# CHAPTER 4: COMMUNICATION

Dr. Trần Hải Anh

# Outline



1. Fundamentals
2. Remote Procedure Call
3. Message-Oriented Communication
4. Stream-Oriented Communication

# 1. Fundamentals

1.1. Layered Protocols

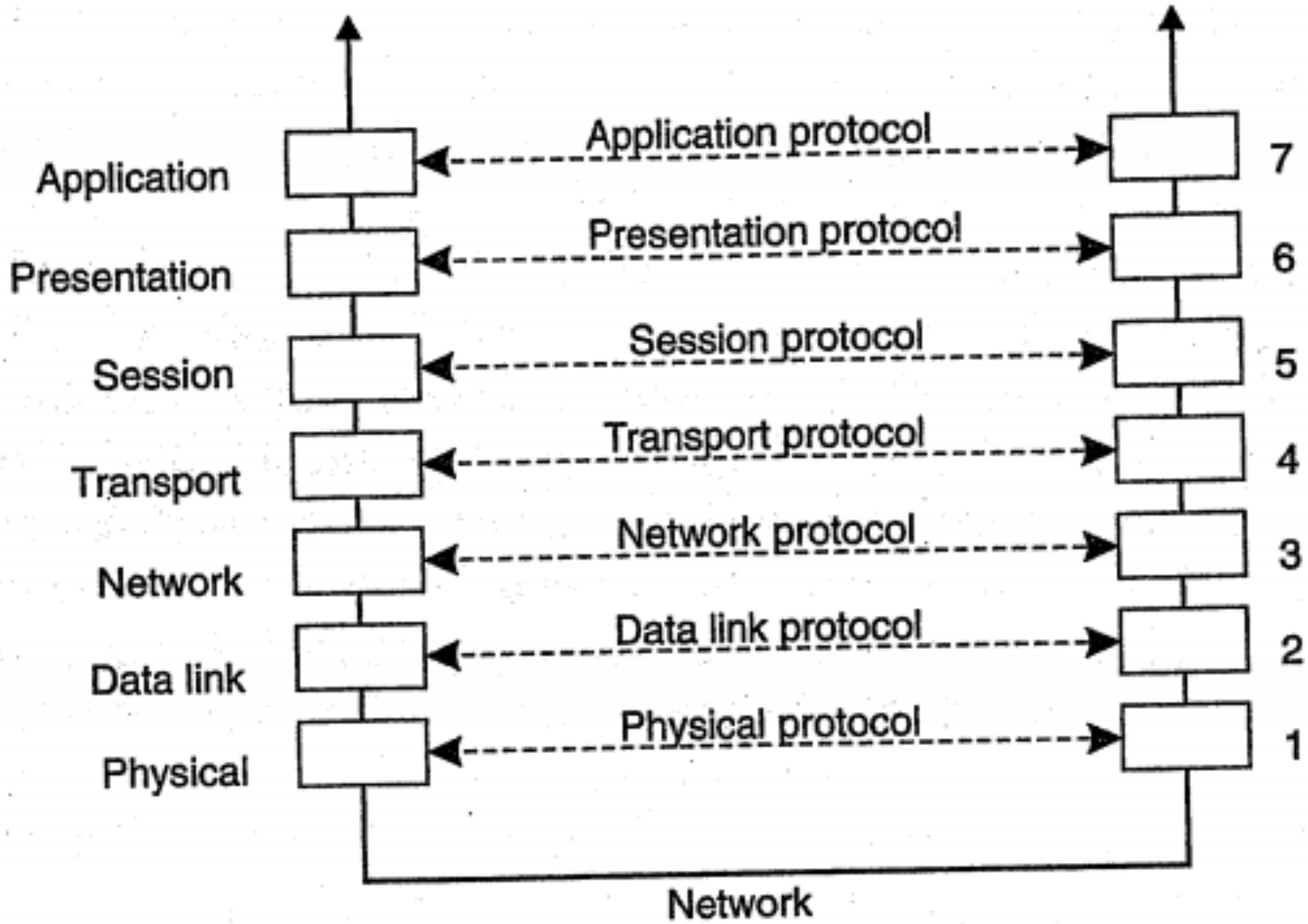
1.2. Communication with UDP

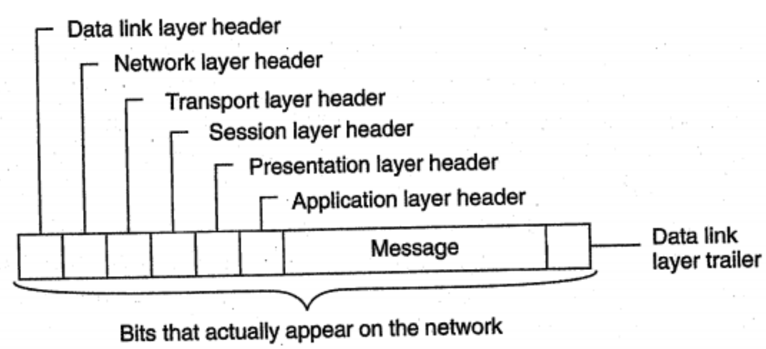
1.3. Communication with TCP

# I. Layered protocols



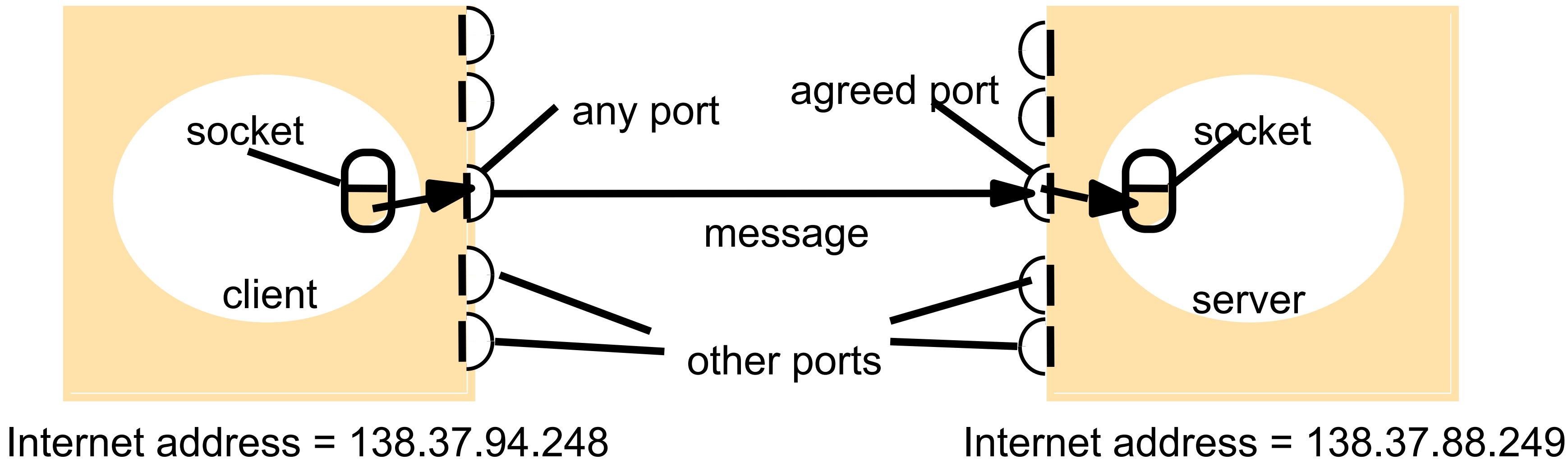
- Agreements are needed at a variety of levels, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed.
- Protocol
  - Message format
  - Message size
  - Message order
  - Faults detection method
  - Etc.
- Layered
- Protocol types:
  - Connection oriented/connectionless protocols, Reliable/Unreliable protocols
- Protocol issues:
  - Send, receive primitives
  - Synchronous, Asynchronous, Blocking or non-blocking


Application
Presentation
Session
Transport
Network
Data link
Physical
Application protocol
Presentation protocol
Session protocol
Transport protocol
Network protocol
Data link protocol
Physical protocol
7
6
5
4
3
2
1
Network

