Modulo II – Comunicação Sistemas Distribuídos

Prof. Ismael H F Santos

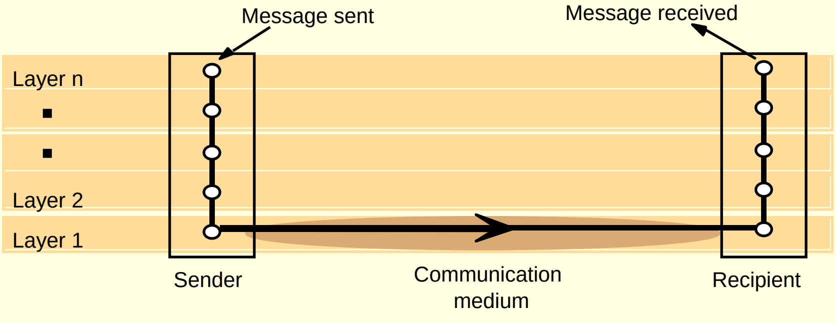
Ementa

- Sistemas Distribuídos
  - *Cliente-Servidor*

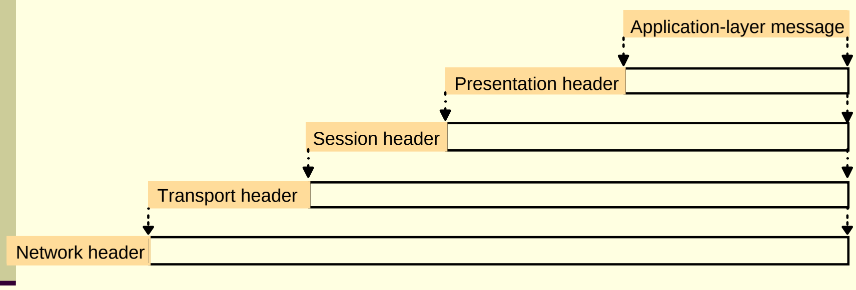
Network types

	Range	Bandwidth (Mbps)	Latency (ms)
LAN	1-2 kms	10-1000	1-10
WAN	worldwide	0.010-600	100-500
MAN	2-50 kms	1-150	10
Wireless LAN	0.15-1.5 km	2-11	5-20
Wireless WAN	worldwide	0.010-2	100-500
Internet	worldwide	0.010-2	100-500

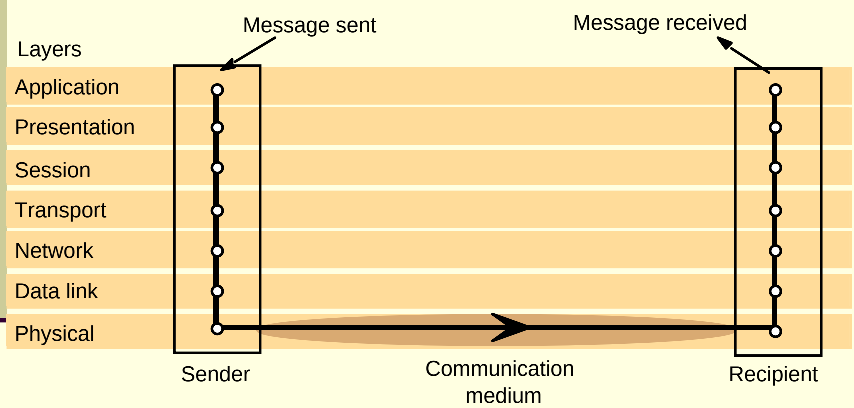
Conceptual layering of protocol software



Encapsulation as it is applied in layered protocols



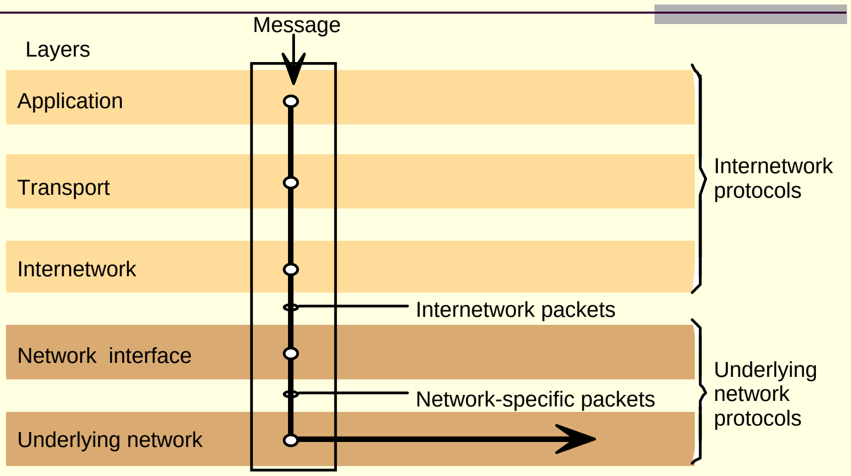
Protocol layers in the ISO Open Systems Interconnection (OSI) model



