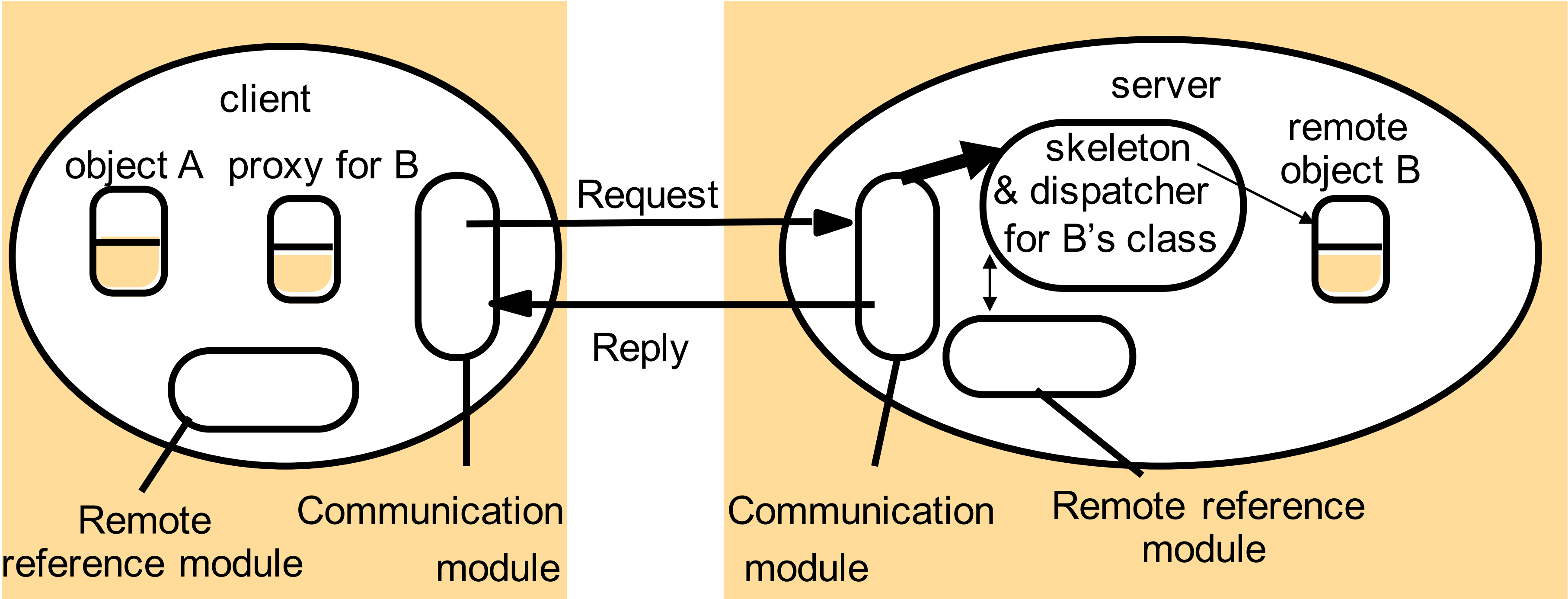
Generic proxies

On the client, the proxy makes the RMI transparent to the caller. It marshals and unmarshals parameters. There is one proxy for each remote object. Proxies hold the remote object reference. A proxy may be automatically created from the IDL.



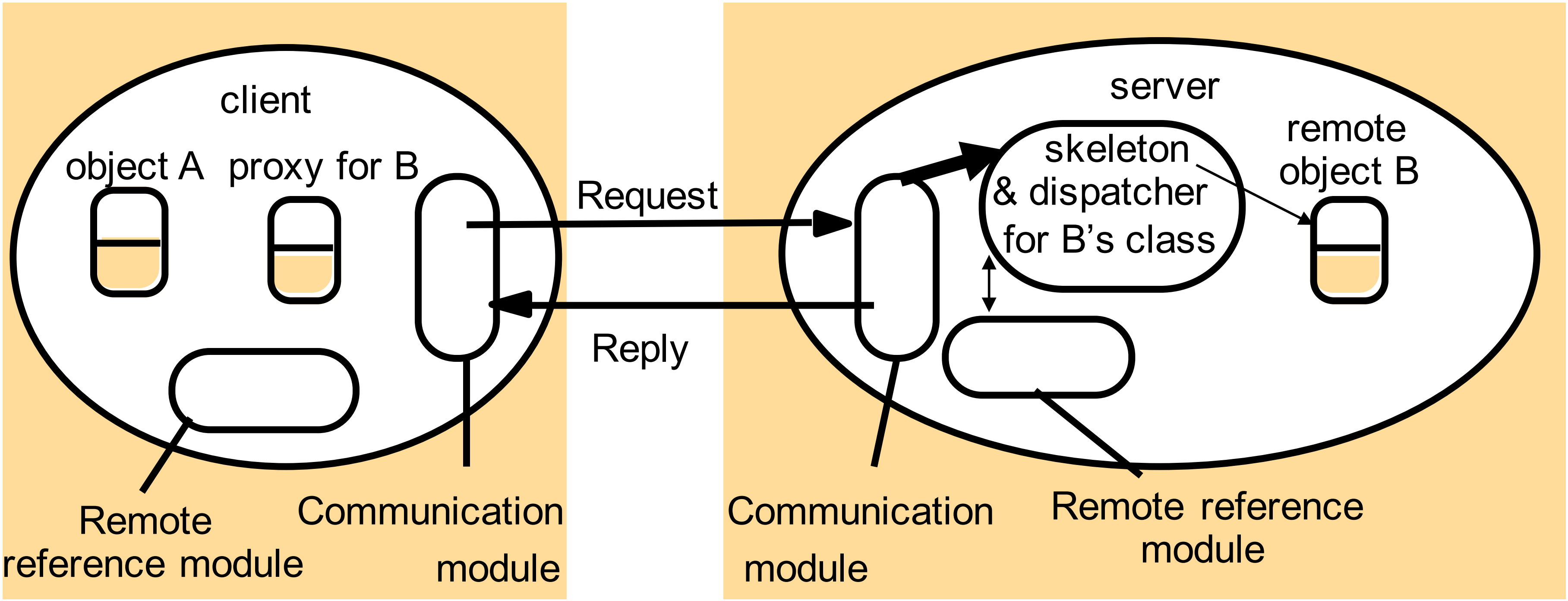
Generic dispatchers and skeletons (1)