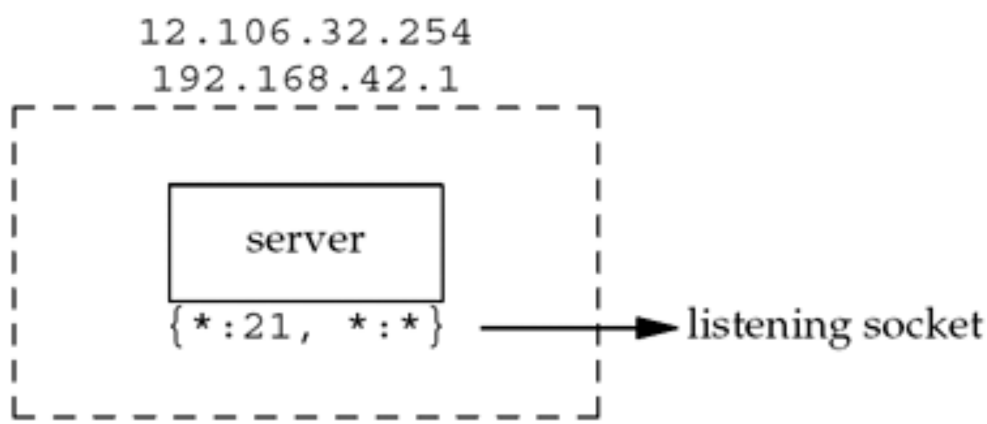

Data link layer header
Network layer header
Transport layer header
Session layer header
Presentation layer header
Application layer header
Message
Data link layer trailer
Bits that actually appear on the network

# Socket-port

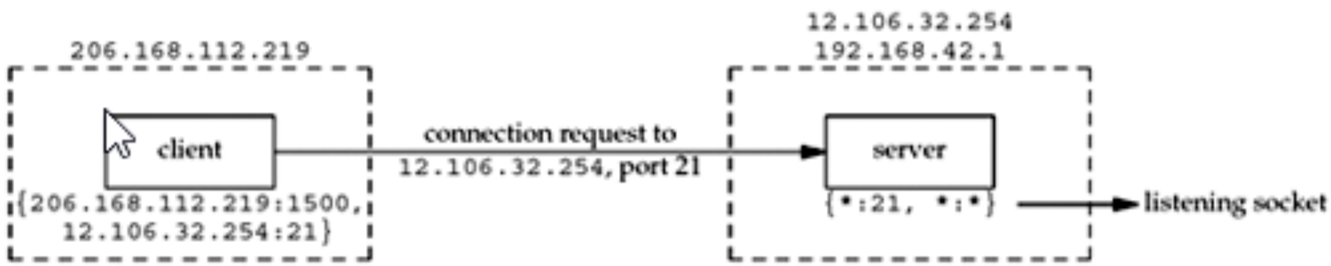



socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

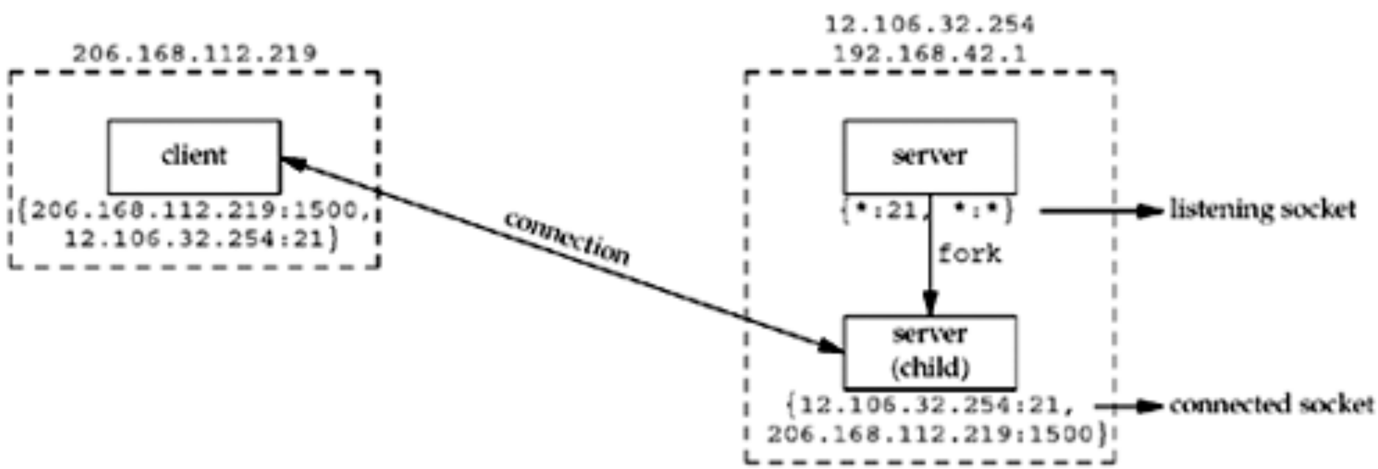
# TCP Port Numbers and Concurrent Servers (1)