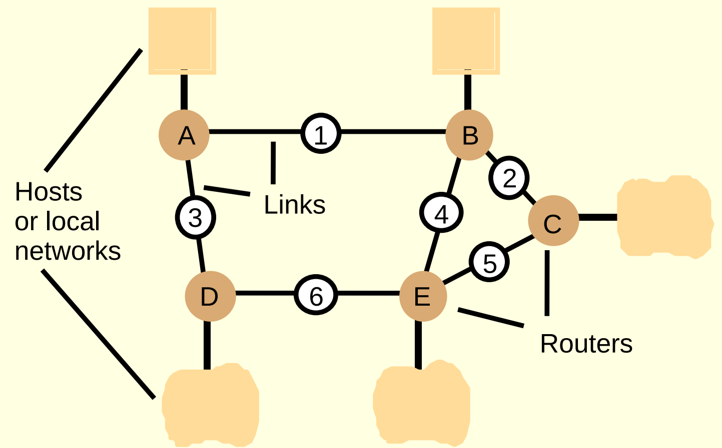
OSI protocol summary

Layer	*Description*	*Examples*
Application	Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.	HTTP,FTP, SMTP, CORBA IIOP
Presentation	Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ. Encryption is also performed in this layer, if required.	Secure Sockets (SSL),CORBA Data Rep.
Session	At this level reliability and adaptation are performed, such as detection of failures and automatic recovery.	
Transport	This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes, Protocols in this layer may be connection-oriented or connectionless.	TCP, UDP
Network	Transfers data packets between computers in a specific network. In a WAN or an internetwork this involves the generation of a route passing through routers. In a single LAN no routing is required.	IP, ATM virtual circuits
Data link	Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN it is between any pair of hosts.	Ethernet MAC, ATM cell transfer, PPP
Physical	The circuits and hardware that drive the network. It transmits sequences of binary data by analogue signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits).	Ethernet base- band signalling, ISDN

Internetwork layers



Routing in a wide area network



Routing tables for the later network

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Routings from B

To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

Routings from C

To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D

To	*Link*	*Cost*
A	3	1
B	3	2
C	6	2
D	local	0
E	6	1

Routings from E

To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

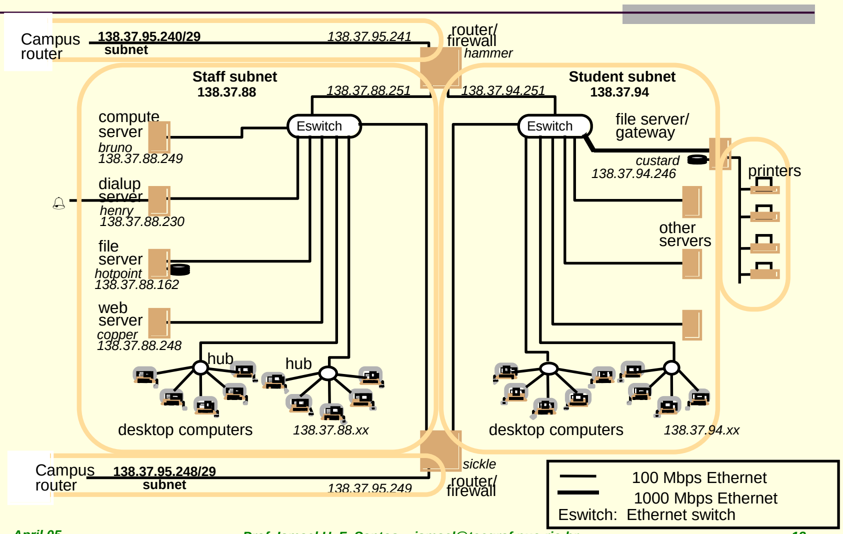
Pseudo-code for RIP routing algorithm

Send: Each *t* seconds or when *Tl* changes, send *Tl* on each non-faulty outgoing link.

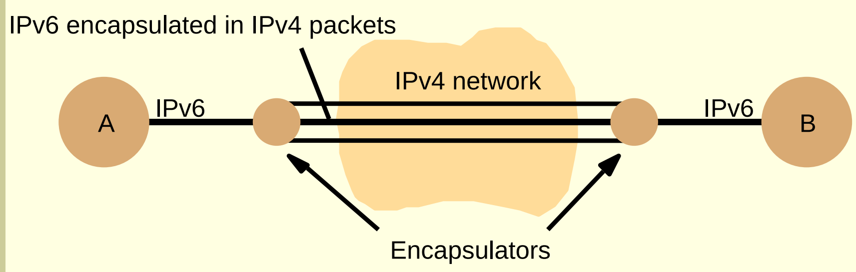
Receive: Whenever a routing table *Tr* is received on link *n*:

for all rows *Rr* in *Tr* {
    if (*Rr.link* | *n*) {
        Rr.cost = *Rr.cost* + 1;
        Rr.link = *n*;
        if (*Rr.destination* is not in *Tl*) add *Rr* to *Tl;*
        // add new destination to *Tl*
        else for all rows *Rl* in *Tl* {
            if (*Rr.destination* = *Rl.destination* and
                    (*Rr.cost* < *Rl.cost* or *Rl.link* = *n*)) *Rl* = *Rr;*
            // Rr.cost < Rl.cost : remote node has better route
            // *Rl.link* = *n* : remote node is more authoritative
        }
    }
}

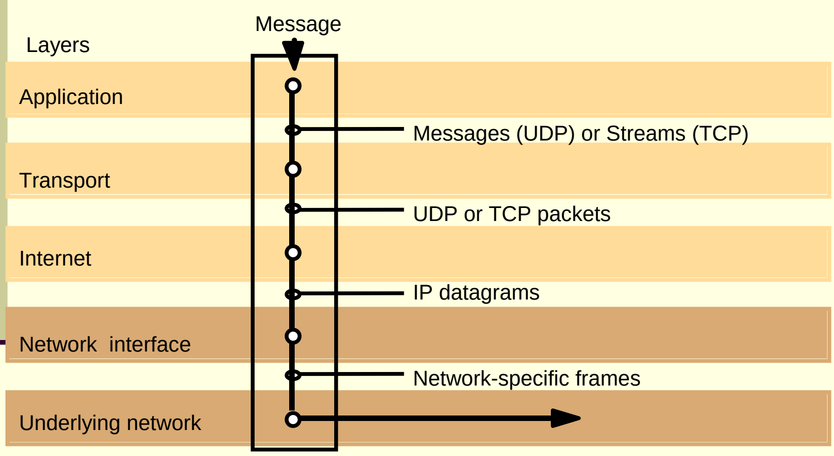
Simplified view of the QMW Computer Science network



