Modulo I
Introdução aos Sistemas Distribuídos

Prof. Ismael H F Santos

Bibliografia

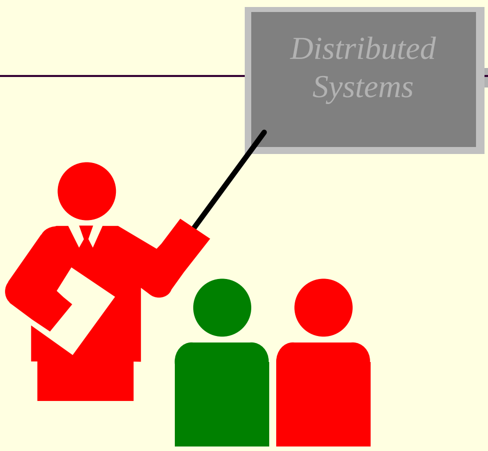
- *Sistemas Distribuídos*
  - *Santos,F., H., Ismael; Notas de Aula, 2005*
- *Sistemas Operacionais e Programação Concorrente*
  - *Toscani e outros, Editora sagra-luzzatto*
- *Fundamentos de Sistemas Operacionais*
  - *Silberschatz, Abraham, Galvin, Peter, Gagne, G., LT*

- *Sistemas Distribuídos*
  - *Andrew S. Tanenbaun; Prentice Hall*
- *Operating System Concepts: Internals and Design Principles*
  - *Williiam Stallings, Prentice Hall*

Ementa

- Distributed Systems
- Hardware for Distibuted Systems
- Client Server Paradigm
- Networking
- Client Server Communication

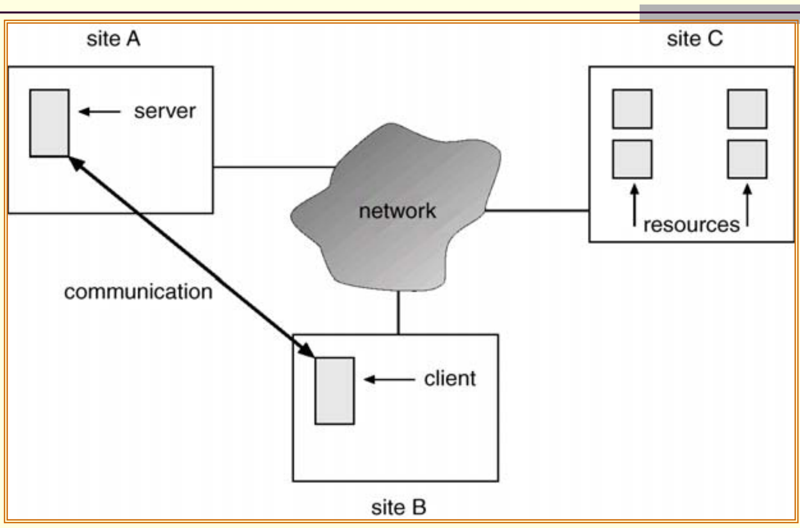
SOA



Motivation

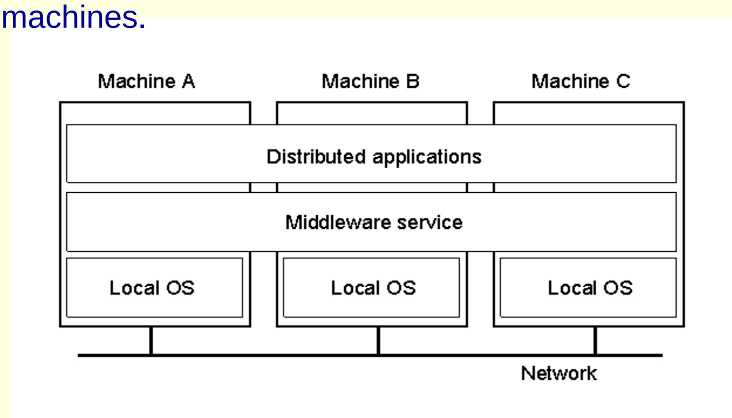
- **Distributed system** is collection of loosely coupled processors interconnected by a communications network
- Processors called *nodes, computers, machines, hosts*
  - *Site* is location of the processor
- Reasons for distributed systems
  - Resource sharing
    - sharing and printing files at remote sites
    - processing information in a distributed database
    - using remote specialized hardware devices
  - Computation speedup – **load sharing**
  - Reliability – detect and recover from site failure, function transfer, reintegrate failed site
  - Communication – message passing

A Distributed System



Definition of a Distributed System (2)

- A distributed system organized as middleware.
- Note that the middleware layer extends over multiple machines.



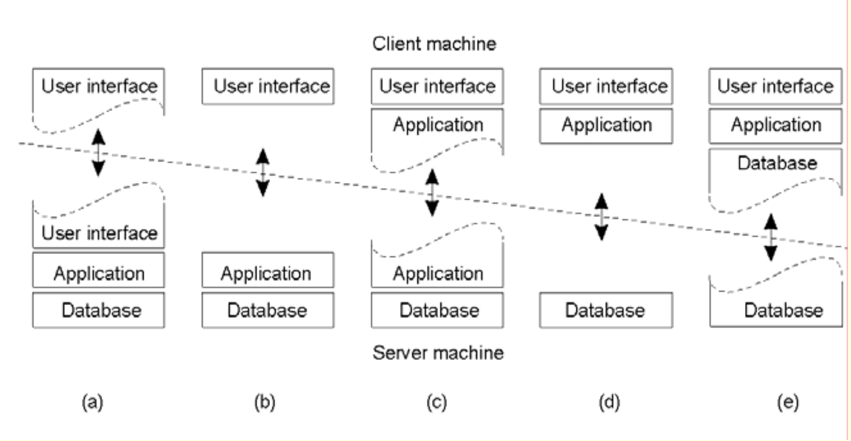
Transparency in a Distributed System

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be shared by several competitive users
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

Different forms of transparency in a distributed system.

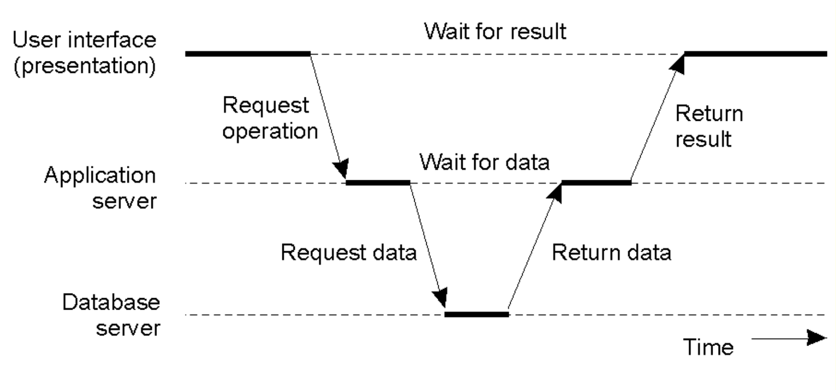
Multitiered Architectures (1)

- Alternative client-server organizations (a) – (e).



Multitiered Architectures (2)

- An example of a server acting as a client.

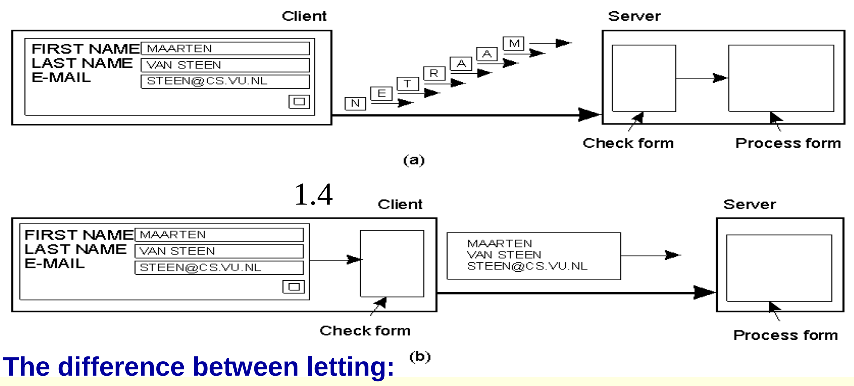


Scalability Problems

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Examples of scalability limitations.

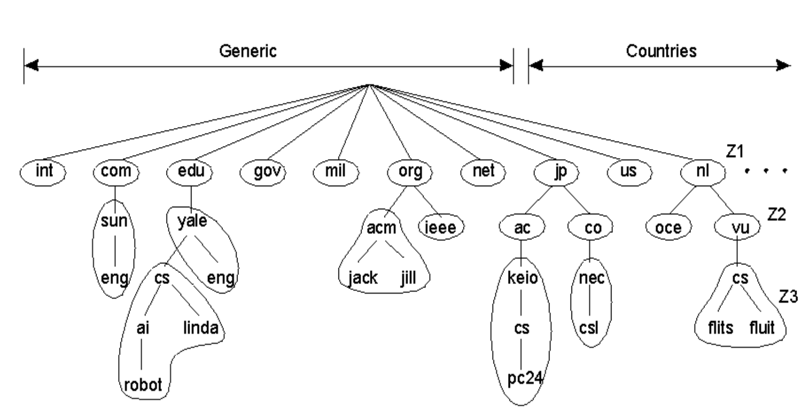
Scaling Techniques (1)



The difference between letting:

- **a) a server or**
- **b) a client check forms as they are being filled**

Scaling Techniques (2)



An example of dividing the DNS name space into zones.