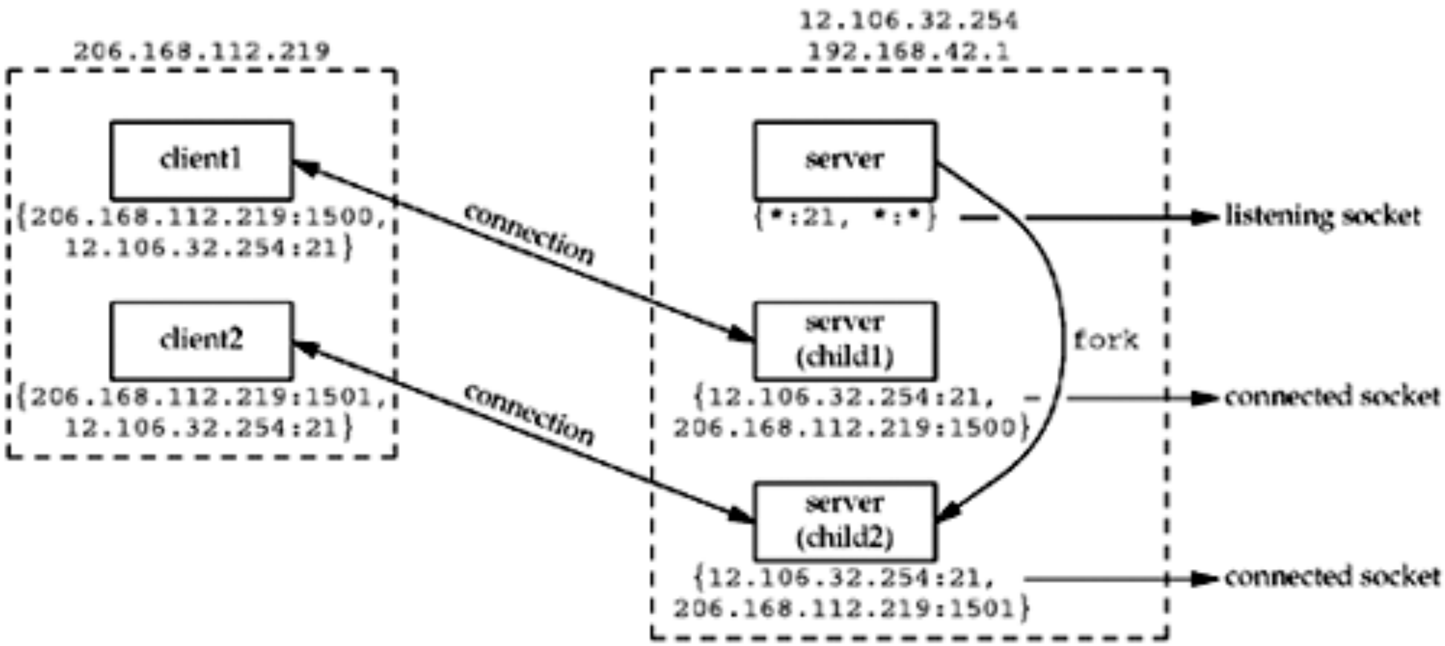
# TCP Port Numbers and Concurrent Servers (2)