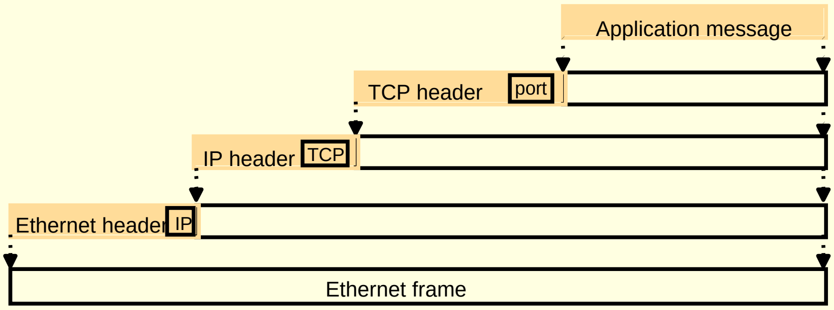
Tunnelling for IPv6 migration



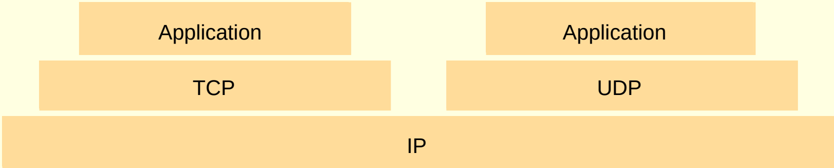
TCP/IP layers



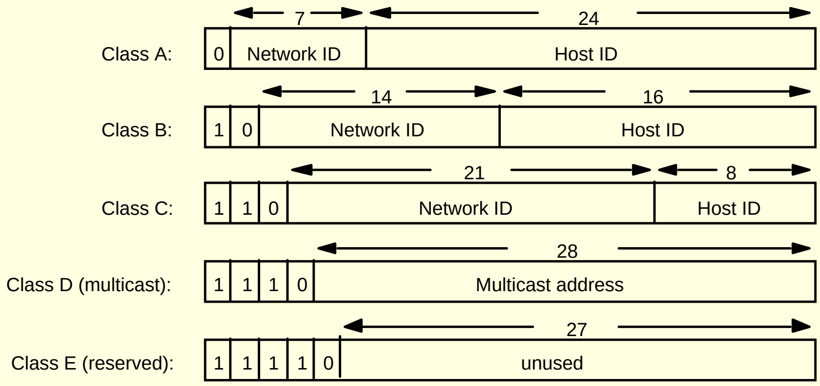
Encapsulation in a message transmitted via TCP over an Ethernet



The programmer's conceptual view of a TCP/IP Internet



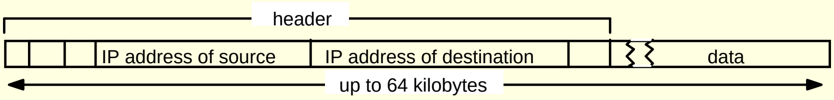
Internet address structure, showing field sizes in bits



Decimal representation of Internet addresses

	octet 1	octet 2	octet 3		Range of addresses
Class A:	Network ID: 1 to 127	Host ID: 0 to 255	0 to 255	0 to 255	1.0.0.0 to 127.255.255.255
Class B:	Network ID: 128 to 191	0 to 255	Host ID: 0 to 255	0 to 255	128.0.0.0 to 191.255.255.255
Class C:	Network ID: 192 to 223	0 to 255	0 to 255	Host ID: 1 to 254	192.0.0.0 to 223.255.255.255
Class D (multicast):	Multicast address: 224 to 239	0 to 255	0 to 255	1 to 254	224.0.0.0 to 239.255.255.255
Class E (reserved):	240 to 255	0 to 255	0 to 255	1 to 254	240.0.0.0 to 255.255.255.255

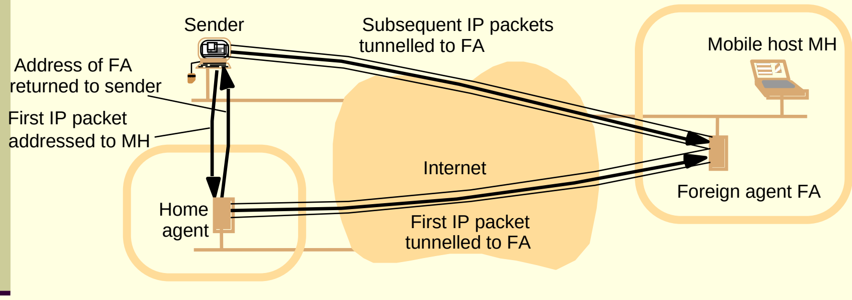
IP packet layout



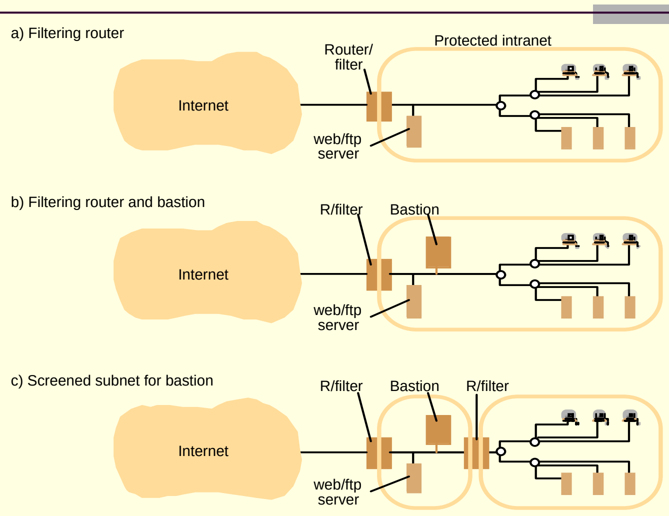
IPv6 header layout

Version (4 bits)	Priority (4 bits)	Flow label (24 bits)		
Payload length (16 bits)			Next header (8 bits)	Hop limit (8 bits)
Source address (128 bits)				
Destination address (128 bits)				