Types of Distributed Operating Systems

- Network Operating Systems
- Distributed Operating Systems

Network-Operating Systems

- Users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:
  - Remote logging into the appropriate remote machine (telnet, ssh)
  - Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism

Distributed-Operating Systems

- Users not aware of multiplicity of machines
  - Access to remote resources similar to access to local resources
- Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task
- Computation Migration – transfer the computation, rather than the data, across the system

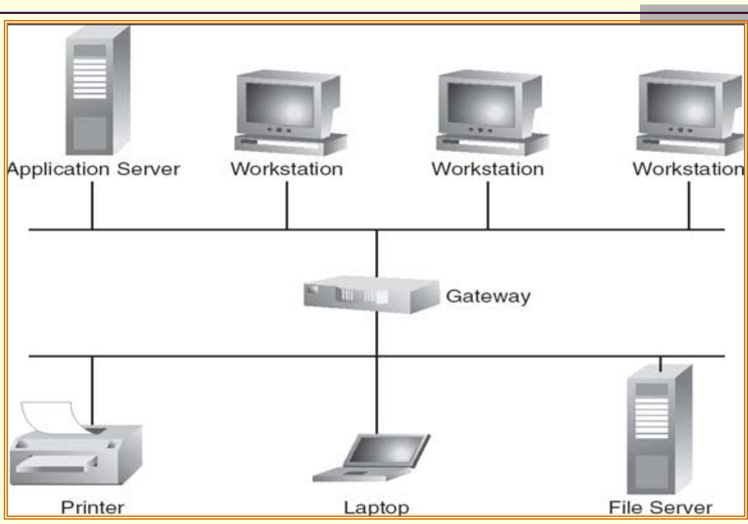
Distributed-Operating Systems (Cont.)

- Process Migration – execute an entire process, or parts of it, at different sites
  - Load balancing – distribute processes across network to even the workload
  - Computation speedup – subprocesses can run concurrently on different sites
  - Hardware preference – process execution may require specialized processor
  - Software preference – required software may be available at only a particular site
  - Data access – run process remotely, rather than transfer all data locally

Network Structure

- Local-Area Network (LAN) – designed to cover small geographical area.
  - Multiaccess bus, ring, or star network
  - Speed ≈ 10 megabits/second, or higher
  - Broadcast is fast and cheap
  - Nodes:
    - usually workstations and/or personal computers
    - a few (usually one or two) mainframes

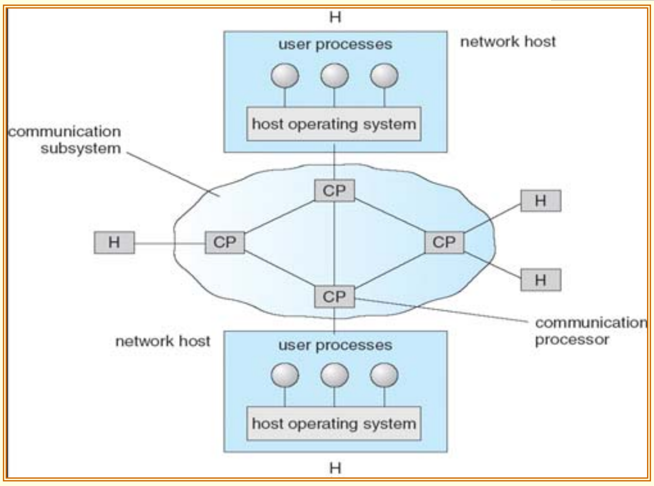
Depiction of typical LAN



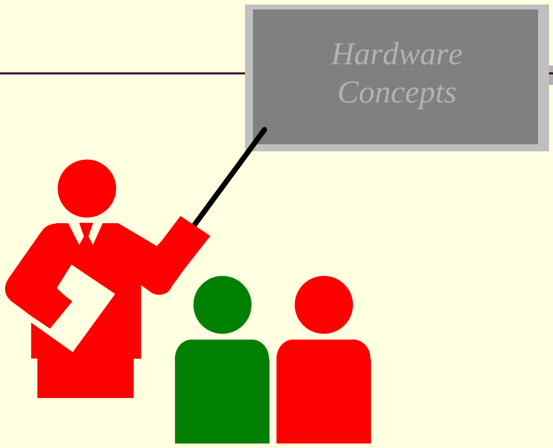
Network Types (Cont.)

- Wide-Area Network (WAN) – links geographically separated sites
  - Point-to-point connections over long-haul lines (often leased from a phone company)
  - Speed ≈ 100 kilobits/second
  - Broadcast usually requires multiple messages
  - Nodes:
    - usually a high percentage of mainframes

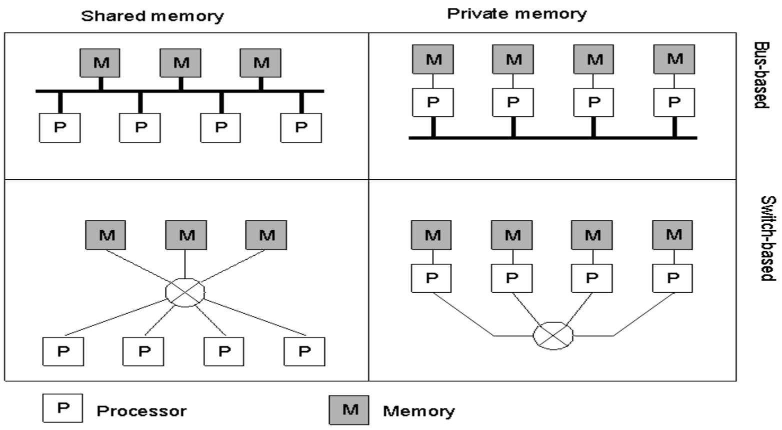
Communication Processors in a Wide-Area Network



SOA

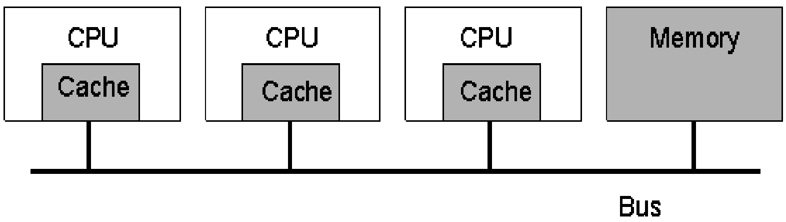


Hardware Concepts



Different basic organizations and memories in distributed computer systems

Multiprocessors (1)

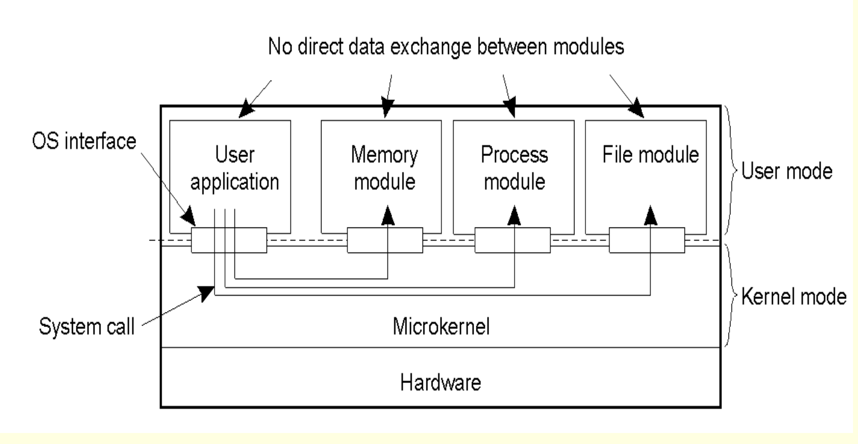


Software Concepts

System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

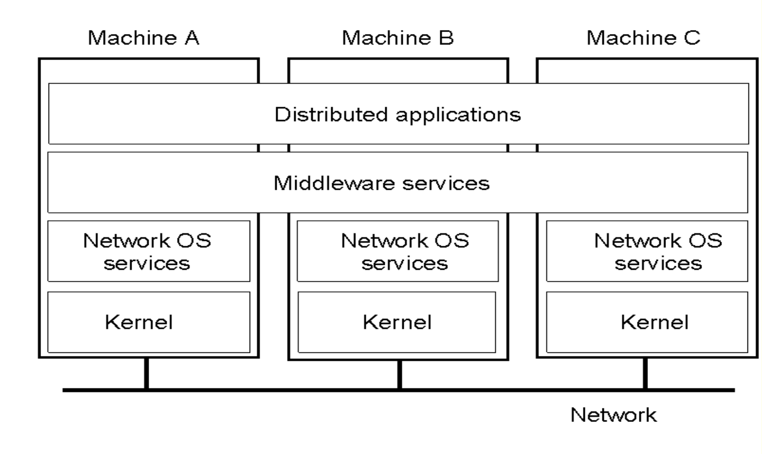
- DOS (Distributed Operating Systems)
- NOS (Network Operating Systems)
- Middleware

Uniprocessor Operating Systems



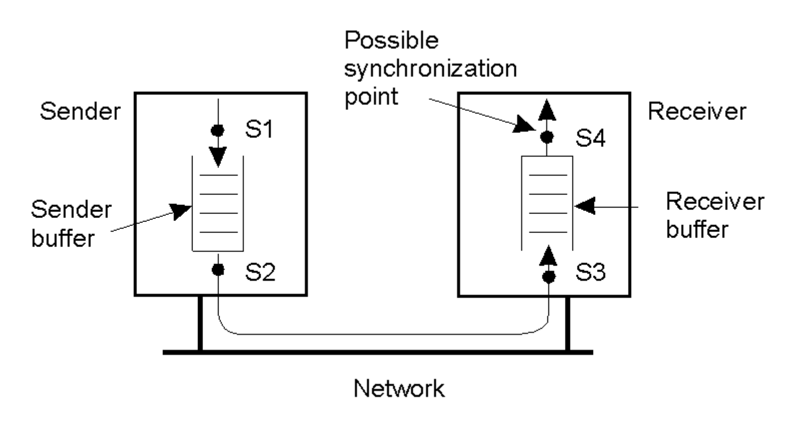
Multicomputer Operating Systems (1)

- General structure of a distributed system as middleware.