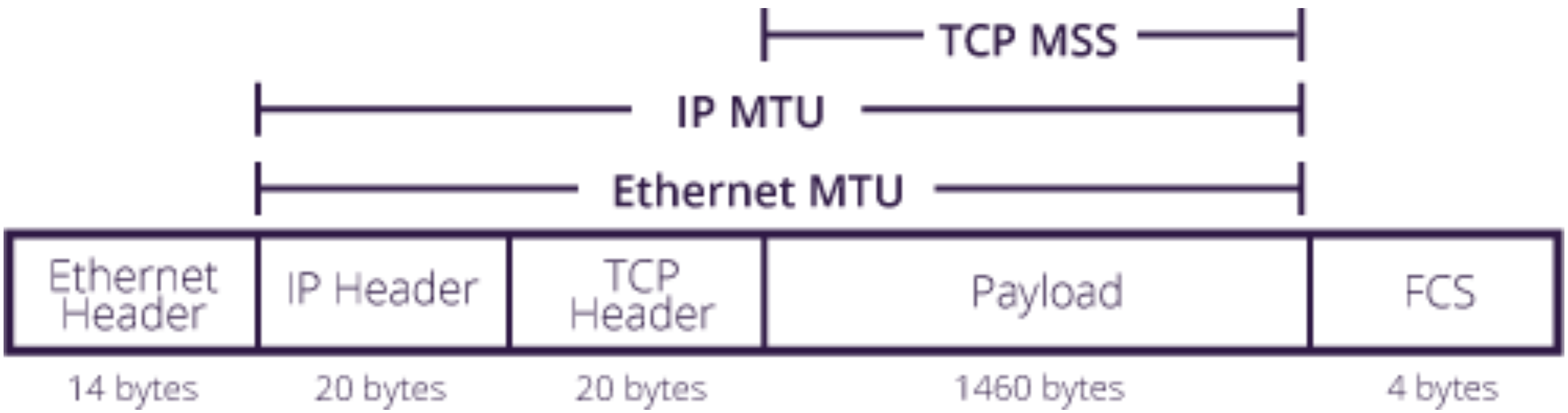
# TCP Port Numbers and Concurrent Servers (3)



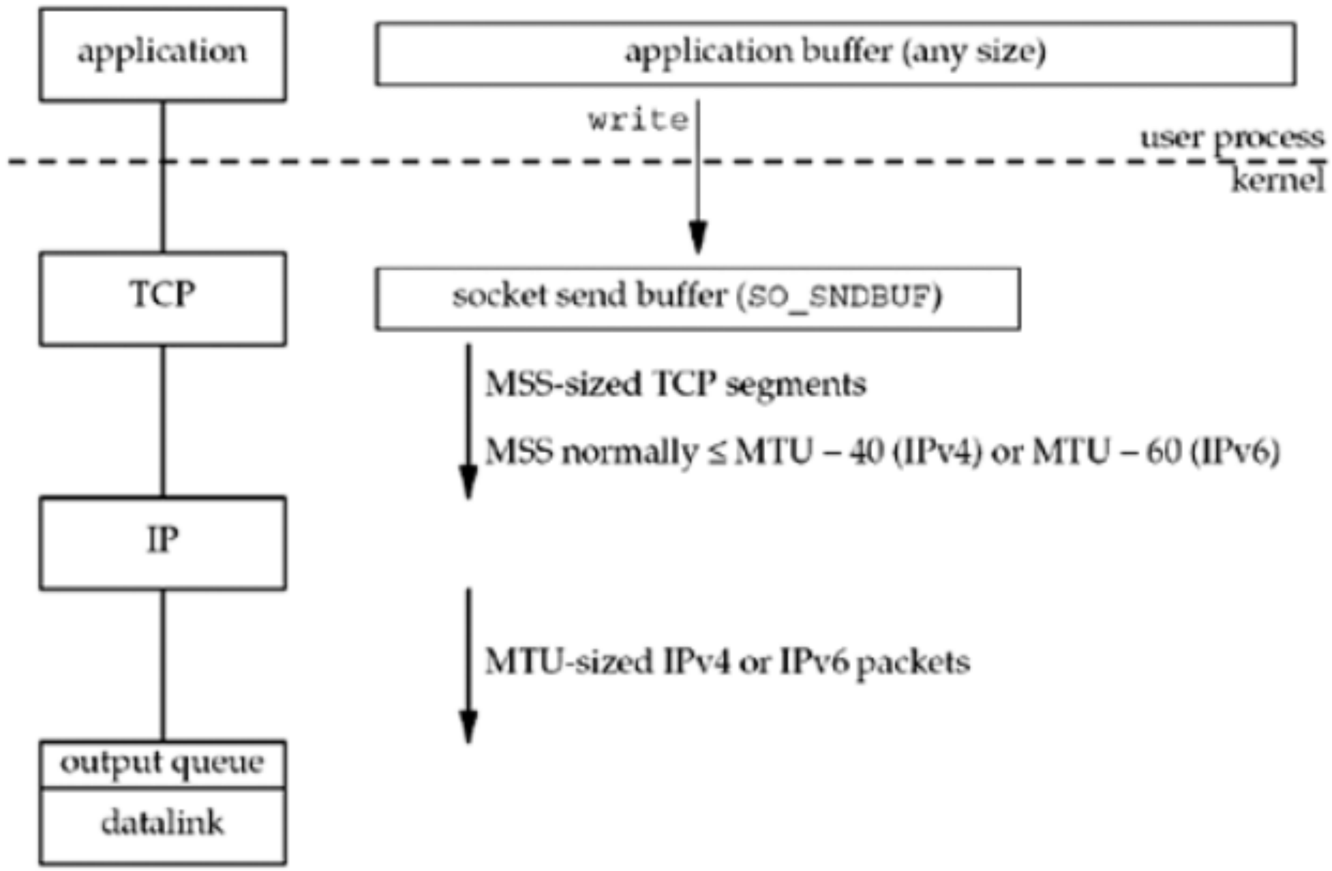
# TCP Port Numbers and Concurrent Servers (4)