The MobileIP routing mechanism



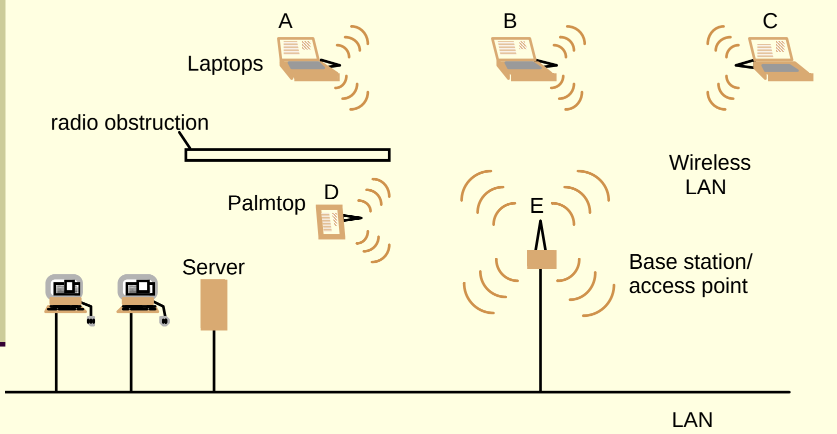
Firewall configurations



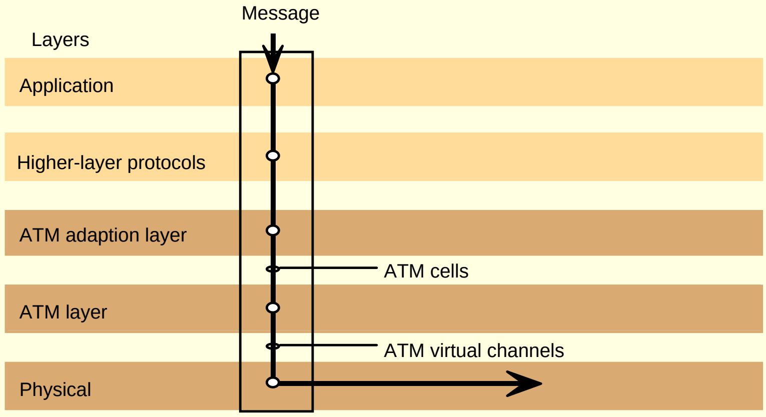
IEEE 802 network standards

IEEE No.	*Title*	*Reference*
802.3	CSMA/CD Networks (Ethernet)	[IEEE 1985a]
802.4	Token Bus Networks	[IEEE 1985b]
802.5	Token Ring Networks	[IEEE 1985c]
802.6	Metropolitan Area Networks	[IEEE 1994]
802.11	Wireless Local Area Networks	[IEEE 1999]