The server has one dispatcher and skeleton for each class representing a remote object. A request message with a methodID is passed from the communication module. The dispatcher calls the method in the skeleton passing the request message. The skeleton implements the remote object’s interface in much the same way that a proxy does. The remote reference module may be asked for the local location associated with the remote reference.



Generic dispatchers and skeletons (2)

The communication module selects the dispatcher based upon the remote object reference. The dispatcher selects the method to call in the skeleton. The skeleton unmarshalls parameters and calls the method in the remote object.

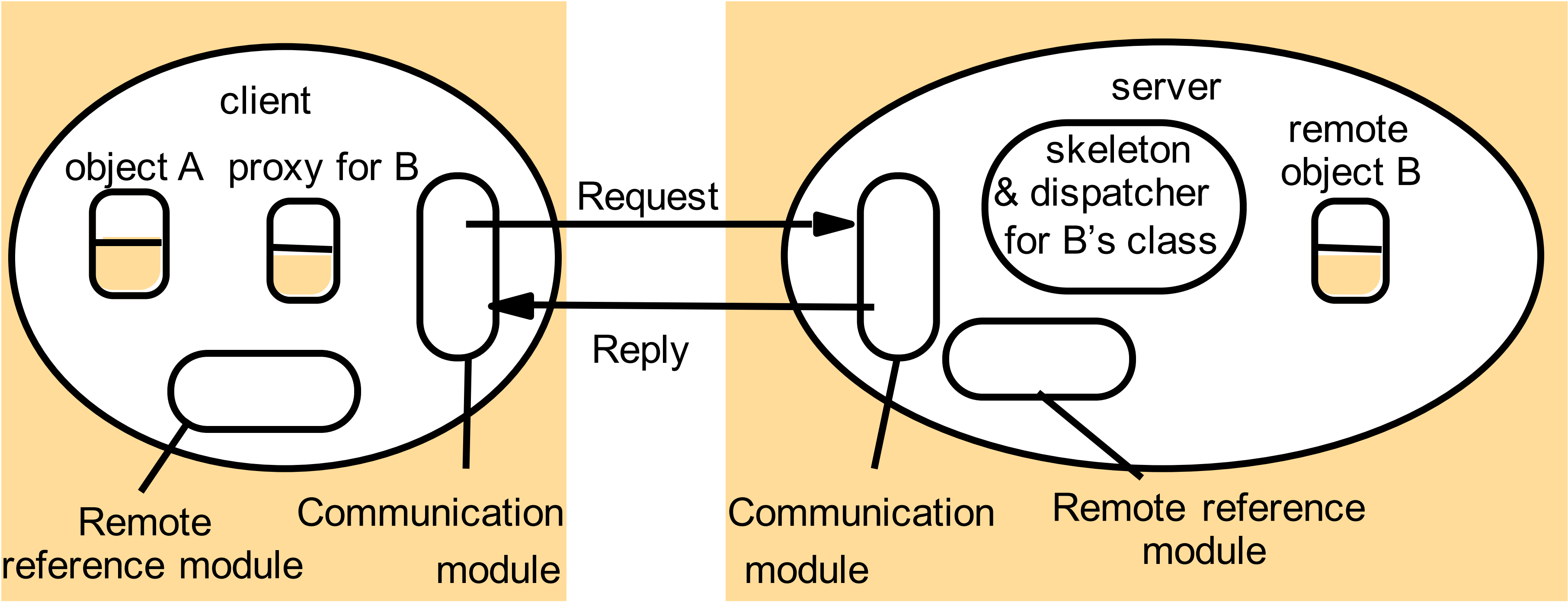


Registries promote space decoupling

Java uses the rmiregistry

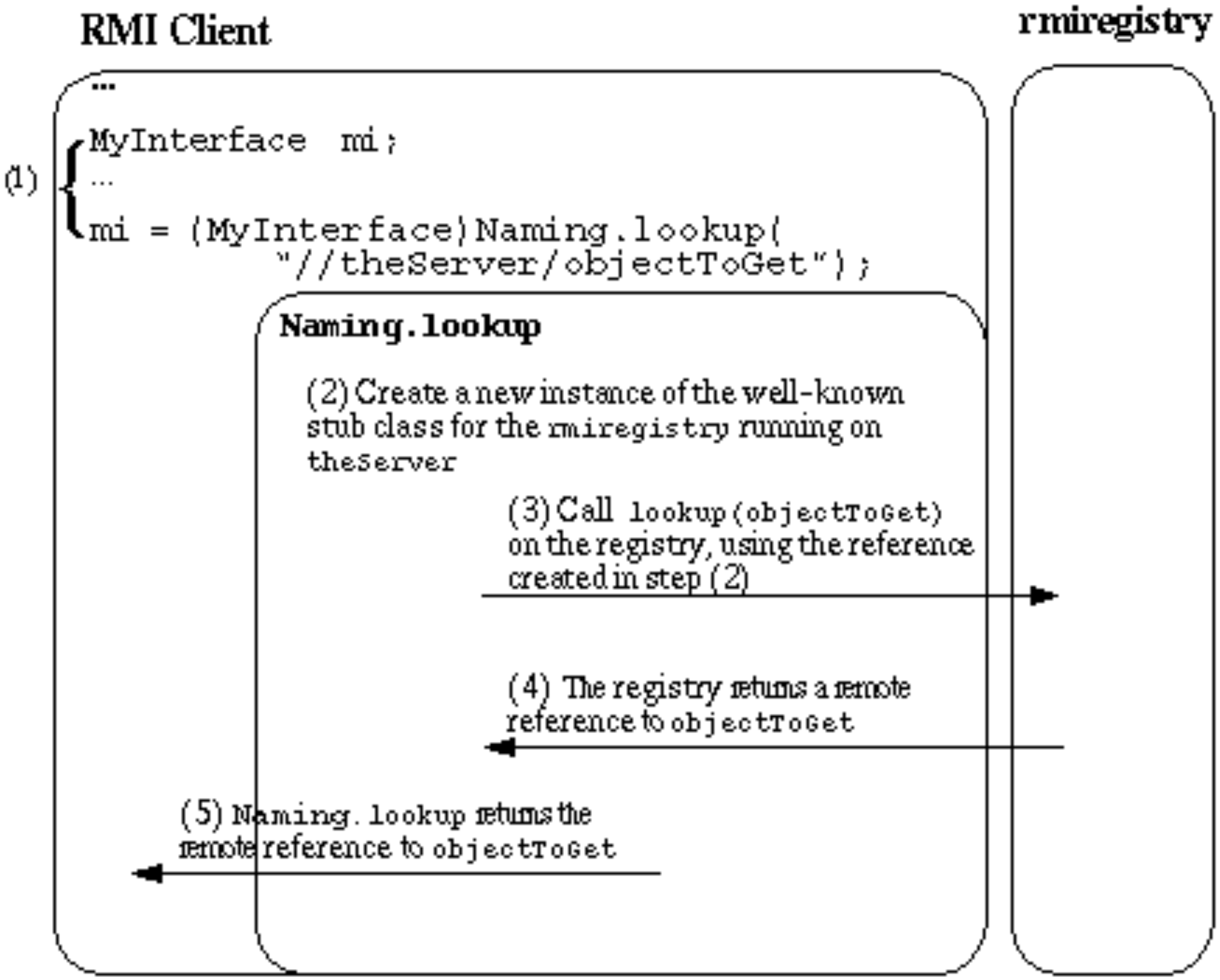
CORBA uses the CORBA Naming Service

Binders allow an object to be named and registered.



Before interacting with the remote object, the RMI registry is used.

Registries promote space decoupling



```
...
MyInterface  mi;
...
mi = (MyInterface)Naming.lookup(
        "//theServer/objectToGet");
```

(2) Create a new instance of the well-known stub class for the rmiregistry running on theServer

(3) Call lookup(objectToGet) on the registry, using the reference created in step (2)