Multicomputer Operating Systems (2)

- Alternatives for blocking and buffering in message passing.



Multicomputer Operating Systems (3)

- Relation between blocking, buffering, and reliable communications.

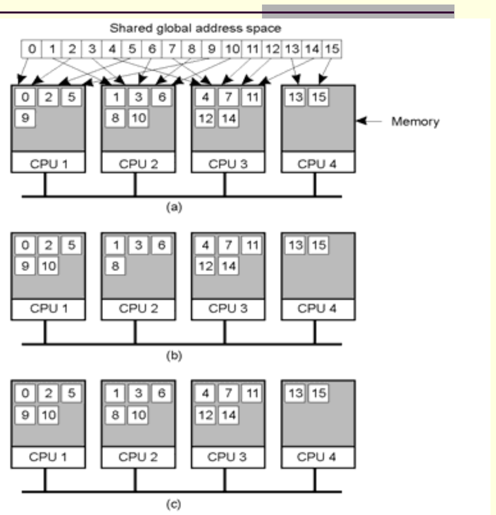
Synchronization point	**Send buffer**	**Reliable comm. guaranteed?**
Block sender until buffer not full	Yes	Not necessary
Block sender until message sent	No	Not necessary
Block sender until message received	No	Necessary
Block sender until message delivered	No	Necessary

Distributed Shared Memory Systems (1)

a) Pages of address space distributed among four machines

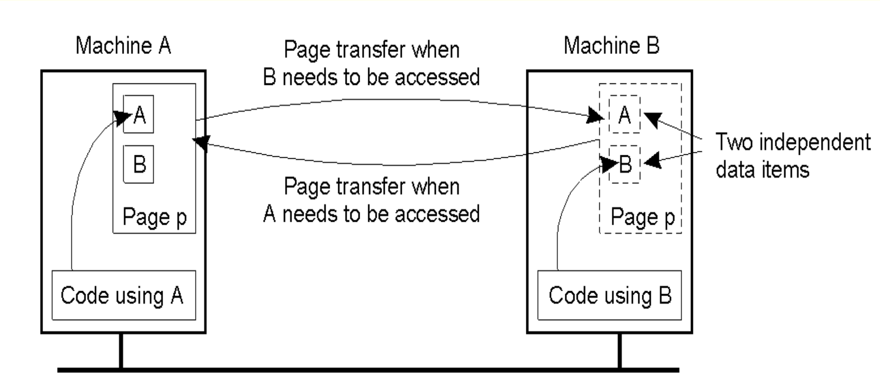
b) Situation after CPU 1 references page 10

c) Situation if page 10 is read only and replication is used



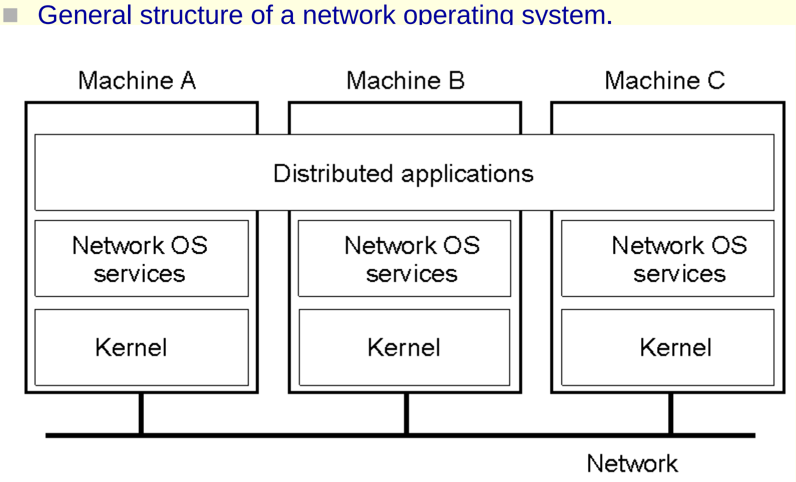
Distributed Shared Memory Systems (2)

- **False sharing of a page between two independent**



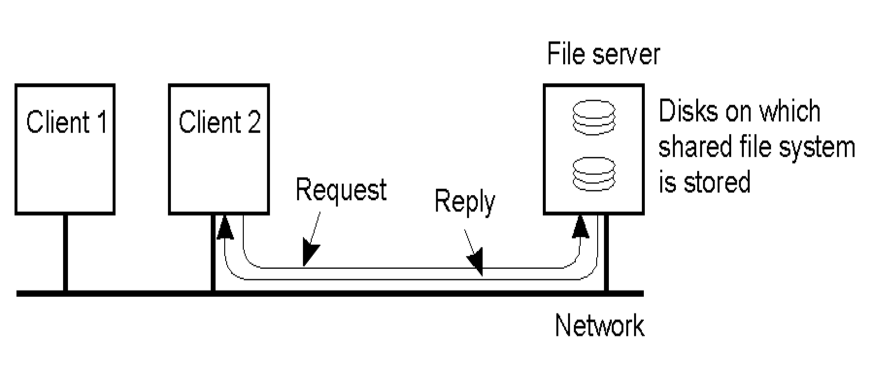
Network Operating System (1)

- General structure of a network operating system.



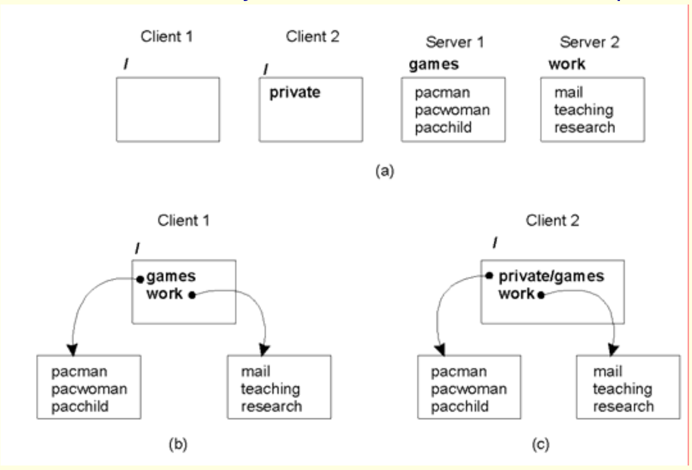
Network Operating System (2)

- Two clients and a server in a network operating system.

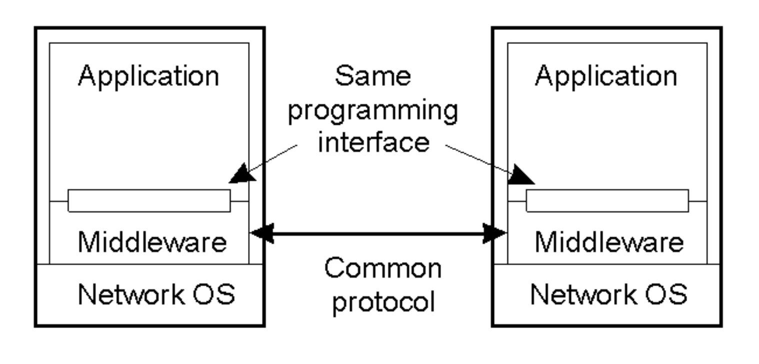


Network Operating System (3)

- Different clients may mount the servers in different places.



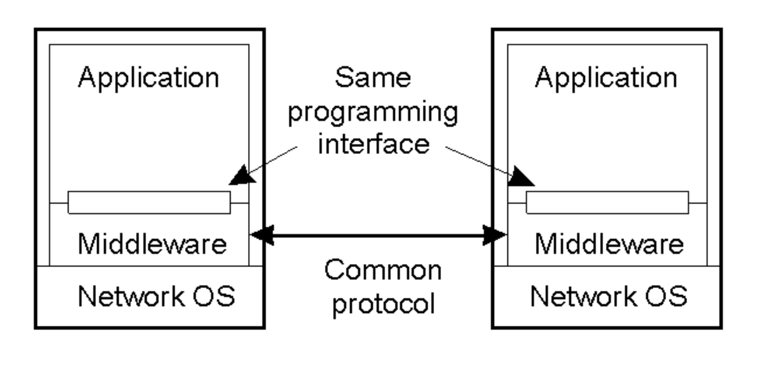
Middleware and Openness



- In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.