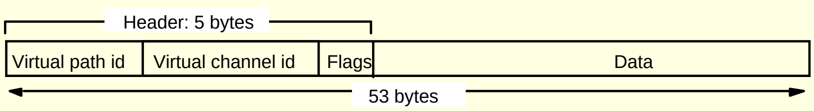
Wireless LAN configuration



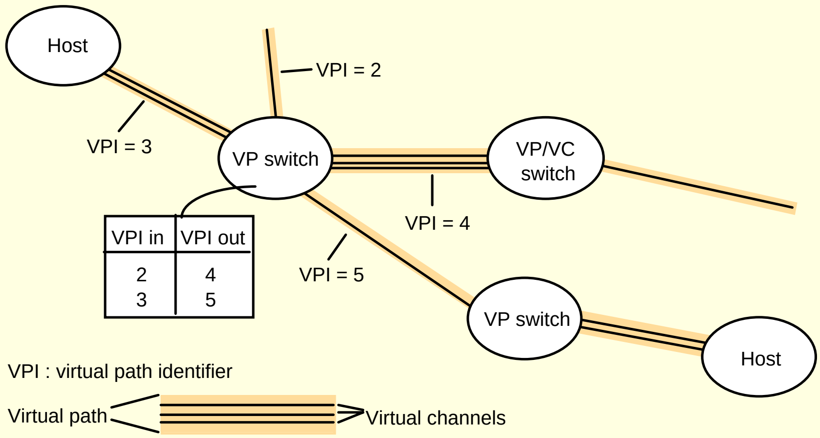
ATM protocol layers



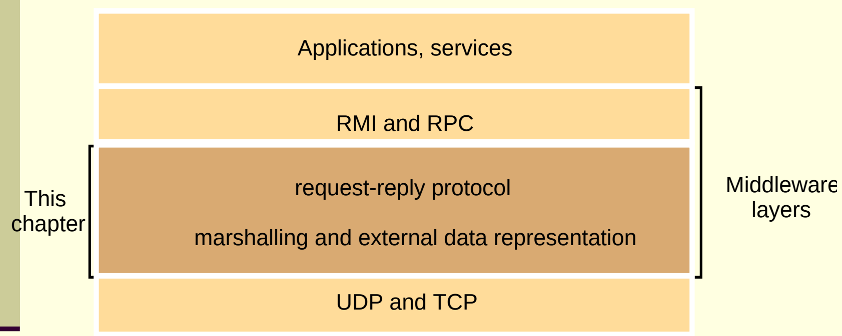
ATM cell layout



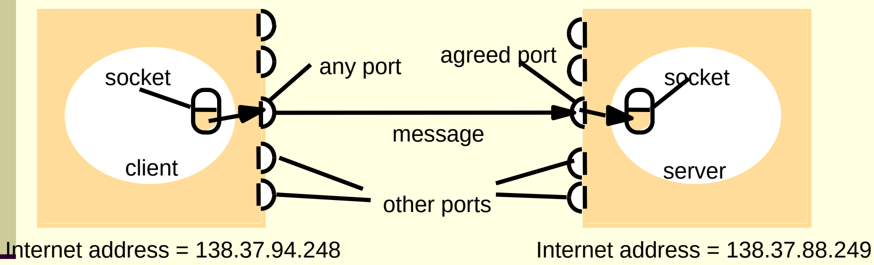
Switching virtual paths in an ATM network



Middleware layers



Sockets and ports



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 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 see Sn 4.3
            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 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
```


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*/}}
   }
}
```

CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

CORBA CDR message

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h "	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on "	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Indication of Java serialized form

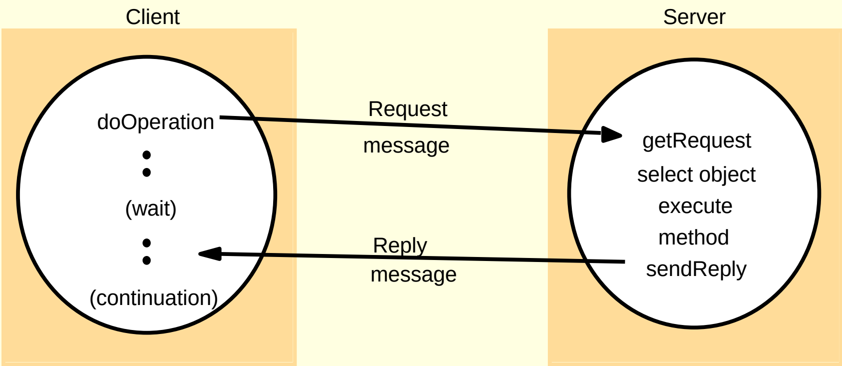
	Serialized values			*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

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

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

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*

HTTP request message

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

HTTP reply message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

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();}
   }
}
```

Sockets used for datagrams

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

ServerAddress and *ClientAddress* are socket addresses

Sockets used for streams

Requesting a connection

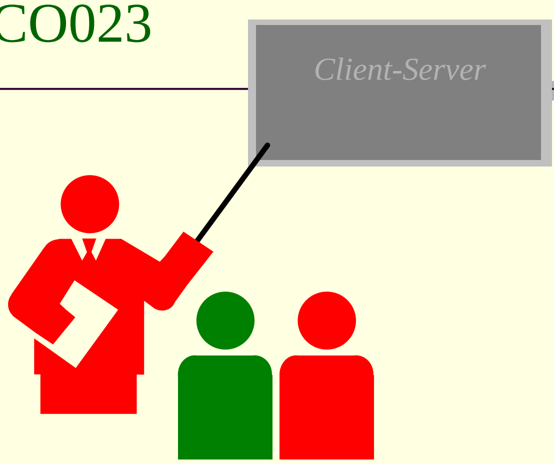
```
s = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

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

SCD – CO023



Client-Server Communication

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

Sockets

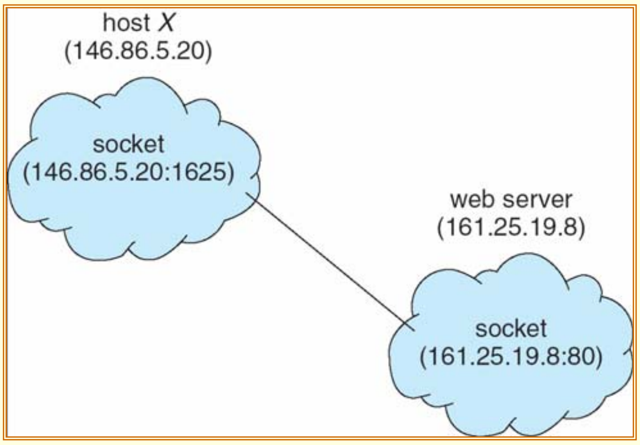
- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All Ports < 1024 are Considered "well-known"

  - TELNET uses port 23

  - FTP uses port 21

  - HTTP server uses port 80

Socket Communication



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.

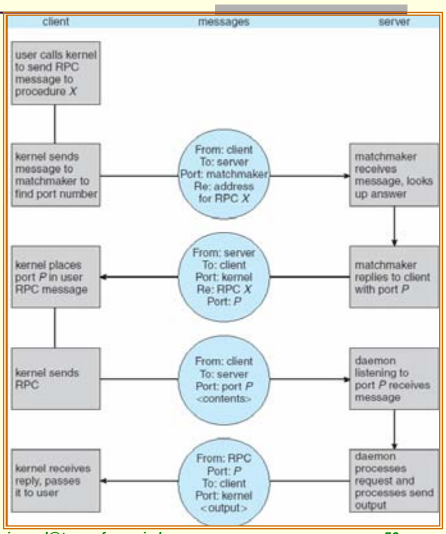
```
Socket s = new Socket(“127.0.0.1”,5155);
```

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.