(4) The registry returns a remote reference to objectToGet

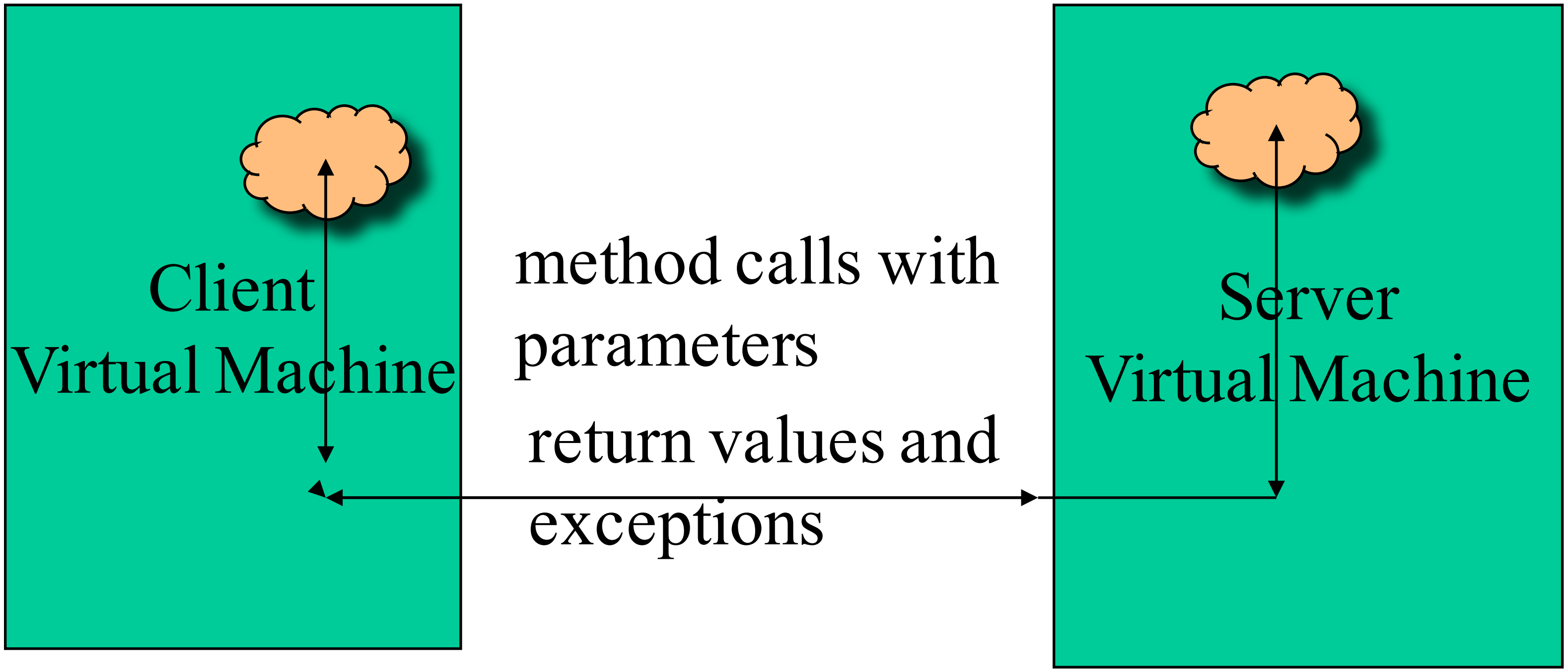
(5) Naming.lookup returns the remote reference to objectToGet

Slide from Sun Microsystems Java RMI documentation

Java RMI

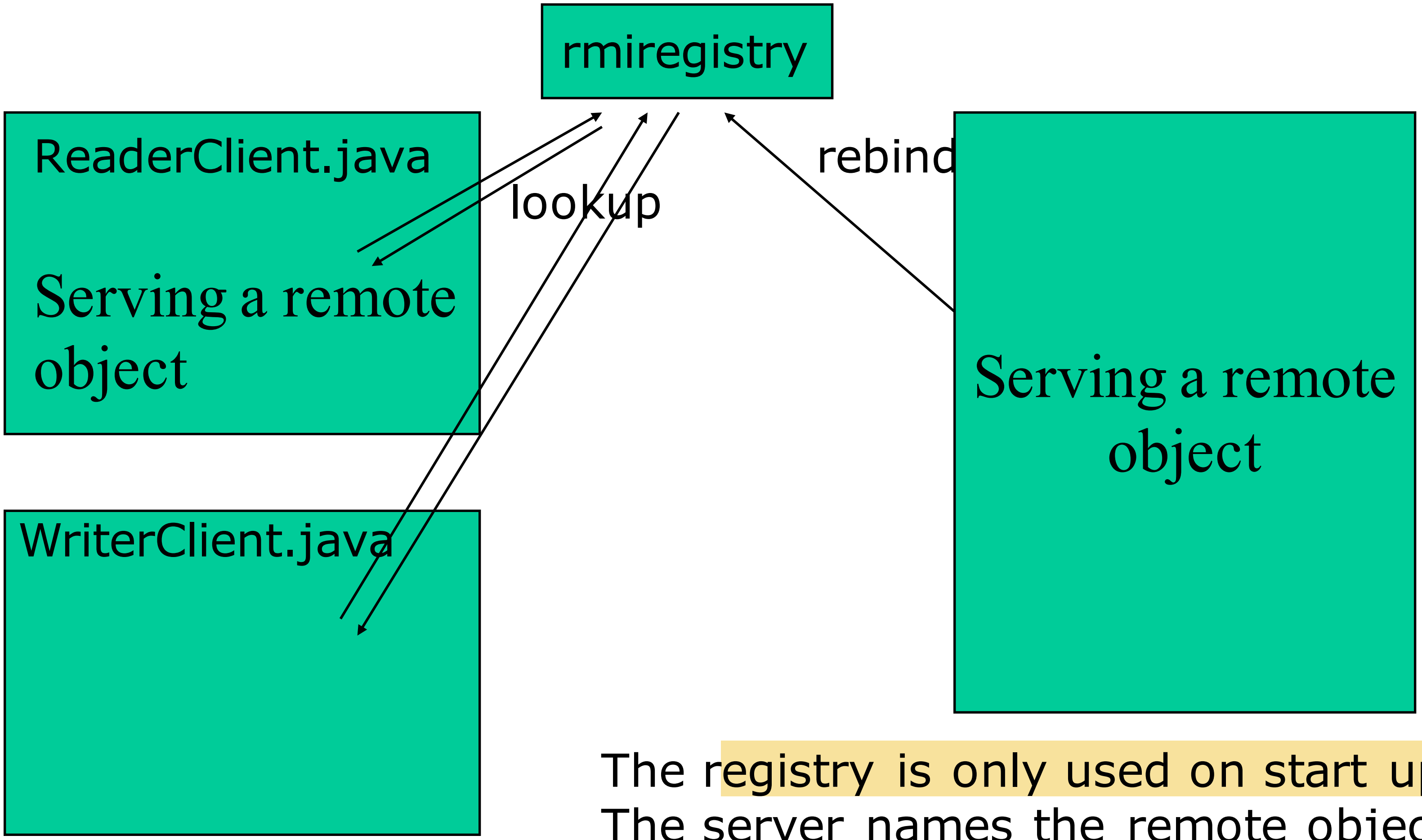
- On the client side, the RMI Registry is accessed through the static class Naming. It provides the method lookup() that a client uses to query a registry.
- The registry is not the only source of remote object references. A remote method may return a remote reference.
- The registry returns references when given a registered name. It may also return stubs to the client. You don’t see the stubs in recent editions of Java.