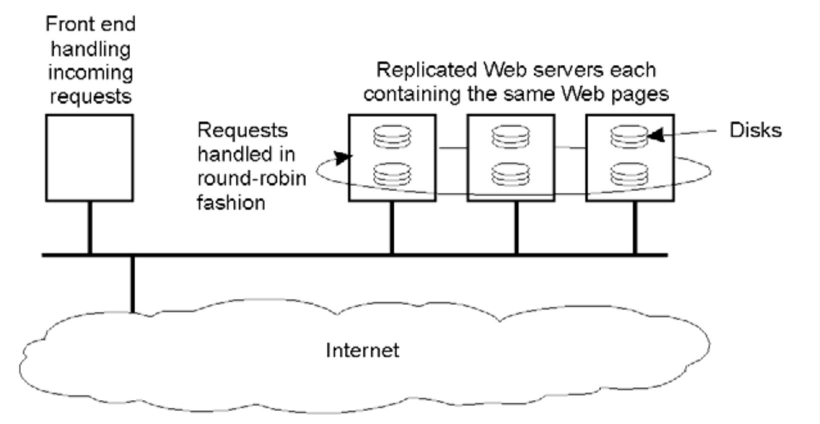
Middleware and Openness

- In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.



Modern Architectures

- An example of horizontal distribution of a Web service.

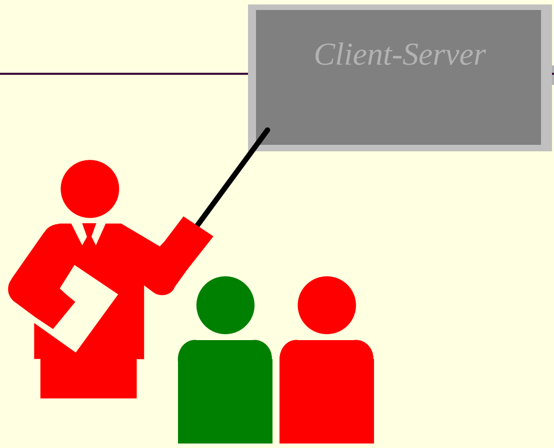


Comparison between Systems

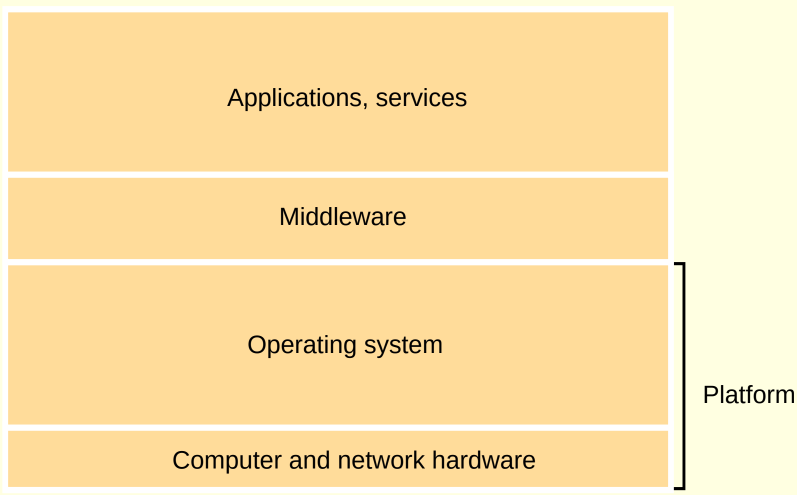
- A comparison between multiprocessor operating systems, multicomputer operating systems, network operating systems, and middleware based distributed systems.

Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

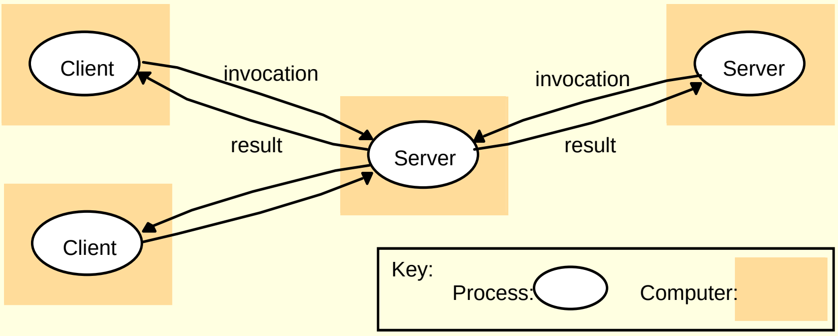
SOA



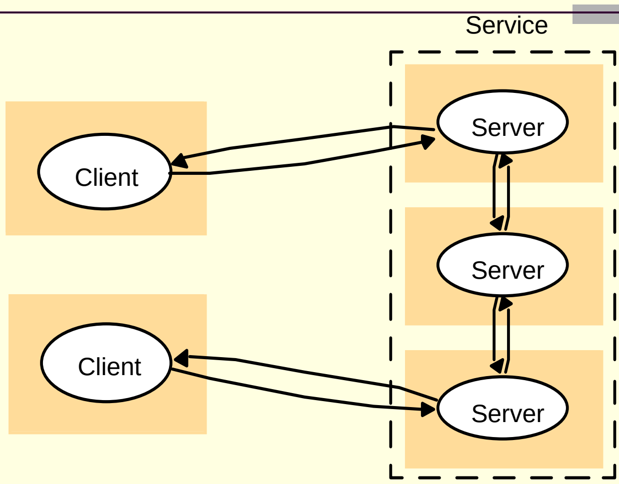
Software and hardware service layers in distributed systems