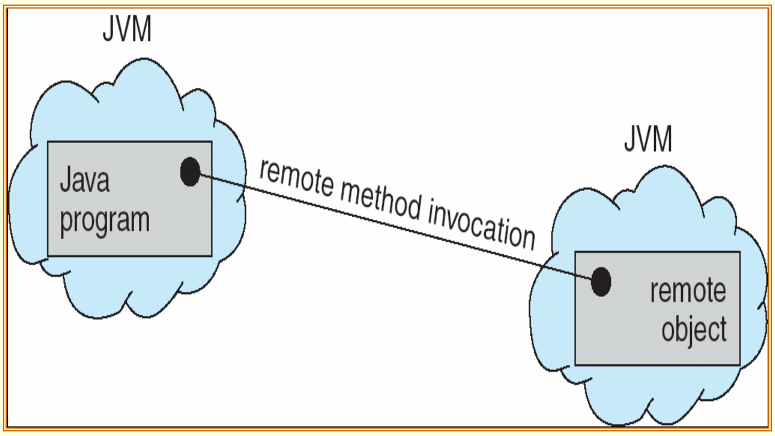
Stubs and Skeletons

- "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.

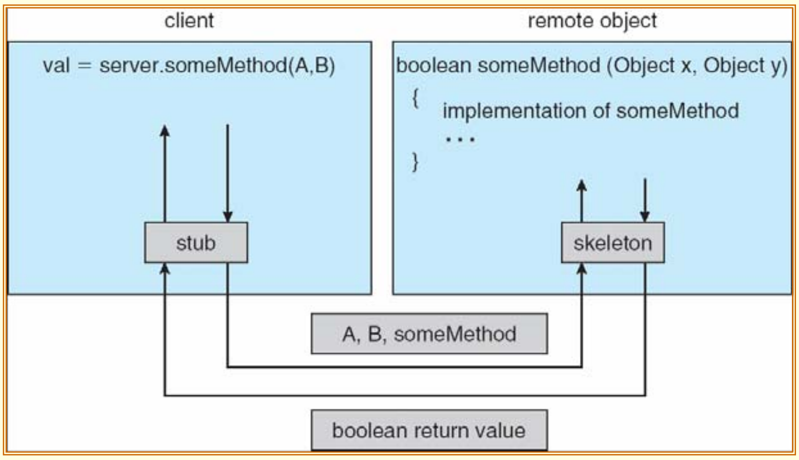
Remote Method Invocation

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

Remote Method Invocation



Marshalling Parameters



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

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

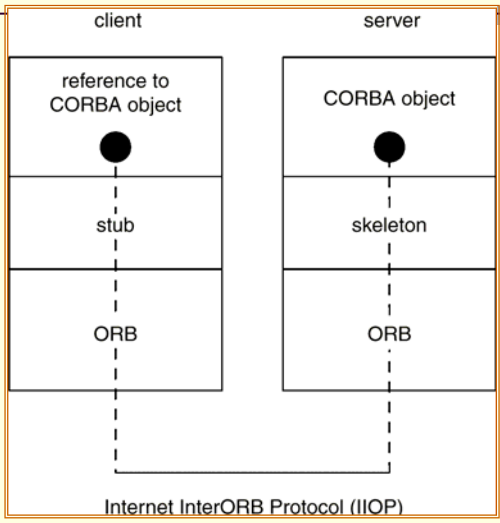
- New with Java 2

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

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.

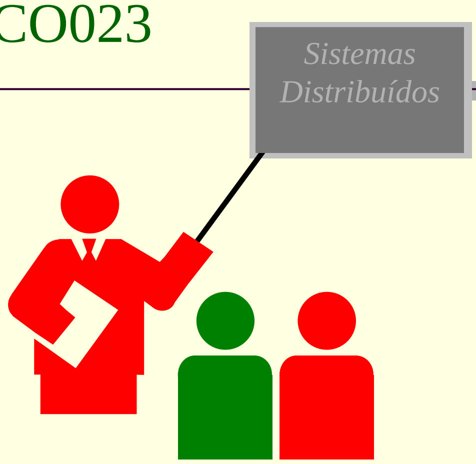
Cobra Model



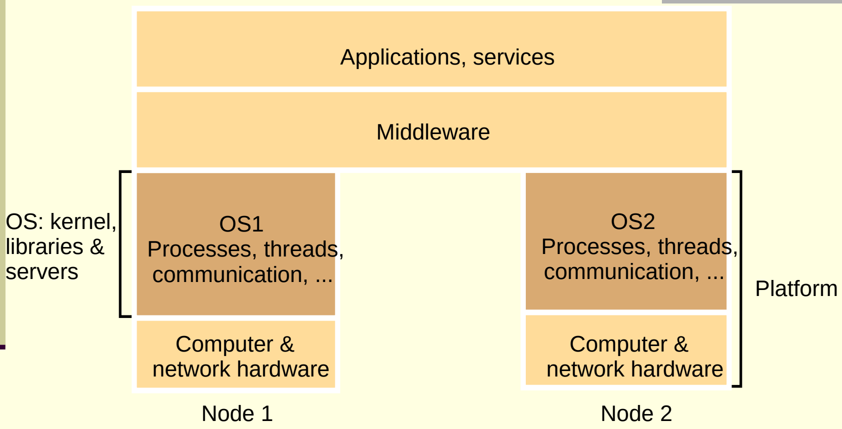
Registration Services

- Registration Service Allows Remote Objects to "register" Their Services.
- RMI, CORBA Require Registration Services