Java RMI



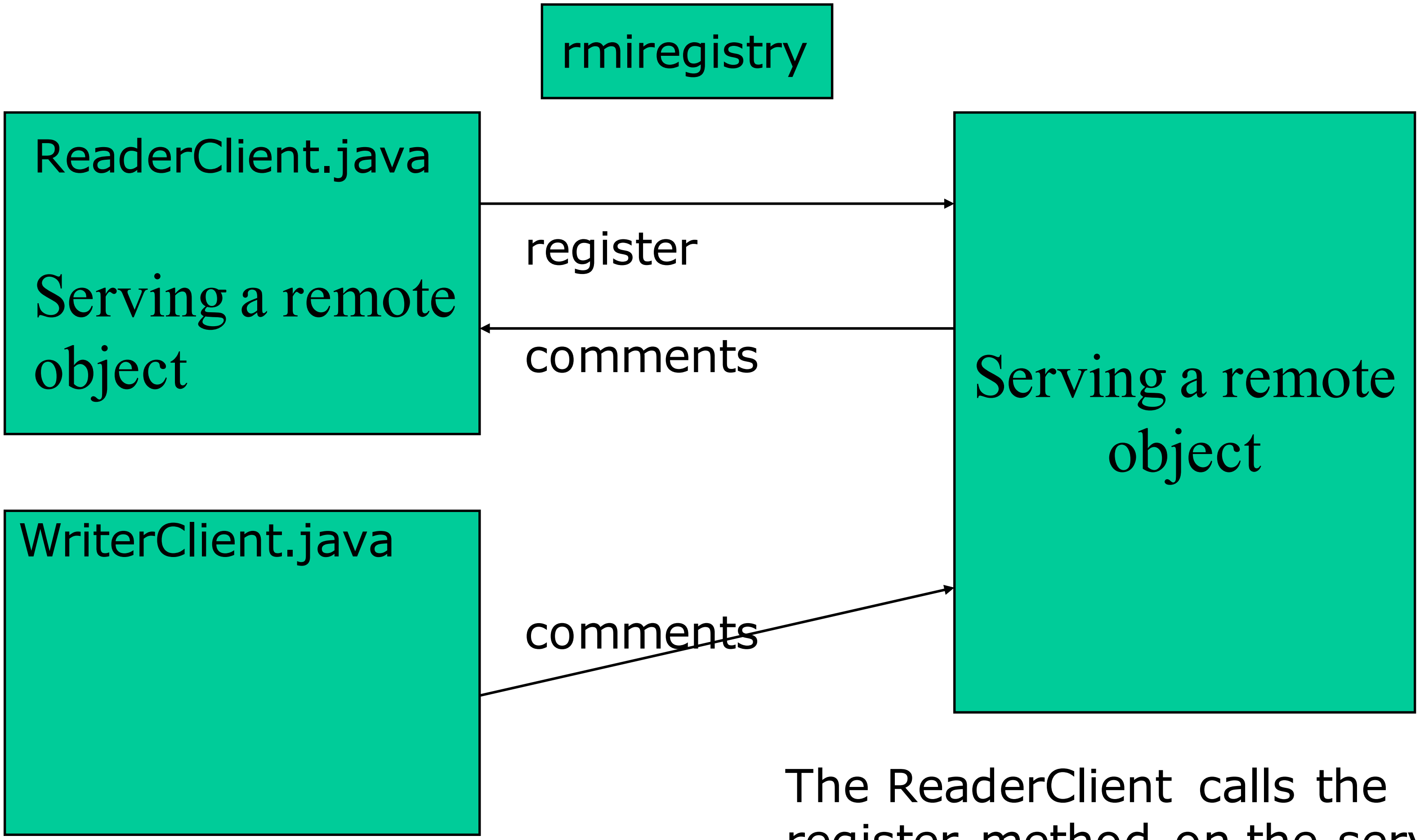
The roles of client and server only apply to a single method call. It is entirely possible for the roles to be reversed. Before this interaction, the registry is queried for the remote reference.