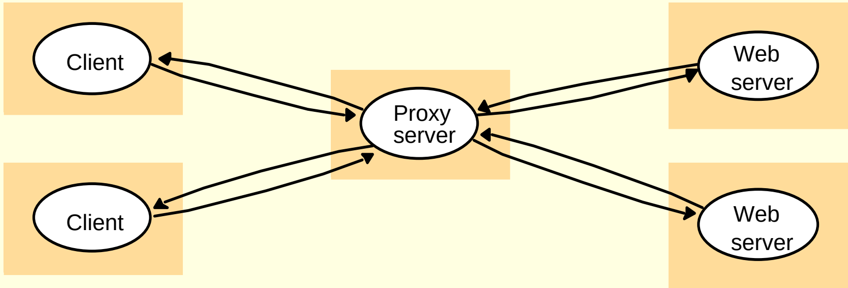
Clients invoke individual servers



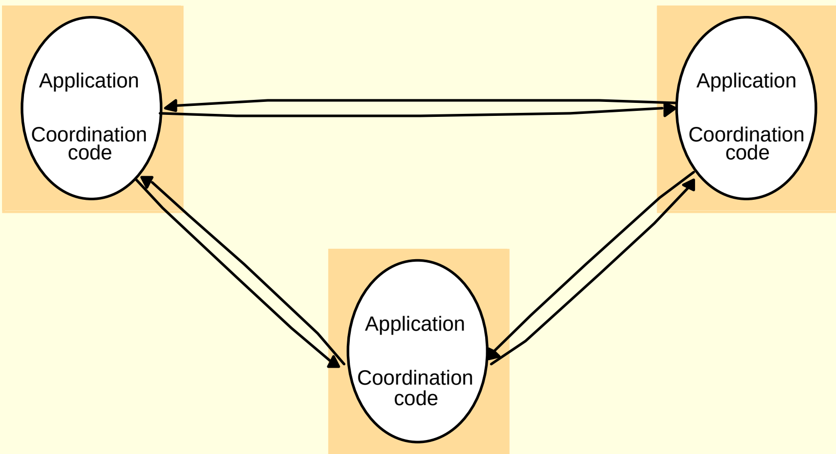
A service provided by multiple servers



Web proxy server

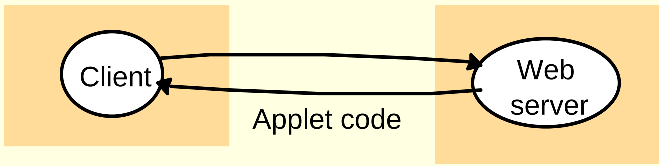


A distributed application based on peer processes



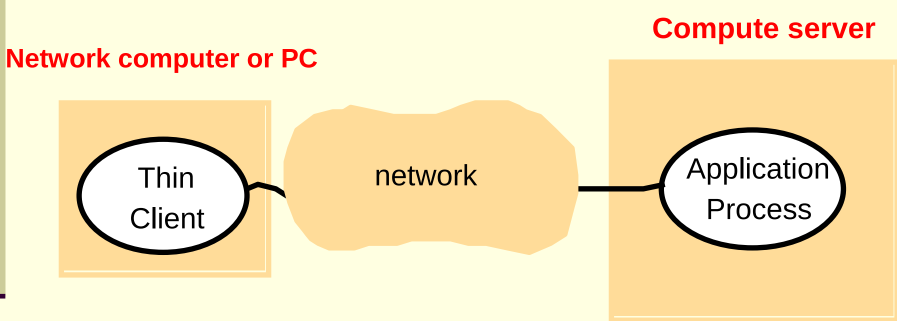
Web applets

a) client request results in the downloading of applet code

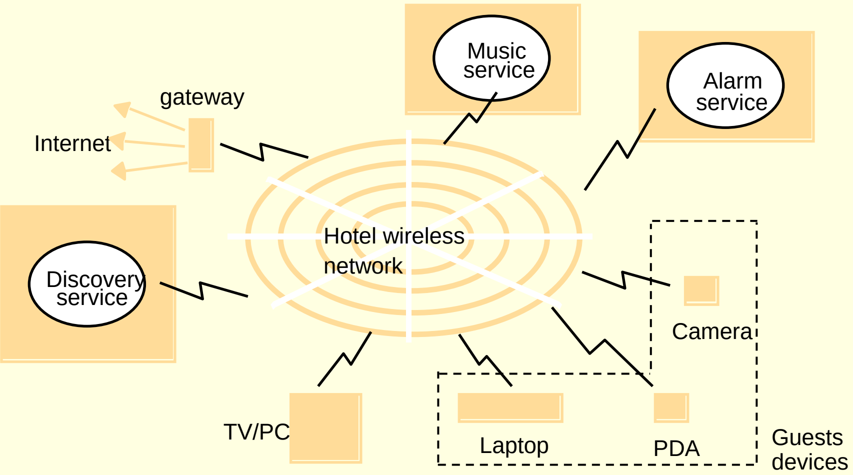


b) client interacts with the applet

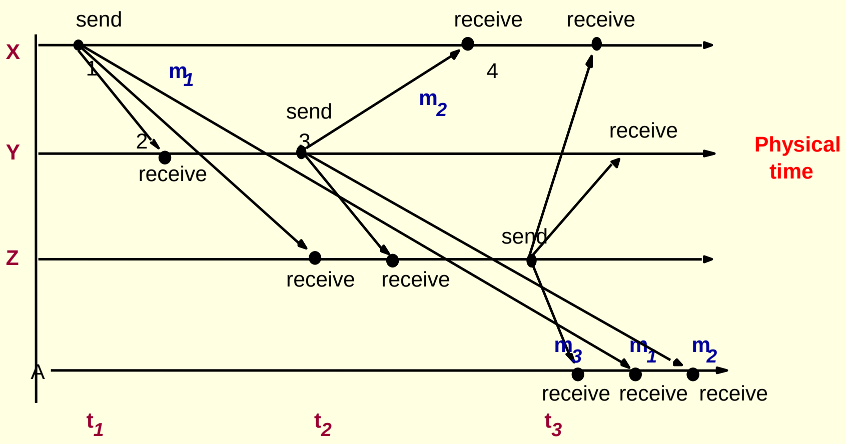
Thin clients and compute servers



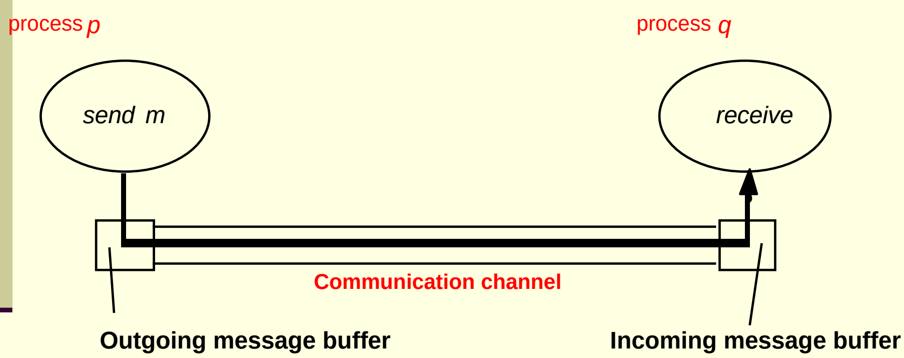
Spontaneous networking in a hotel



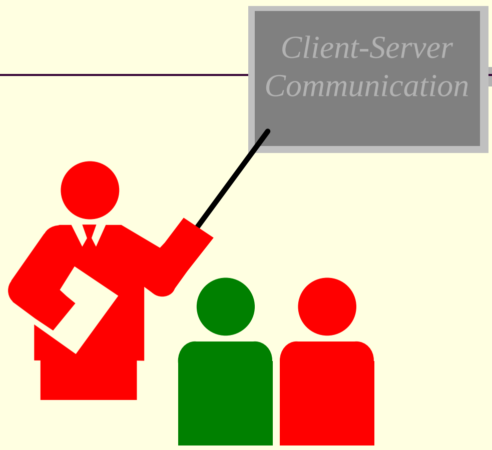
Real-time ordering of events



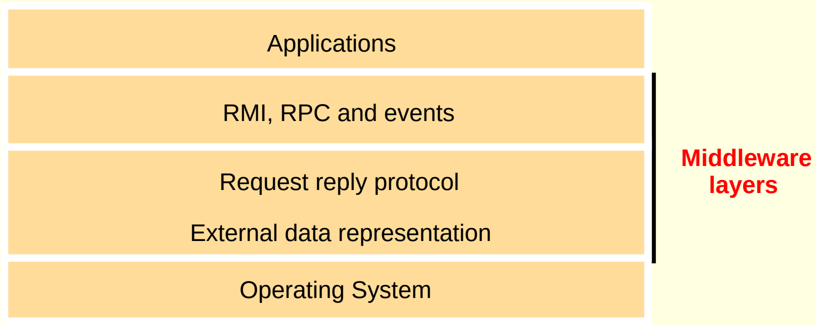
Processes and channels



SOA



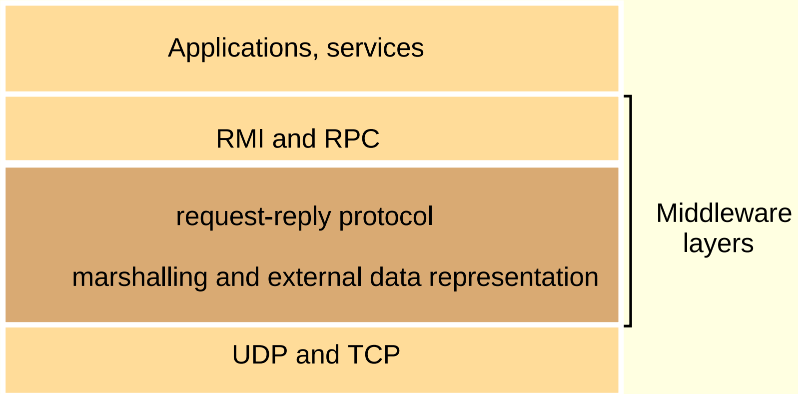
Middleware layers



Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)
- CORBA
- Object Registration

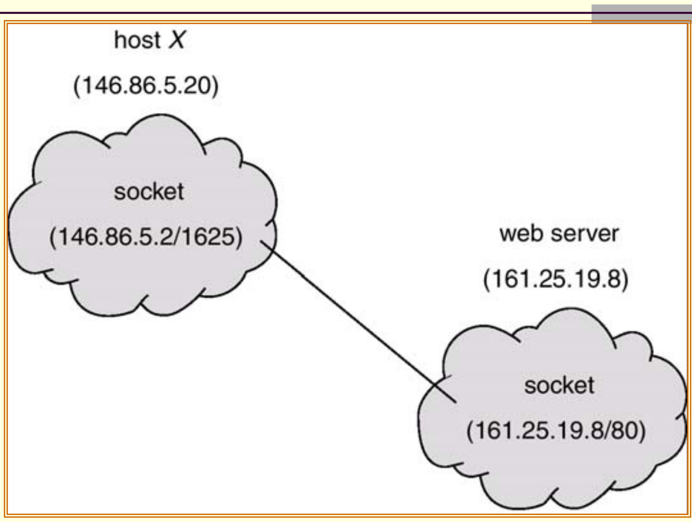
Middleware layers



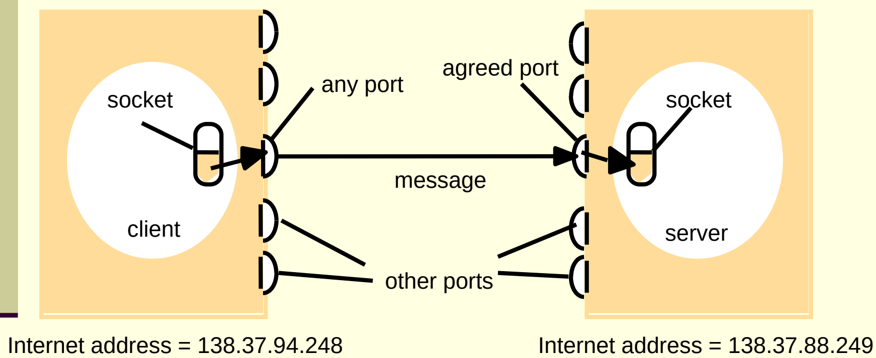
Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All Ports < 1024 are Considered “well-known”
  - TELNET uses port 23
  - FTP uses port 21
  - HTTP server uses port 80

Socket Communication



Sockets and ports



Java Sockets

- Java Provides:
  - Connection-Oriented (TCP) Sockets
  - Connection-less (UDP) Sockets
  - Multicast Connection-less Socket

Time-Of-Day Server/Client

- Server uses ServerSocket to Create the Socket on Port 5155

```
ServerSocket s = new ServerSocket(5155);
```

- To Accept Connections From Clients:

```
Socket client = s.accept();
```

- Connections are Often Serviced in Separate Threads
- The Client Connects to the Server Using Socket class with the IP Address of the Server.

```
Socket s = new Socket(“127.0.0.1”,5155);
```