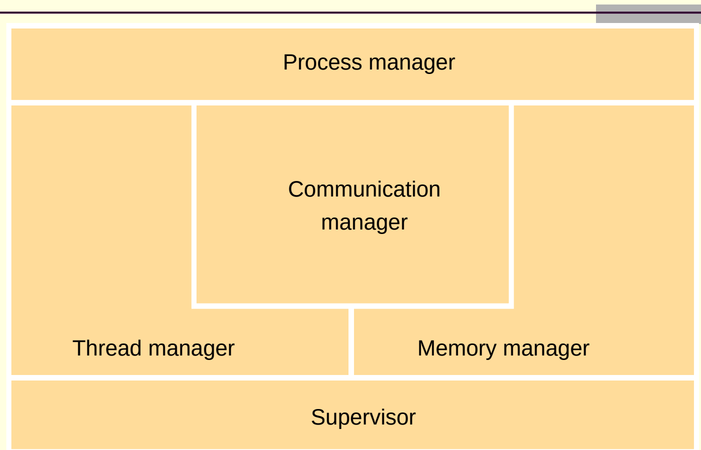
SCD – CO023



System layers



Core OS functionality



Address space

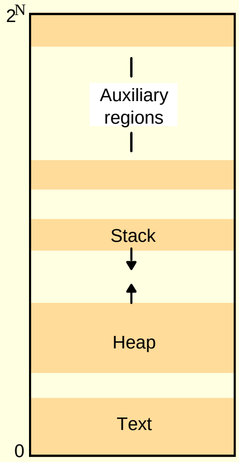


Figure 6.4
Copy-on-write

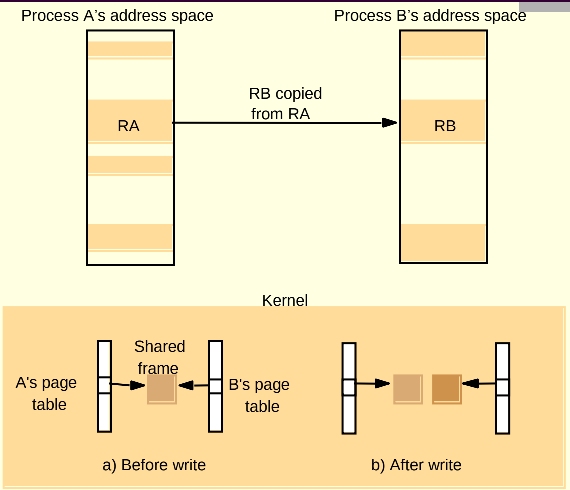


Figure 6.5
Client and server with threads

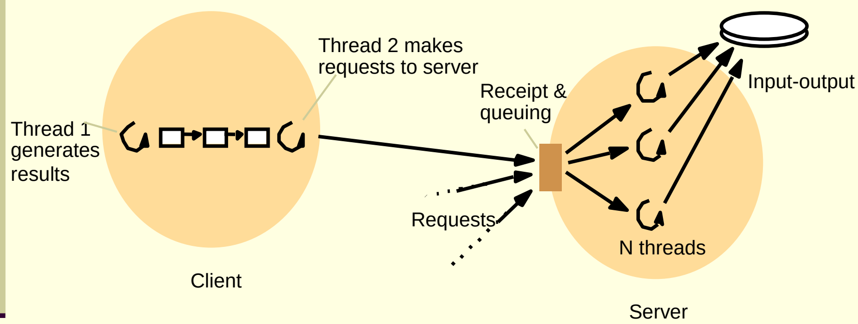
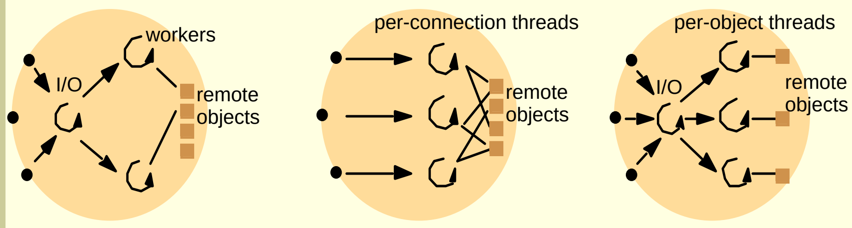


Figure 6.6
Alternative server threading architectures (see also Figure 6.5)



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

Figure 6.7
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	

Figure 6.8
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.

destroy()
Destroy the thread.

Figure 6.9
Java thread synchronization calls

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()*.

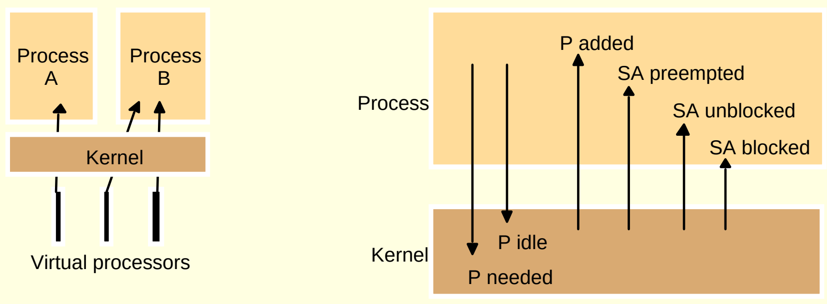
object.wait(long millisecs, int nanosecs)

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*.