Example: Asynchronous Chat (1)



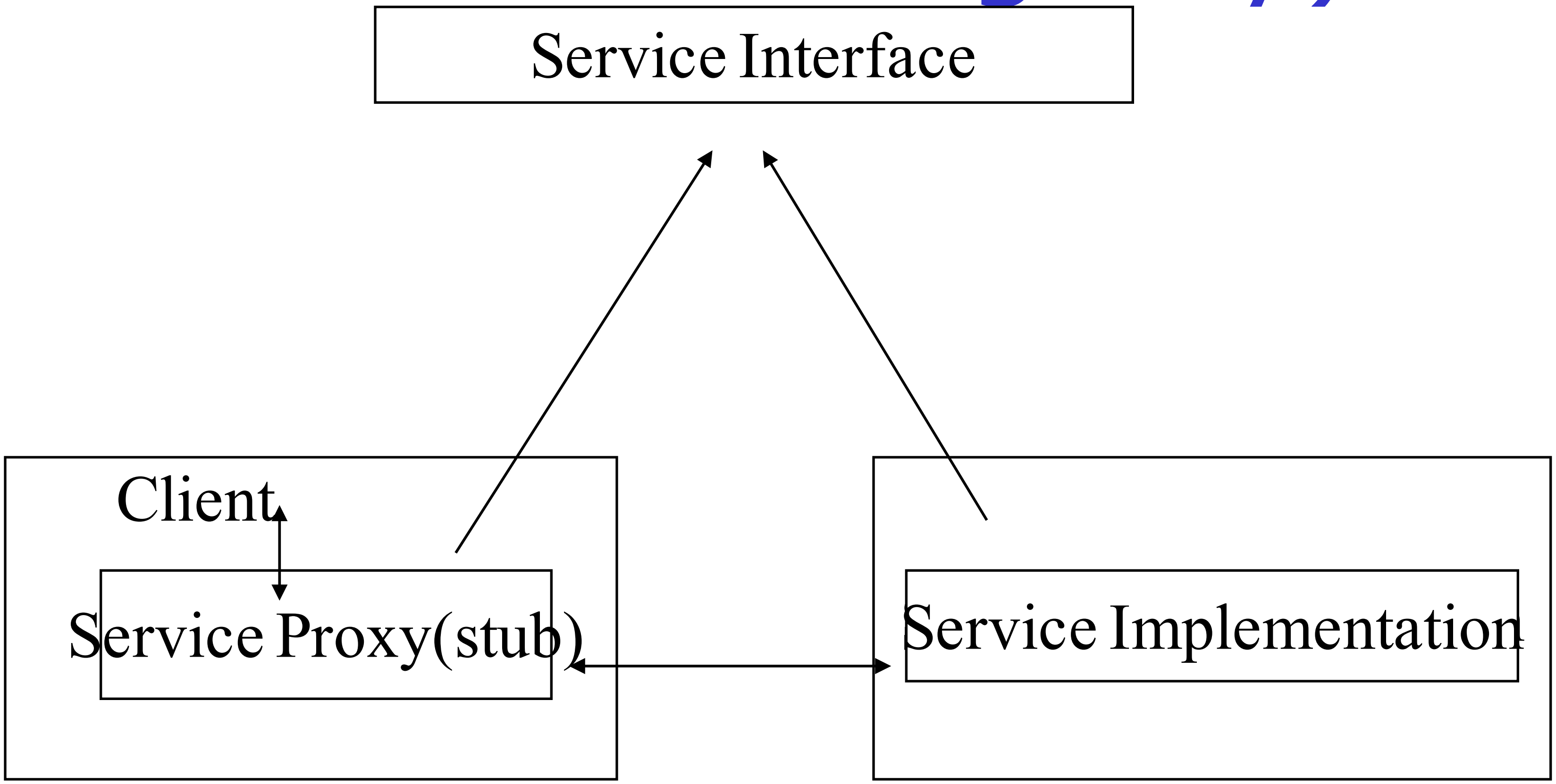
The registry is only used on start up. The server names the remote object and each type of client does a single lookup.

Example: Asynchronous Chat (2)



The ReaderClient calls the register method on the server side remote object. It passes a remote object reference.

The proxy design pattern (used to talk to the service and the registry)



A simple Java RMI client

```
import java.rmi.*;

public class ProductClient {

  public static void main(String args[]) {

        System.setSecurityManager( new RMISecurityManager());

        String url = "rmi://localhost/";
try {       // get remote references
            Product c1 = (Product)Naming.lookup(url + "toaster");
            Product c2 = (Product)Naming.lookup(url + "microwave");
            // make calls on local stubs
            // get two String objects from server
            System.out.println(c1.getDescription());
            System.out.println(c2.getDescription());
    }
    catch( Exception e) {

        System.out.println("Error " + e);

    }
    System.out.println.exit(0);
  }
}
```

Note: In this example, the server passes two String objects to the client. The registry passes a remote object reference to the client.

Notes about the client(1)

- The default behavior when running a Java application is that no security manager is installed. A Java application can read and write files, open sockets, start print jobs and so on.

- Applets, on the other hand, immediately install a security manager that is quite restrictive.

- A security manager may be installed with a call to the static setSecurityManager method in the System class.

Notes about the client(2)

- Any time you load code from another source (as this client might be doing by dynamically downloading the stub class), it's wise to use a security manager.