Sockets used for streams

Requesting a connection

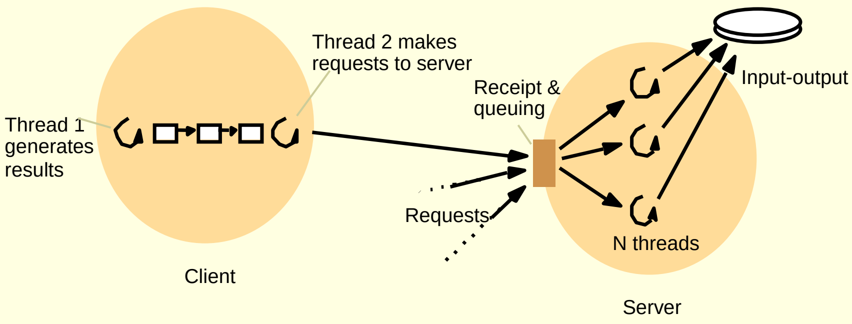
```
s = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
bind(s, ServerAddress);
listen(s,5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

ServerAddress and *ClientAddress* are socket addresses

Client and server with threads



TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
  public static void main (String args[]) {
  // arguments supply message and hostname of destination
    Socket s = null;
     try{     int serverPort = 7896;
              s = new Socket(args[1], serverPort);
              DataInputStream in = new DataInputStream( s.getInputStream());
              DataOutputStream out = new DataOutputStream( s.getOutputStream());
              out.writeUTF(args[0]);            // UTF is a string encoding
              String data = in.readUTF();        System.out.println("Received: "+ data) ;
    } catch (UnknownHostException e){ System.out.println("Sock:"+e.getMessage());
    } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
    } catch (IOException e){System.out.println("IO:"+e.getMessage());}
    } finally {     if(s!=null)
                        try {s.close();
                        }catch (IOException e){System.out.println("close:"+e.getMessage());}}
              }
}
```


TCP server continued

```
class Connection extends Thread {
   DataInputStream in;
   DataOutputStream out;
   Socket clientSocket;
    public Connection (Socket aClientSocket) {
       try {
              clientSocket = aClientSocket;
              in = new DataInputStream( clientSocket.getInputStream());
              out =new DataOutputStream( clientSocket.getOutputStream());
              this.start();
        } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
       try {                                          // an echo server
              String data = in.readUTF();
              out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
   }
}
```

An Example Client and Server (1)

- The *header.h* file used by the client and server.

```
/* Definitions needed by clients and servers.                    */
#define TRUE                1
#define MAX_PATH            255    /* maximum length of file name          */
#define BUF_SIZE            1024   /* how much data to transfer at once    */
#define FILE_SERVER         243    /* file server's network address        */

/* Definitions of the allowed operations */
#define CREATE              1      /* create a new file                    */
#define READ                2      /* read data from a file and return it  */
#define WRITE               3      /* write data to a file                 */
#define DELETE              4      /* delete an existing file              */

/* Error codes. */
#define OK                  0      /* operation performed correctly        */
#define E_BAD_OPCODE        -1     /* unknown operation requested          */
#define E_BAD_PARAM         -2     /* error in a parameter                 */
#define E_IO                -3     /* disk error or other I/O error        */

/* Definition of the message format. */
struct message {
    long source;                   /* sender's identity                    */
    long dest;                     /* receiver's identity                  */
    long opcode;                   /* requested operation                  */
    long count;                    /* number of bytes to transfer          */
    long offset;                   /* position in file to start I/O        */
    long result;                   /* result of the operation              */
    char name[MAX_PATH];           /* name of file being operated on       */
    char data[BUF_SIZE];           /* data to be read or written           */
};
```


An Example Client and Server (2)

- A sample server.

```
#include <header.h>
void main(void) {
    struct message ml, m2;                 /* incoming and outgoing messages */
    int r;                                 /* result code                    */

    while(TRUE) {                          /* server runs forever            */
        receive(FILE_SERVER, &ml);         /* block waiting for a message    */
        switch(ml.opcode) {                /* dispatch on type of request    */
            case CREATE:    r = do_create(&ml, &m2); break;
            case READ:      r = do_read(&ml, &m2); break;
            case WRITE:     r = do_write(&ml, &m2); break;
            case DELETE:    r = do_delete(&ml, &m2); break;
            default:        r = E_BAD_OPCODE;
        }
        m2.result = r;                     /* return result to client        */
        send(ml.source, &m2);              /* send reply                     */
    }
}
```

An Example Client and Server (3)

- A client using the server to copy a file.

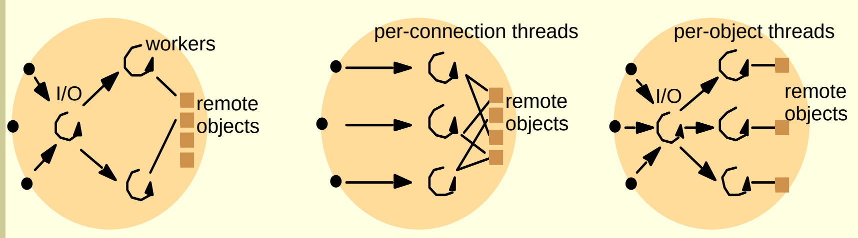
(a)

```
#include <header.h>
int copy(char *src, char *dst){              /* procedure to copy file using the server   */
    struct message ml;                       /* message buffer                            */
    long position;                           /* current file position                     */
    long client = 110;                       /* client's address                          */

    initialize( );                           /* prepare for execution                     */
    position = 0;
    do {
        ml.opcode = READ;                    /* operation is a read                       */
        ml.offset = position;                /* current position in the file              */
        ml.count  = BUF_SIZE;                /* how many bytes to read                    */
        strcpy(&ml.name, src);               /* copy name of file to be read to message   */
        send(FILESERVER, &ml);               /* send the message to the file server       */
        receive(client, &ml);                /* block waiting for the reply               */

        /* Write the data just received to the destination file.                          */
        ml.opcode = WRITE;                   /* operation is a write                      */
        ml.offset = position;                /* current position in the file              */
        ml.count  = ml.result;               /* how many bytes to write                   */
        strcpy(&ml.name, dst);               /* copy name of file to be written to buf    */
        send(FILE_SERVER, &ml);              /* send the message to the file server       */
        receive(client, &ml);                /* block waiting for the reply               */
        position += ml.result;               /* ml.result is number of bytes written      */
    } while( ml.result > 0 );                /* iterate until done                        */
    return(ml.result >= 0 ? OK : ml result); /* return OK or error code                   */
}
```


Alternative server threading architectures



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier

Pages of address space resident in memory; hardware cache entries

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Set and return the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
Enter the *READY* state and invoke the scheduler.

Java thread synchronization calls

destroy

Destroy the thread.

thread.join(int millisecs)

Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

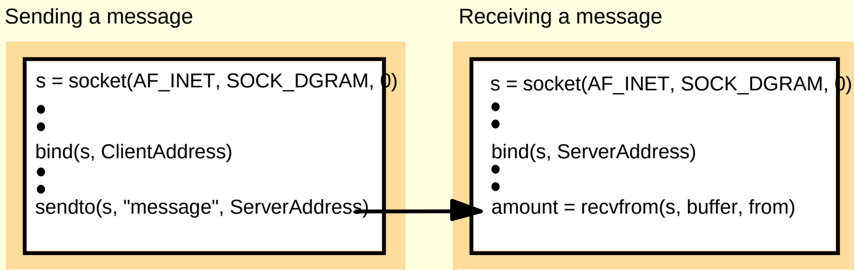
object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), object.notifyAll()

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Sockets used for datagrams



ServerAddress and *ClientAddress* are socket addresses

UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
     // args give message contents and server hostname
     DatagramSocket aSocket = null;
     try {  aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
           DatagramPacket request = new DatagramPacket(m,  args[0].length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
      }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
      }catch (IOException e){System.out.println("IO: " + e.getMessage());}
      }finally {if(aSocket != null) aSocket.close();}    }
}
```


UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
  public static void main(String args[]){
    DatagramSocket aSocket = null;
    try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
               DatagramPacket request = new DatagramPacket(buffer, buffer.length);
               aSocket.receive(request);
               DatagramPacket reply = new DatagramPacket(request.getData(),
                request.getLength(), request.getAddress(), request.getPort());
                 aSocket.send(reply);
            }
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
  }
}
```

TCP server makes a connection for each client and then echoes the client's request

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
     try{
       int serverPort = 7896;
       ServerSocket listenSocket = new ServerSocket(serverPort);
        while(true) {
             Socket clientSocket = listenSocket.accept();
             Connection c = new Connection(clientSocket);
         }
     } catch(IOException e) {
       System.out.println("Listen :"+e.getMessage());
     }
   }
}

// this figure continues on the next slide
```


Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
   public static void main(String args[]){
    // args give message contents & destination multicast group (e.g. "228.5.6.7")
   MulticastSocket s =null;
   try {
     InetAddress group = InetAddress.getByName(args[1]);
     s = new MulticastSocket(6789);
     s.joinGroup(group);
     byte [] m = args[0].getBytes();
     DatagramPacket messageOut = new DatagramPacket(m, m.length,
                                                    group, 6789);
     s.send(messageOut);

  // this figure continued on the next slide
```

