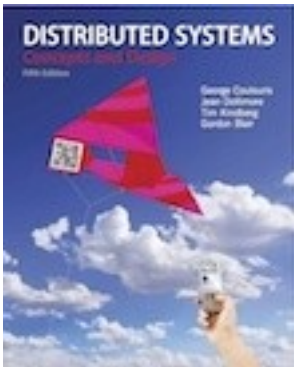


Slides for Chapter 5: Remote invocation

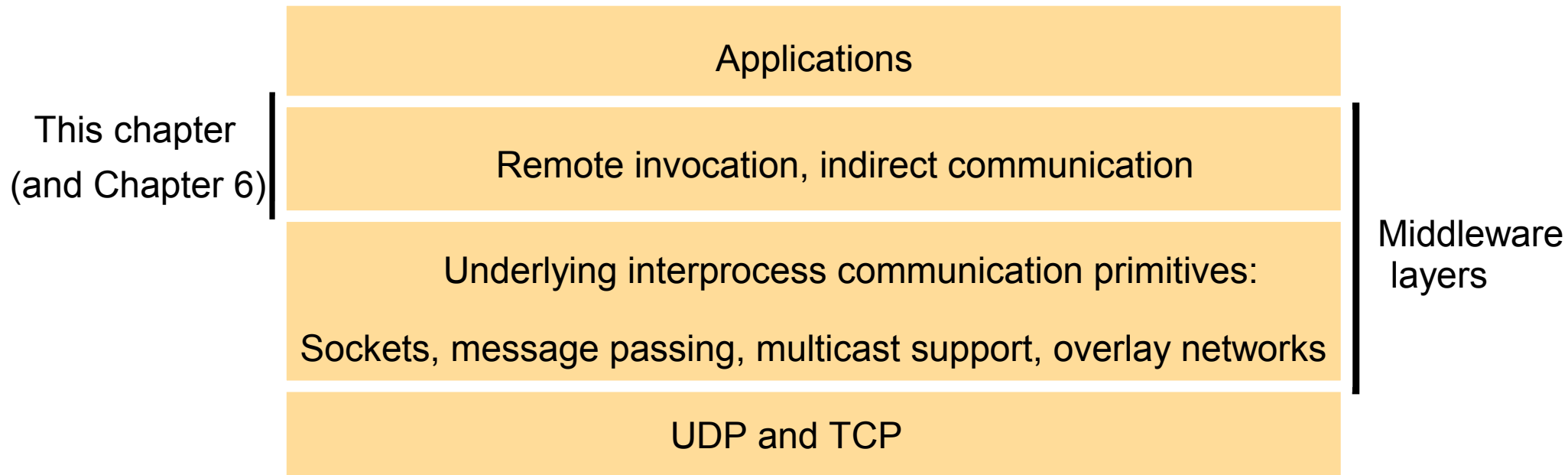


From **Coulouris, Dollimore, Kindberg and Blair**
Distributed Systems:
Concepts and Design