- By default, the RMISecurityManager restricts all code in the program from establishing network connections. But, this program needs network connections.

  -- to reach the RMI registry
  -- to contact the server objects

- So, Java requires that we inform the security manager through a policy file.

Notes about the client(3)

- The Naming class provides methods for storing and obtaining references to remote objects in the remote object registry.
- Callers on a remote (or local) host can lookup the remote object by name, obtain its reference, and then invoke remote methods on the object.
- lookup is a static method of the Naming class that returns a reference to an object that implements the remote interface. Its single parameter contains a URL and the name of the object.

Notes about the client(4)

The object references c1 and c2 do not actually refer to objects on the server. Instead, these references refer to a stub class that must exist on the client. The stub holds the remote object reference.

```
Product c1 = (Product)Naming.lookup(url + "toaster");
Product c2 = (Product)Naming.lookup(url + "microwave");
```

The stub class is in charge of object serialization and transmission. it’ s the stub object that actually gets called by the client with the line

```
System.out.println(c1.getDescription());
```

File client.policy

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

This policy file allows an application to make any network connection to a port with port number at least 1024. (The RMI port is 1099 by default, and the server objects also use ports >= 1024.)

Notes About the client(5)

When running the client, we must set a system property that describes where we have stored the policy.

```
javac ProductClient.java
java –Djava.security.policy=client.policy ProductClient
```

Files on the Server Product.java

```
// Product.java is also available on the client. Why?

import java.rmi.*;

public interface Product extends Remote {

    String getDescription() throws RemoteException;

}
```

Notes on Product Interface

- This interface must reside on both the client and the server. Both need the interface before compilation. The use of interfaces promotes separation of concerns.
- All interfaces for remote objects must extend remote.
- Each method requires the caller to handle a RemoteException (if any network problems occur).

Files on the Server
ProductImpl.java

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

public class ProductImpl extends UnicastRemoteObject
                              implements  Product {
     private String name;

     public ProductImpl(String n) throws RemoteException  {
          name = n;
     }
     public String getDescription()  throws RemoteException  {
          return  "I am a " + name + ". Buy me!";
     }
}
```

Notes on ProductImpl.java

- This file resides on the server.

- It is used to automatically generate the stub class that is required by the client. In order to create such a stub class we can use the rmic program on the server:

```
javac ProductImpl.java
rmic –v1.2 ProductImpl
```

- This creates the file ProductImpl_Stub.class (skeleton classes are no longer needed in JDK1.2)

Files on the server ProductServer.java

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

public class ProductServer {

    public static void main(String args[]) {

        try {
            System.out.println("Constructing server implementations...");
            ProductImpl p1 = new ProductImpl("Blackwell Toaster");
            ProductImpl p2 = new ProductImpl("ZapXpress Microwave")
System.out.println("Binding server implementations to registry...");

Naming.rebind("toaster", p1);

Naming.rebind("microwave",p2);

System.out.println("Waiting for invocations from clients...");

}
        catch(Exception e) {

          System.out.println("Error: " + e);
        }
   }
}
```

Notes on the ProductServer.java

- The server program registers objects with the bootstrap registry service, and the client retrieves stubs to those objects.

- You register a server object by giving the bootstrap registry service a reference to the object and a unique name.

```
ProductImpl p1 = new ProductImpl(“Blackwell Toaster”);
Naming.rebind("toaster", p1);
```

Summary of Activities

1. Compile the java files:

   javac *.java

2. Run rmic on the ProductImpl.class producing the file ProductImpl_Stub.class (Now, optional)

   rmic –v1.2 ProductImpl

3. Start the RMI registry

   start rmiregistry

4. Start the server

   start java ProductServer

5. Run the client

   java –Djava.security.policy=client.policy ProductClient

Parameter passing in remote methods

When a remote object is passed from the server, the client receives a stub (or already has one locally):

```
    Product c1 = (Product)Naming.lookup(url + "toaster");
```

Using the stub, it can manipulate the server object by invoking remote methods. The object, however, remains on the server.

Parameter passing in remote methods

It is also possible to pass and return *any* objects via a remote method call, not just those that implement the remote interface.

The method call

c1.getDescription()

returned a full blown String object to the client. This then became the client' s String object. It has been copied via Java serialization.

Parameter passing in remote methods

This differs from local method calls where we pass and return references to objects.

To summarize, remote objects are passed across the network as stubs (remote references). Nonremote objects are copied.

Whenever code calls a remote method, the stub makes a package that contains copies of all parameter values and sends it to the server, using the object serialization mechanism to marshall the parameters.

Quiz

- What is a proxy?
- What is an interface?
- What is an implementation?
- What must the client code have at runtime?
- What must the client code have at compile time?