Multicast peer continued …

```
// get messages from others in group
byte[] buffer = new byte[1000];
 for(int i=0; i< 3; i++) {
    DatagramPacket messageIn =  new DatagramPacket(buffer, buffer.length);
    s.receive(messageIn);
    System.out.println("Received:" + new String(messageIn.getData()));
 }
 s.leaveGroup(group);
}catch (SocketException e) {
  System.out.println("Socket: " + e.getMessage());
}catch (IOException e){
  System.out.println("IO: " + e.getMessage());
} finally {
  if(s != null) s.close();}
 }
}
```


Indication of Java serialized form

Serialized values				Explanation
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

Representation of a remote object reference

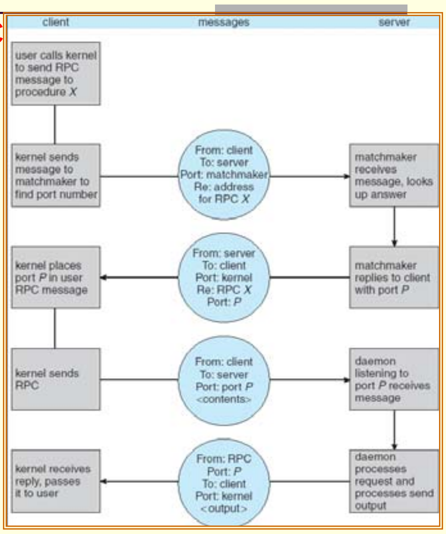
32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface of remote object

Remote Procedure Calls

- Sockets are Considered Low-level.
- RPCs Offer a higher-level Form of Communication
- Client Makes Procedure Call to “Remote” Server Using Ordinary Procedure Call Mechanisms.
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.

Remote Procedure Calls

- Remote procedure call, RPC
  - **Stubs** – client-side proxy for the actual procedure on the server.
  - The client-side stub locates the server and *marshalls* the parameters.
  - The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

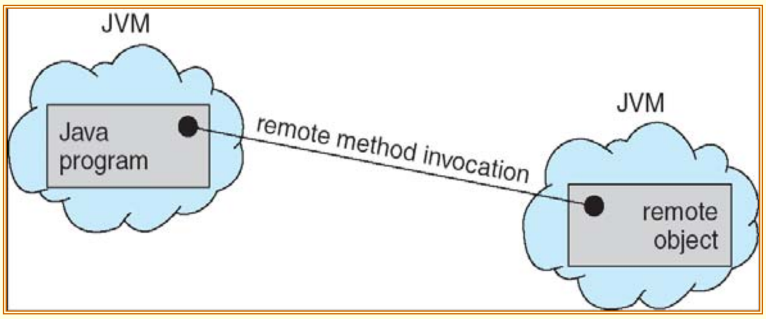


Stubs and Skeletons

- "Stub" is a Proxy for the Remote Object – Resides on Client.
- The Stub "Marshalls" the Parameters and Sends Them to the Server.
- "Skeleton" is on Server Side.
- Skeleton "Unmarshalls" the Parameters and Delivers Them to the Server.

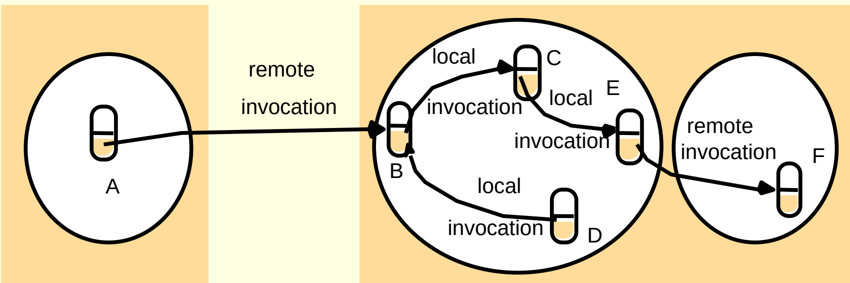
Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.

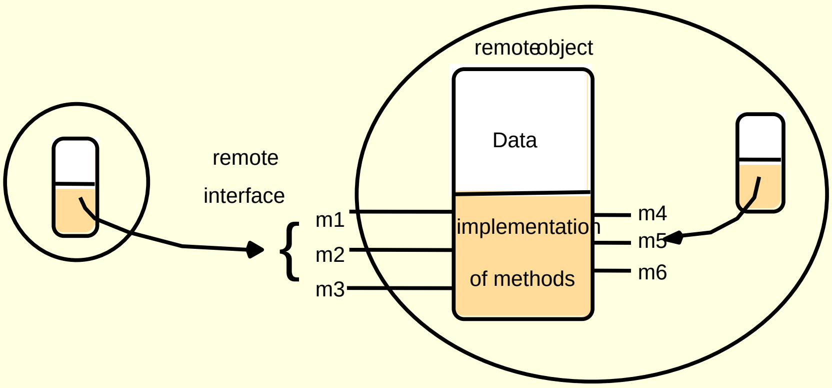


Remote Method Invocation

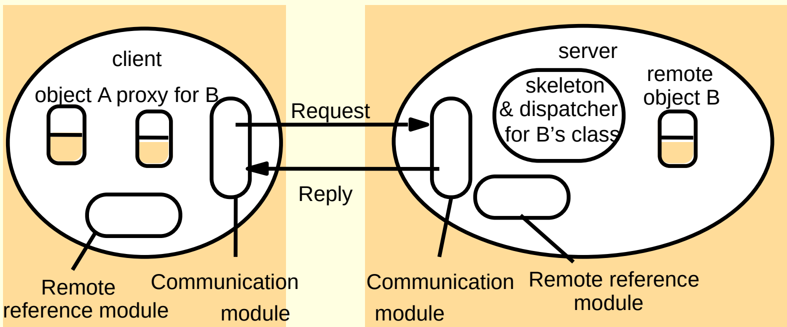
- A Thread May Invoke a Method on a Remote Object
- An Object is Considered “remote” if it Resides in a Separate Java Virtual Machine.



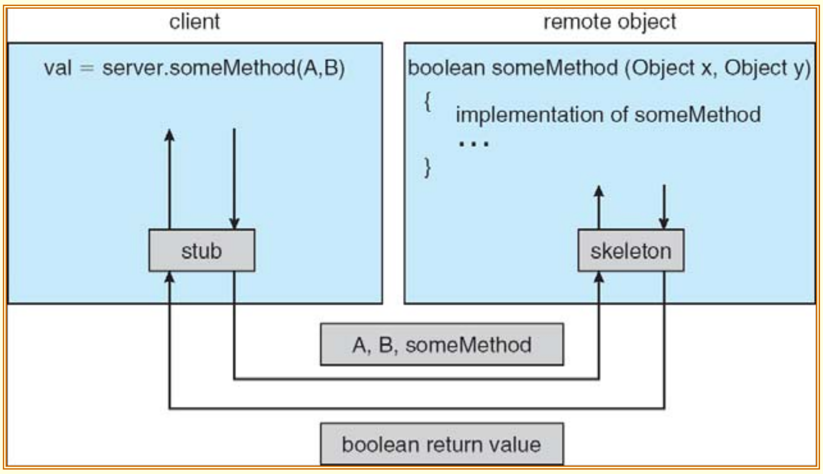
A remote object and its remote interface



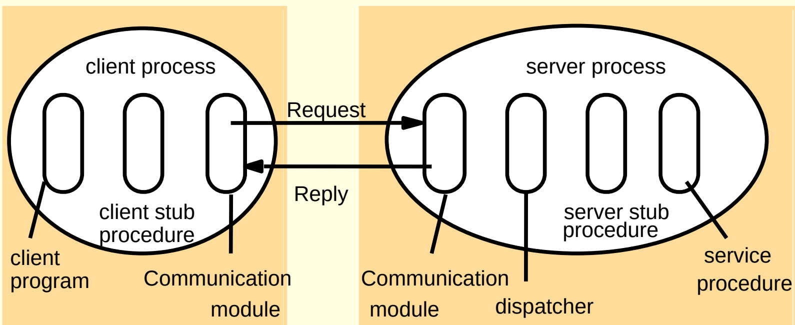
The role of proxy and skeleton in remote method invocation



Marshalling Parameters



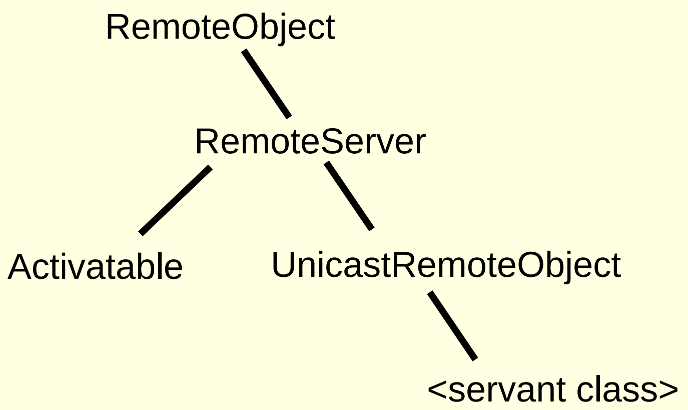
Role of client and server stub procedures in RPC



RPC versus RMI

- RPC's Support Procedural Programming Style
- RMI Supports Object-Oriented Programming Style
- Parameters to RPCs are Ordinary Data Structures
- Parameters to RMI are Objects