Edition 5, © Addison-Wesley 2012

Figure 5.1

Middleware layers

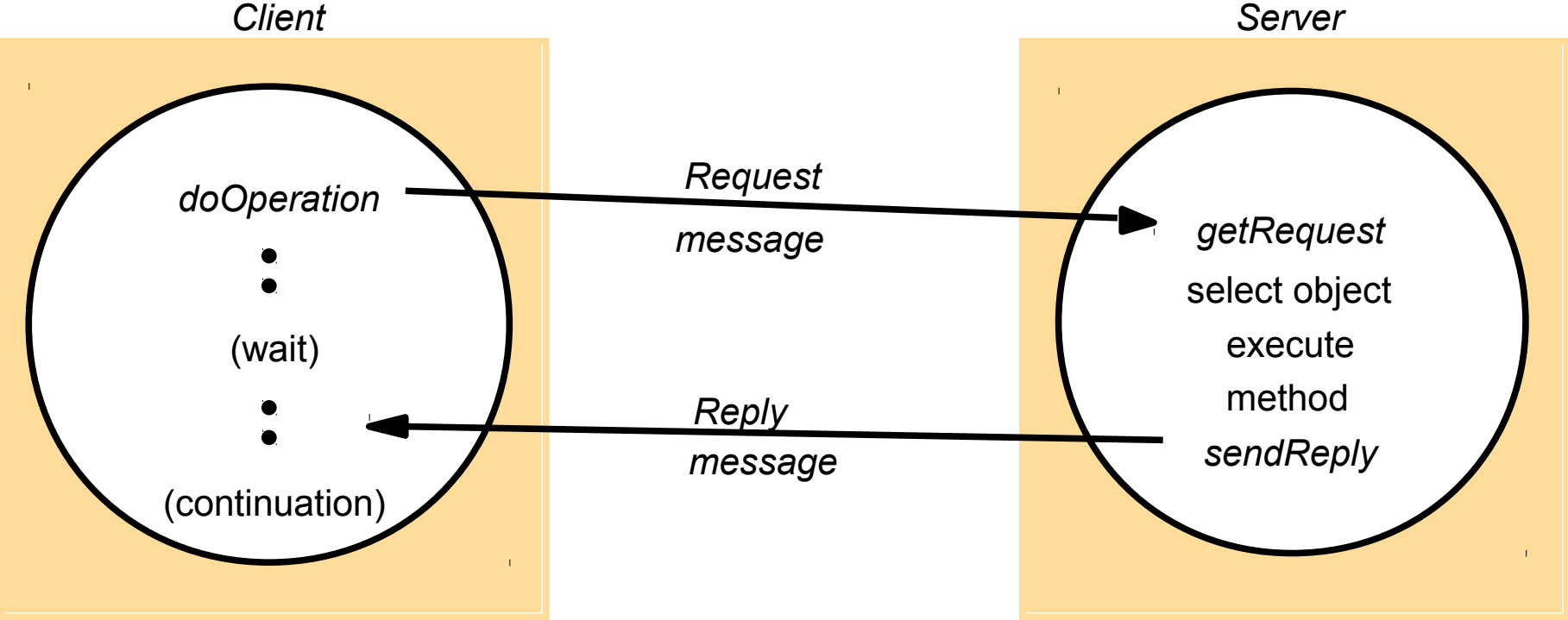


Request-reply communication

request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client.

Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later.

Figure 5.2
Request-reply communication



Operations of the request-reply protocol

The request-reply protocol:

The protocol is based on a trio of communication primitives, *doOperation*, *getRequest* and *sendReply*,

This request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.

The *doOperation* method is used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply.

getRequest is used by a server process to acquire service requests.

sendReply is used to send the reply message to the client. When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues.

Figure 5.3

Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)

sends a request message to the remote server and returns the reply.

The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

Three protocols that produce differing behaviours in the presence of communication failures are used for implementing various types of request behaviour. They were originally identified by Spector [1982]:

- the request (R) protocol;
- the request-reply (RR) protocol;
- the request-reply-acknowledge reply (RRA) protocol.

Figure 5.4
Request-reply message structure

messageType	<i>int (0=Request, 1= Reply)</i>
requestId	<i>int</i>
remoteReference	<i>RemoteRef</i>
operationId	<i>int or Operation</i>
arguments	<i>array of bytes</i>

Figure 5.5

RPC exchange protocols

<i>Name</i>	<i>Messages sent by</i>		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

HTTP

HTTP is a protocol that specifies the messages involved in a request-reply exchange, the methods, arguments and results, and the rules for representing them in the messages.

It supports a fixed set of methods (GET, PUT, POST, etc) that are applicable to all of the server's resources.


HTTP

In addition to invoking methods on web resources, the protocol allows for content negotiation and password-style authentication:

Content negotiation: Clients' requests can include information as to what data representations they can accept (for example, language or media type), enabling the server to choose the representation that is the most appropriate for the user.