Java RMI Example 2 - RMI Whiteboard

- See Coulouris Text
- Client and Server code stored in separate directories
- Stub code available to client and server (in their classpaths) and so no need for RMISecurity Manager
- All classes and interfaces available to both sides

Client Directory

GraphicalObject.class
GraphicalObject.java
Shape.class
Shape.java
ShapeList.class
ShapeList.java
ShapeListClient.class
ShapeListClient.java
ShapeListServant_Stub.class
ShapeServant_Stub.class

Client side steps

The stub classes were created on the server side and copied to the client

javac *.java
java ShapeListClient

Server Directory

```
GraphicalObject.class
GraphicalObject.java
Shape.class
Shape.java
ShapeList.class
ShapeList.java
ShapeListServant.class
ShapeListServant.java
ShapeListServant_
Stub.class
ShapeListServer.class
ShapeListServer.java
ShapeServant.class
ShapeServant.java
ShapeServant_Stub.class
```

Server side steps

```
javac *.java
rmic –V1.2 ShapeServant
rmic –V1.2 ShapeListServant
copy stubs to client
start rmiregistry
java ShapeListServer
```

GraphicalObject.java

```
// GraphicalObject.java
// Holds information on a Graphical shape

import java.awt.Rectangle;
import java.awt.Color;
import java.io.Serializable;

public class GraphicalObject implements Serializable{

    public String type;
    public Rectangle enclosing;
    public Color line;
    public Color fill;
    public boolean isFilled;
//  constructors
    public GraphicalObject() { }

    public GraphicalObject(String aType, Rectangle anEnclosing,
    Color aLine,Color aFill, boolean anIsFilled) {
            type = aType;
            enclosing = anEnclosing;
            line = aLine;
            fill = aFill;
            isFilled = anIsFilled;
    }

    public void print(){
            System.out.print(type);
            System.out.print(enclosing.x + " , " + enclosing.y + " , "
    + enclosing.width + " , " + enclosing.height);
            if(isFilled) System.out.println("- filled");else
    System.out.println("not filled");
    }
}
```

Shape.java

```
// Shape.java
// Interface for a Shape

import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {

   int getVersion() throws RemoteException;

   GraphicalObject getAllState() throws RemoteException;

}
```

ShapeServant.java

```
// ShapeServant.java
// Remote object that wraps a Shape

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ShapeServant extends UnicastRemoteObject implements
     Shape {

    int myVersion;
    GraphicalObject theG;

    public ShapeServant(GraphicalObject g, int version)throws
     RemoteException{
     theG = g;
     myVersion = version;
    }
public int getVersion() throws RemoteException {
            return myVersion;
   }

   public GraphicalObject  getAllState() throws RemoteException{
      return theG;
  }
}
```

ShapeList.java

```
// ShapeList.java
// Interface for a list of Shapes

import java.rmi.*;
import java.util.Vector;

public interface ShapeList extends Remote {

        Shape newShape(GraphicalObject g) throws RemoteException;
        Vector allShapes()throws RemoteException;
        int getVersion() throws RemoteException;
}
```

ShapeListServant.java

```
// ShapeList.java
// Remote Object that implements ShapeList

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ShapeListServant extends UnicastRemoteObject
                     implements ShapeList{
```

```
 private Vector theList;
 private int version;

 public ShapeListServant()throws RemoteException{
    theList = new Vector();
    version = 0;
 }

public Shape newShape(GraphicalObject g) throws
                RemoteException{
    version++;
       Shape s = new ShapeServant( g, version);
    theList.addElement(s);
    return s;
  }
```

```
    public   Vector allShapes() throws RemoteException{
          return theList;
    }

    public int getVersion() throws RemoteException{
       return version;
    }
}
```

ShapeListServer.java

```
// ShapeListServer.java
// Server to install remote objects

// Assume all stubs available to client and server
// so no need to create a
// RMISecurityManager  with java.security.policy

import java.rmi.*;

public class ShapeListServer {

    public static void main(String args[]){

         System.out.println("Main  OK");
try{
        ShapeList aShapelist = new ShapeListServant();

        System.out.println("Created shape list object");
        System.out.println("Placing in registry");

        Naming.rebind("ShapeList", aShapelist);

        System.out.println("ShapeList server ready");

       }catch(Exception e) {
           System.out.println("ShapeList server main " +
                                    e.getMessage());
     }
   }
}
```

ShapeListClient.java

```
// ShapeListClient.java
// Client - Gets a list of remote shapes or adds a shape
// to the remote list

import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
import java.awt.Rectangle;
import java.awt.Color;


public class ShapeListClient{
```

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

        String option = "Read";
        String shapeType = "Rectangle";

        // read or write
        if(args.length > 0)  option = args[0];

        // specify Circle, Line etc
        if(args.length > 1)  shapeType = args[1];

        System.out.println("option = " + option +
                                  "shape = " + shapeType);
        ShapeList aShapeList = null;
```

```
try{
        aShapeList = (ShapeList)
                          Naming.lookup("//localhost/ShapeList");

        System.out.println("Found server");
```

```
Vector sList = aShapeList.allShapes();
 System.out.println("Got vector");
 if(option.equals("Read")){
           for(int i=0; i<sList.size(); i++){
                         GraphicalObject g =
                                  ((Shape)sList.elementAt(i)).getAllState();
                         g.print();
           }
  }
  else {  // write to server
          GraphicalObject g = new
                        GraphicalObject(
                                 shapeType, new Rectangle(50,50,300,400),
                                        Color.red,Color.blue, false);
                 System.out.println("Created graphical object");
                 aShapeList.newShape(g);
                 System.out.println("Stored shape");
           }
```

```
}catch(RemoteException e) {

            System.out.println("allShapes: " + e.getMessage());

            }catch(Exception e) {

            System.out.println("Lookup: " + e.getMessage());}
  }
}
```