Classes supporting Java RMI



Parameters

- Local (Non-Remote) Objects are Passed by Copy using Object Serialization
- Remote Objects are Passed by Reference

Remote Objects

- Remote Objects are Declared by Specifying an interface that extends `java.rmi.Remote`
- Every Method Must Throw `java.rmi.RemoteException`

MessageQueue interface

```
import java.rmi.*;

public interface MessageQueue extends Remote
{
  public void send(Object item) throws
                              RemoteException;
  public Object receive() throws RemoteException;
}
```


MessageQueue implementation

```
import java.rmi.*;
public class MessageQueueIMPL
  extends server.UnicastRemoteObject
  implements MessageQueue
{
  public void send(Object item) throws
                              RemoteException
  { /* implementation */
  }
  public Object receive() throws RemoteException
  { /* implementation */
  }
}
```

The Client

- The Client Must

  (1) Install a Security Manager:

```
System.setSecurityManager(
      new RMISecurityManager());
```

  (2) Get a Reference to the Remote Object

```
MessageQueue mb;
mb = (MessageQueue)Naming.lookup(
        “rmi://127.0.0.1/MessageServer”’);
```


Running the Producer-Consumer Using RMI

- Compile All Source Files and Generate Stubs

```
javac *.java; rmic MessageQueueImpl
```

- Start the Registry Service

```
rmiregistry
```

- Create the Remote Object

```
java –Djava.security.policy=java.policy
        MessageQueueImpl
```

- Start the Client

```
java –Djava.security.policy=java.policy
        Factory
```

Policy File

- New with Java 2

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


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;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws
RemoteException;    2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

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 15.13, 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 15.15 line 1. A remote object reference is returned.

String [] list()

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

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();
                Naming.rebind("Shape List", aShapeList );
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

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          1
    private int version;
   public ShapeListServant()throws RemoteException{...}
   public Shape newShape(GraphicalObject g) throws RemoteException {        2
       version++;
       Shape s = new ShapeServant( g, version);                             3
       theList.addElement(s);
       return s;
   }
   public  Vector allShapes()throws RemoteException{...}
   public int getVersion() throws RemoteException { ... }
}
```

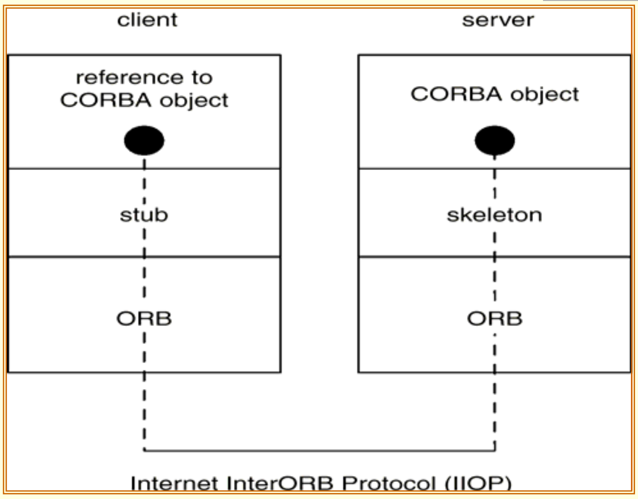

Java client of *ShapeList*

```
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());}
  }
}
```

CORBA

- **RMI** is Java-to-Java Technology
- **CORBA** is Middleware that Allows Heterogeneous Client and Server Applications to Communicate
- **Interface Definition Language (IDL)** is a Generic Way to Describe an Interface to a Service a Remote Object Provides
- **Object Request Broker (ORB)** Allows Client and Server to Communicate through IDL.
- **Internet InterORB Protocol (IIOP)** is a Protocol Specifying how the ORBs can Communicate.

Corba Model



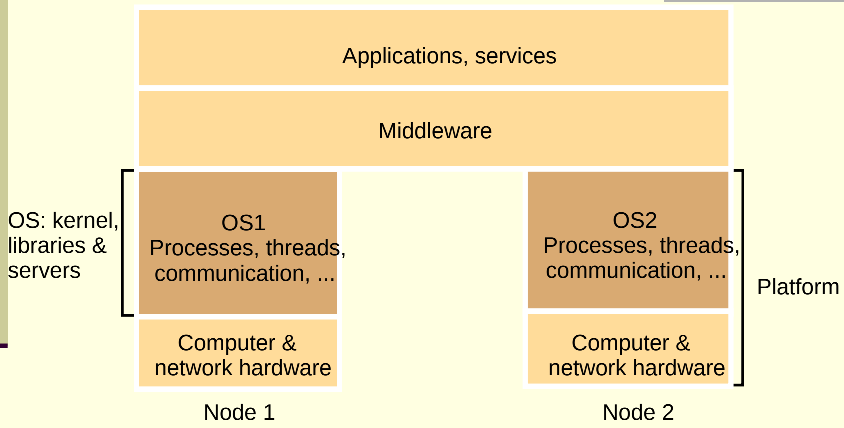
CORBA IDL example

```
// 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();
};
```

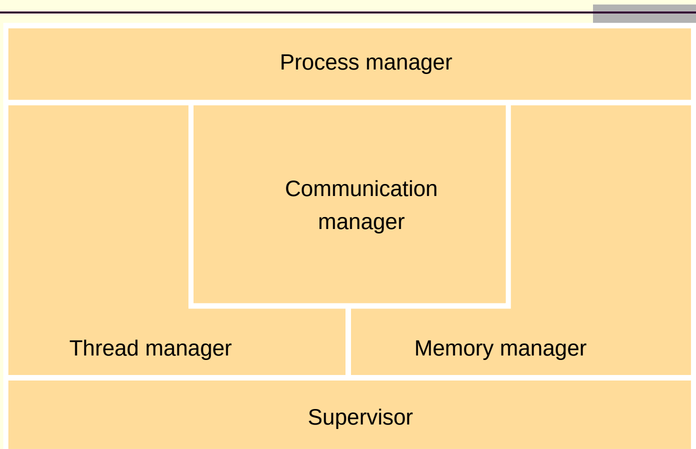

Registration Services

- Registration Service Allows Remote Objects to “register” their services.
- RMI, CORBA Require Registration Services

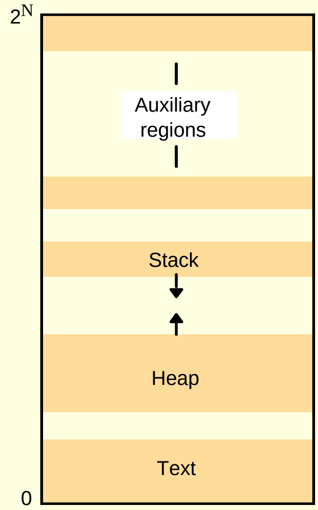
System layers



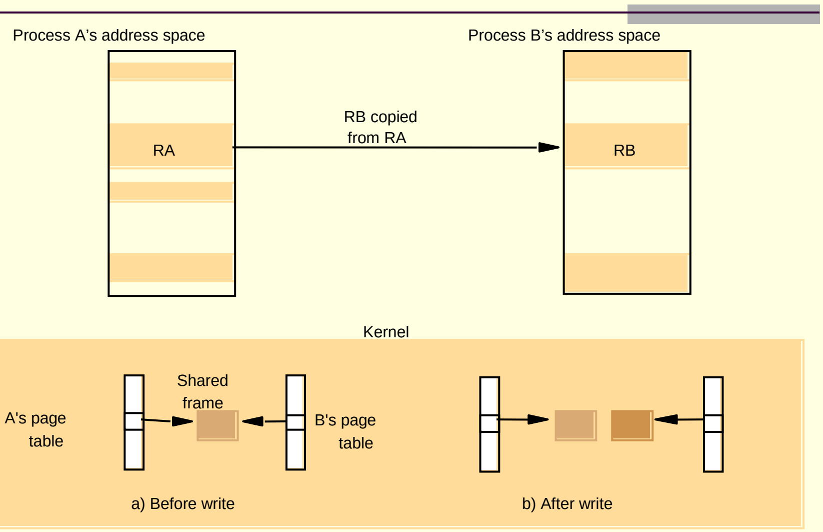
Core OS functionality



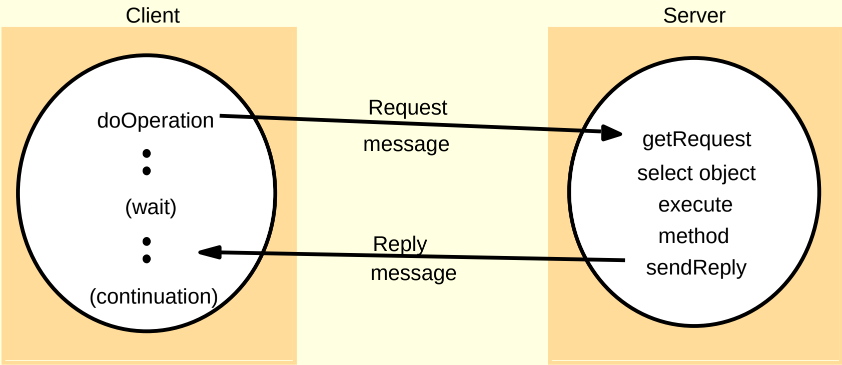
Address space



Copy-on-write



Request-reply communication



Operations of the request-reply protocol

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

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.

Request-reply message structure

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

RPC exchange protocols

Name	***Messages sent by***		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*