Figure 6.10
Scheduler activations



A. Assignment of virtual processors to processes

B. Events between user-level scheduler & kerne
Key: P = processor; SA = scheduler activatior

Figure 6.11
Invocations between address spaces

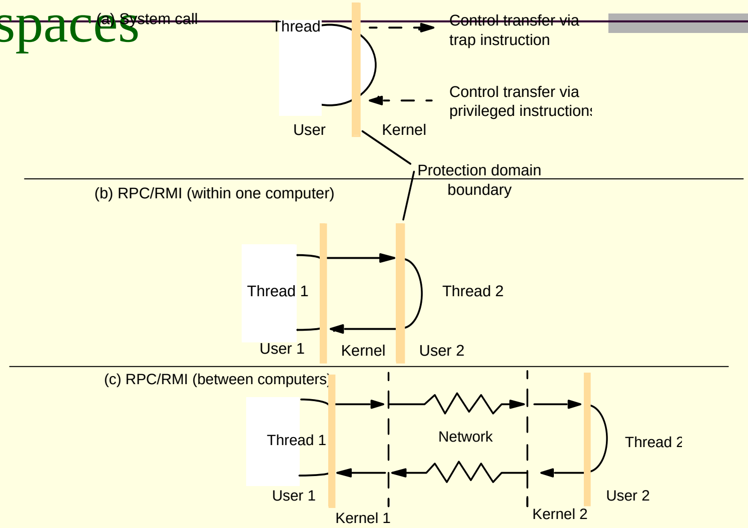


Figure 6.12
RPC delay against parameter size

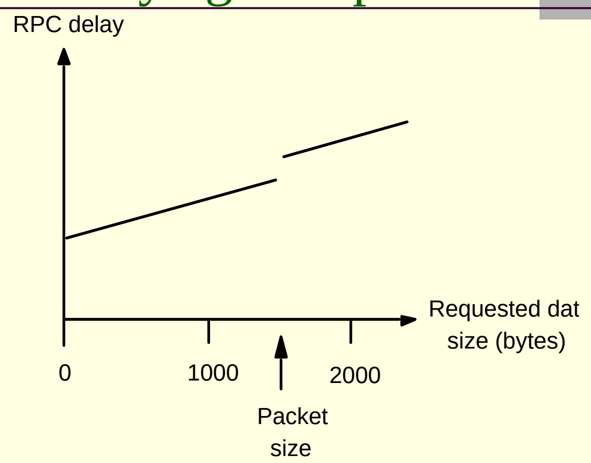


Figure 6.13
A lightweight remote procedure call

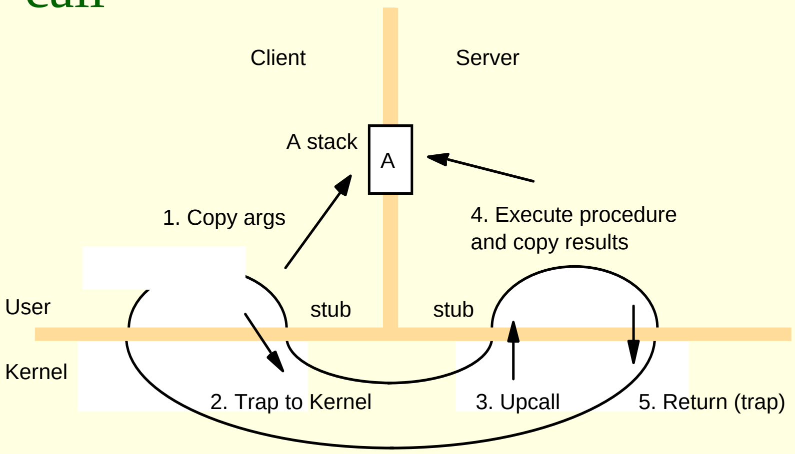


Figure 6.14
Times for serialized and concurrent invocations

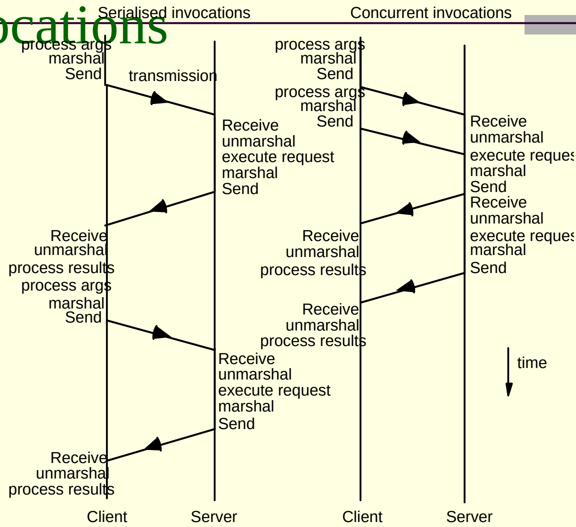


Figure 6.15
Monolithic kernel and microkernel

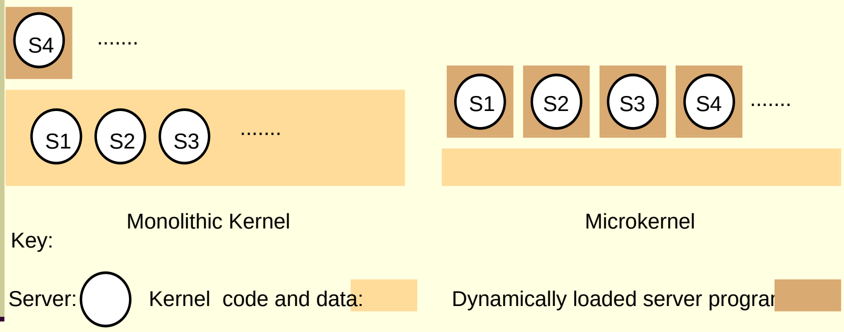
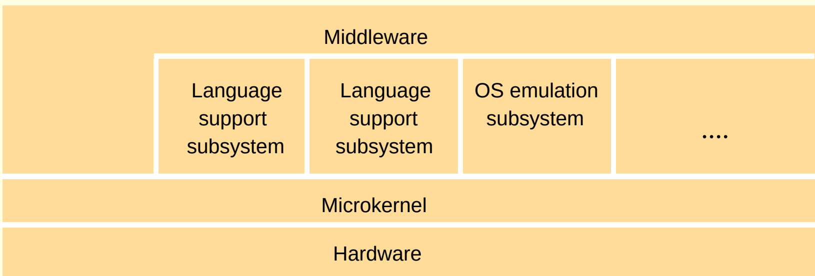


Figure 6.16
The role of the microkernel



The microkernel supports middleware via subsystems