Authentication: Credentials and challenges are used to support password-style authentication. On the first attempt to access a password-protected area, the server reply contains a challenge applicable to the resource. When a client receives a challenge, it gets the user to type a name and password and submits the associated credentials with subsequent requests.

Figure 5.6
HTTP request message



<i>method</i>	<i>URL or pathname</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure 5.7
HTTP *Reply* message

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

Interface definition languages IDL

An RPC mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces, allowing input and output parameters to be mapped onto the language's normal use of parameters.

This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language, for example, Java, for local and remote invocation.

However, many existing useful services are written in C++ and other languages. It would be beneficial to allow programs written in a variety of languages, including Java, to access them remotely.

Interface definition languages (IDLs) are designed **to allow procedures implemented in different languages to invoke one another.**

An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.

Figure 5.8

CORBA Interface Definition Languages (IDL) example

```
// In file Person.idl  
struct Person {  
    string name;  
    string place;  
    long year;  
};  
interface PersonList {  
    readonly attribute string listname;  
    void addPerson(in Person p) ;  
    void getPerson(in string name, out Person p);  
    long number();  
};
```

RPC call semantics

Request-reply protocols showed that doOperation can be implemented in different ways to provide different delivery guarantees.

The main choices are:

Retry request message: Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

Duplicate filtering: Controls when retransmissions are used and whether to filter out duplicate requests at the server.

Retransmission of results: Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

Figure 5.9
Call semantics

<i>Fault tolerance measures</i>			<i>Call semantics</i>
<i>Retransmit request message</i>	<i>Duplicate filtering</i>	<i>Re-execute procedure or retransmit reply</i>	
No	Not applicable	Not applicable	<i>Maybe</i>
Yes	No	Re-execute procedure	<i>At-least-once</i>
Yes	Yes	Retransmit reply	<i>At-most-once</i>

Figure 5.10

Role of client and server stub procedures in RPC

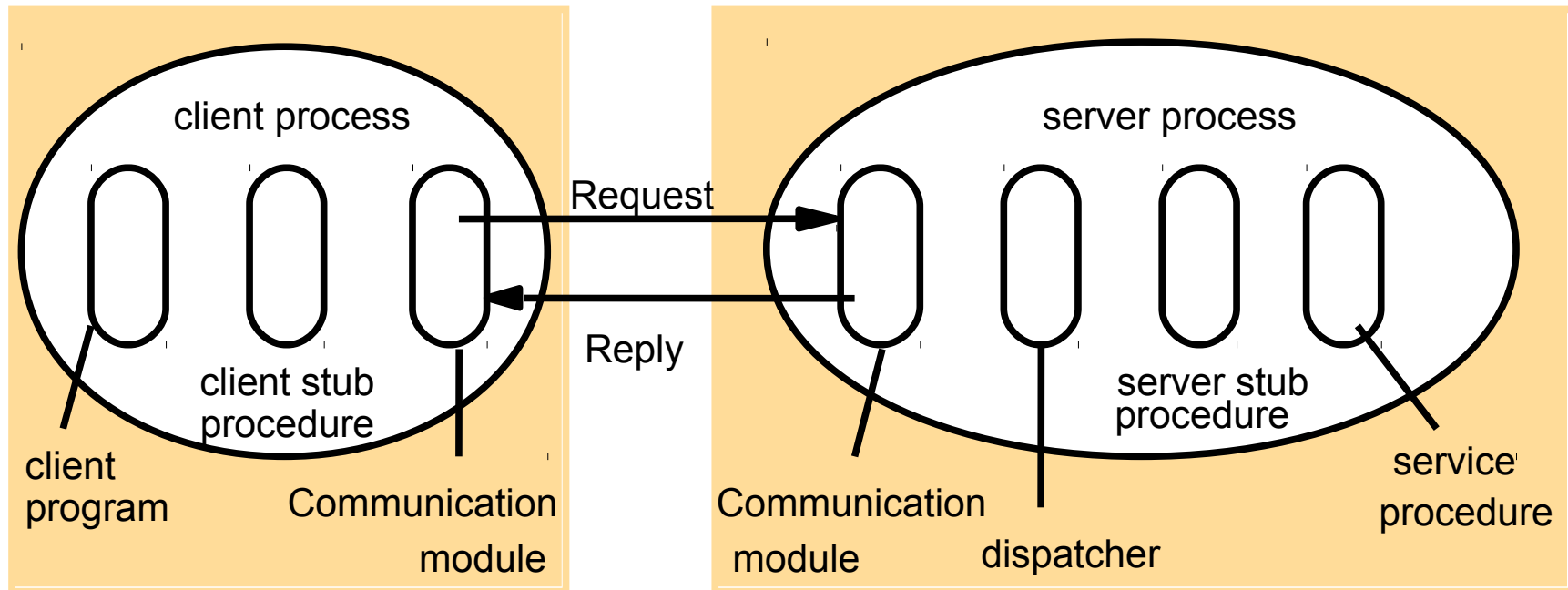


Figure 5.11
Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};
```

```
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
    version VERSION {
        void WRITE(writeargs)=1;
        Data READ(readargs)=2;
        }=2;
    } = 9999;
```

Figure 5.12
Remote and local method invocations

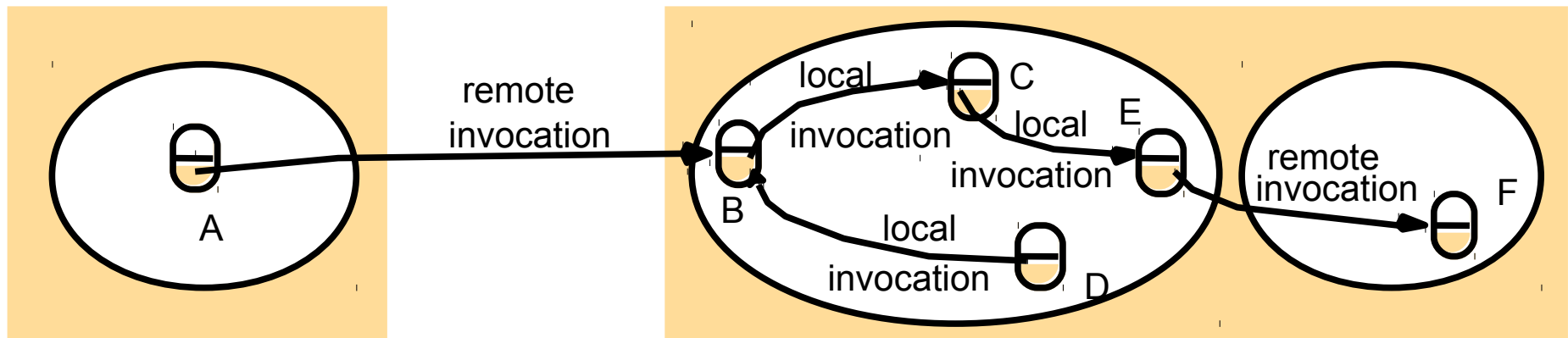


Figure 5.13
A remote object and its remote interface

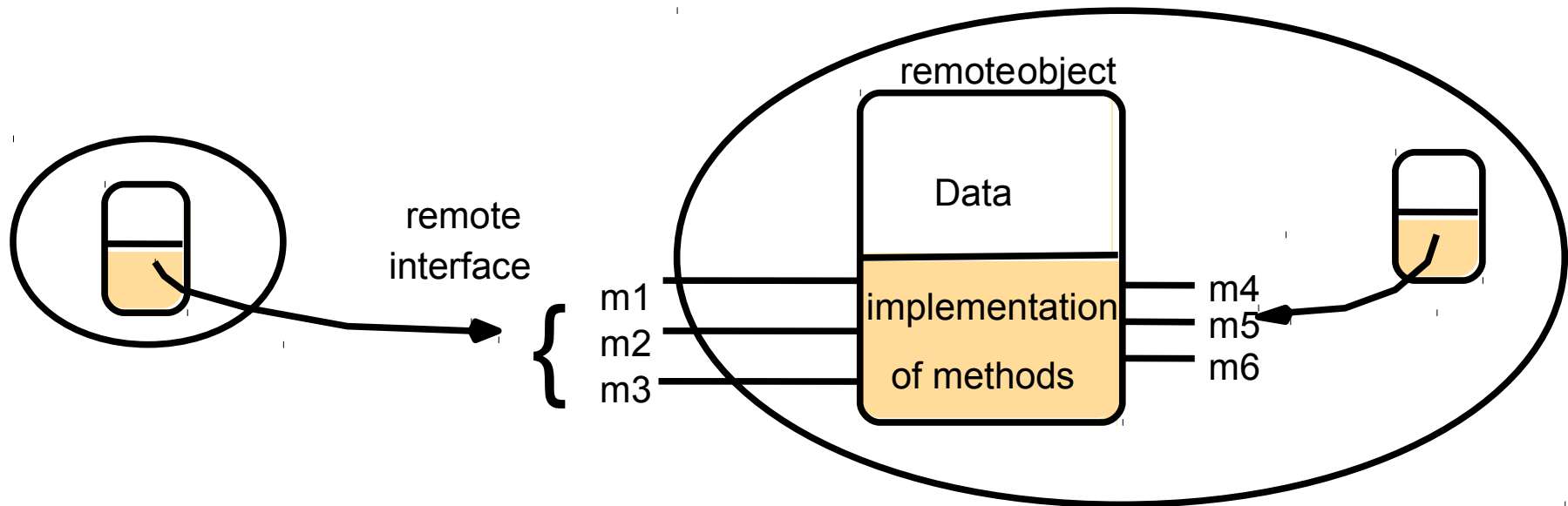


Figure 5.14
Instantiation of remote objects

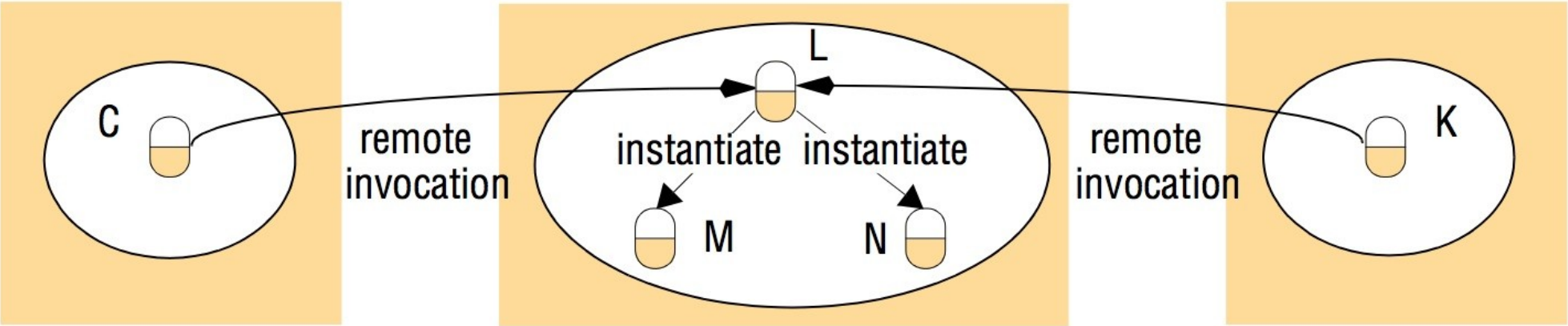


Figure 5.15

The role of proxy and skeleton in remote method invocation

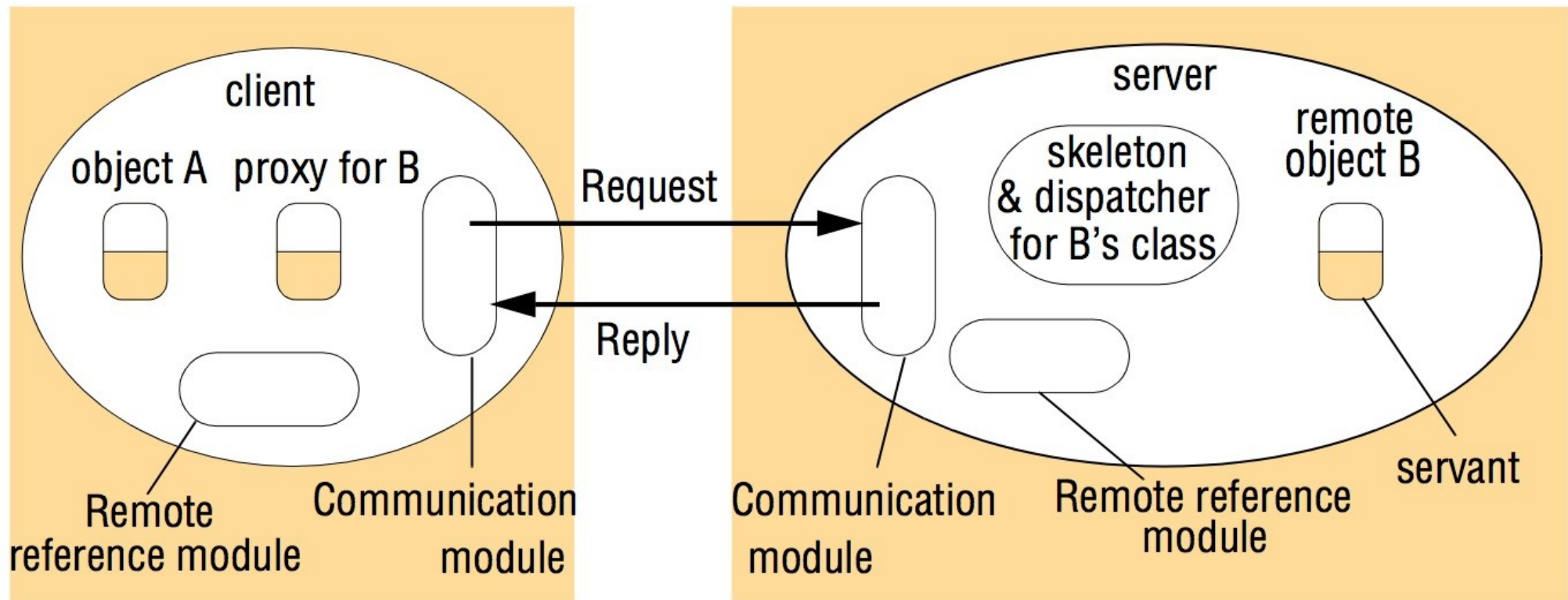


Figure 5.16

Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException; 1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```


Figure 5.17

The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.18, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup (String name)

This method is used by clients to look up a remote object by name, as shown in Figure 5.20 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

Figure 5.18

Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();           1
            Naming.rebind("Shape List", aShapeList );                2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
        }
    }
```

Figure 5.19

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList;                // contains the list of Shapes
    private int version;
    public ShapeListServant() throws RemoteException {...}
    public Shape newShape(GraphicalObject g) throws RemoteException {      1
        version++;
        Shape s = new ShapeServant( g, version);                          2
        theList.addElement(s);
        return s;
    }
    public Vector allShapes() throws RemoteException {...}
    public int getVersion() throws RemoteException { ... }
}
```

Figure 5.20

Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList") ;
            1
            Vector sList = aShapeList.allShapes();
            2
        } catch(RemoteException e) {System.out.println(e.getMessage());}
        } catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```

Figure 5.21
Classes supporting Java RMI