# Buffer Sizes and Limitations

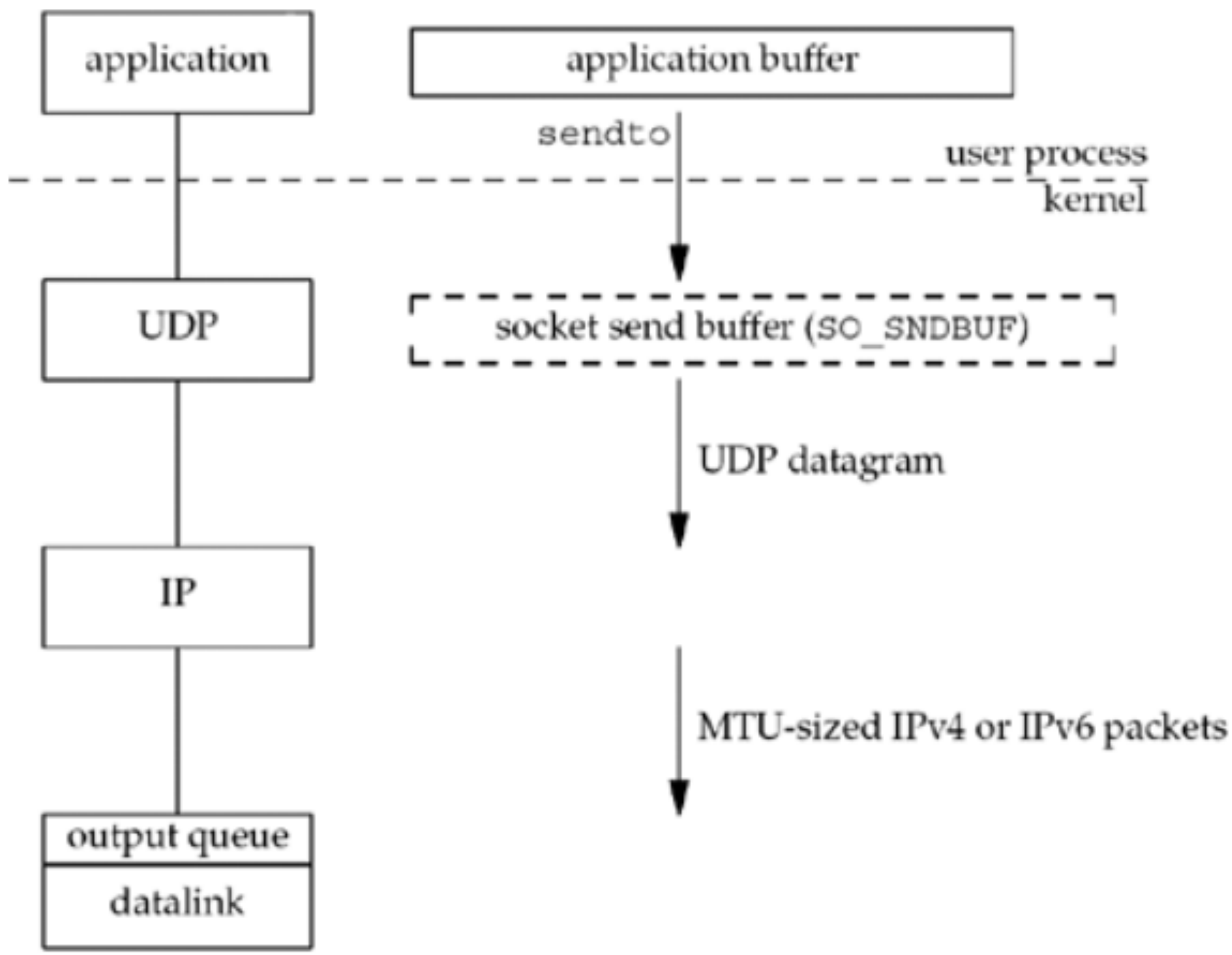


# TCP output



application
application buffer (any size)
write
user process
kernel
TCP
socket send buffer (SO_SNDBUF)
MSS-sized TCP segments
MSS normally ≤ MTU – 40 (IPv4) or MTU – 60 (IPv6)
IP
MTU-sized IPv4 or IPv6 packets
output queue
datalink

# UDP output


application
application buffer
sendto
user process
kernel
UDP
socket send buffer (SO_SNDBUF)
UDP datagram
IP
MTU-sized IPv4 or IPv6 packets
output queue
datalink

# In Java



- Class InetAddress:
- Working with IP address and domain name
- `InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");`

# 1.2. Communication with UDP



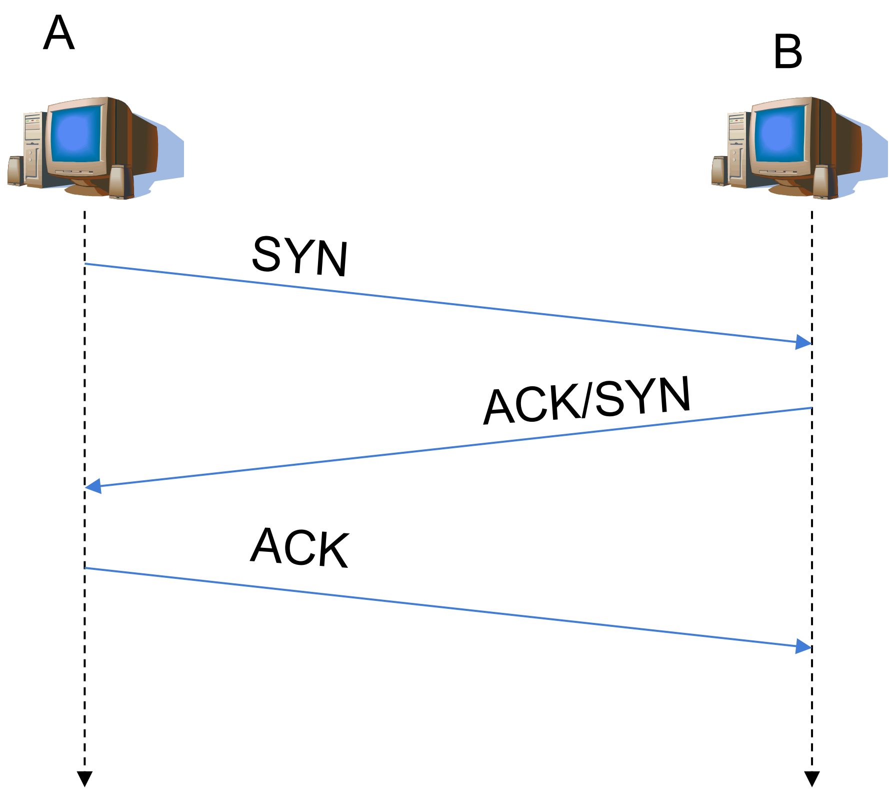
- Characteristics:
  - Connectionless
  - Unreliable
  - Asynchronous
- Issues:
  - Message size
  - Blocking (non-blocking *send* ;blocking *receive*)
  - Timeouts
  - Receive from any

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

```
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,  m.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();}
  }
}
```

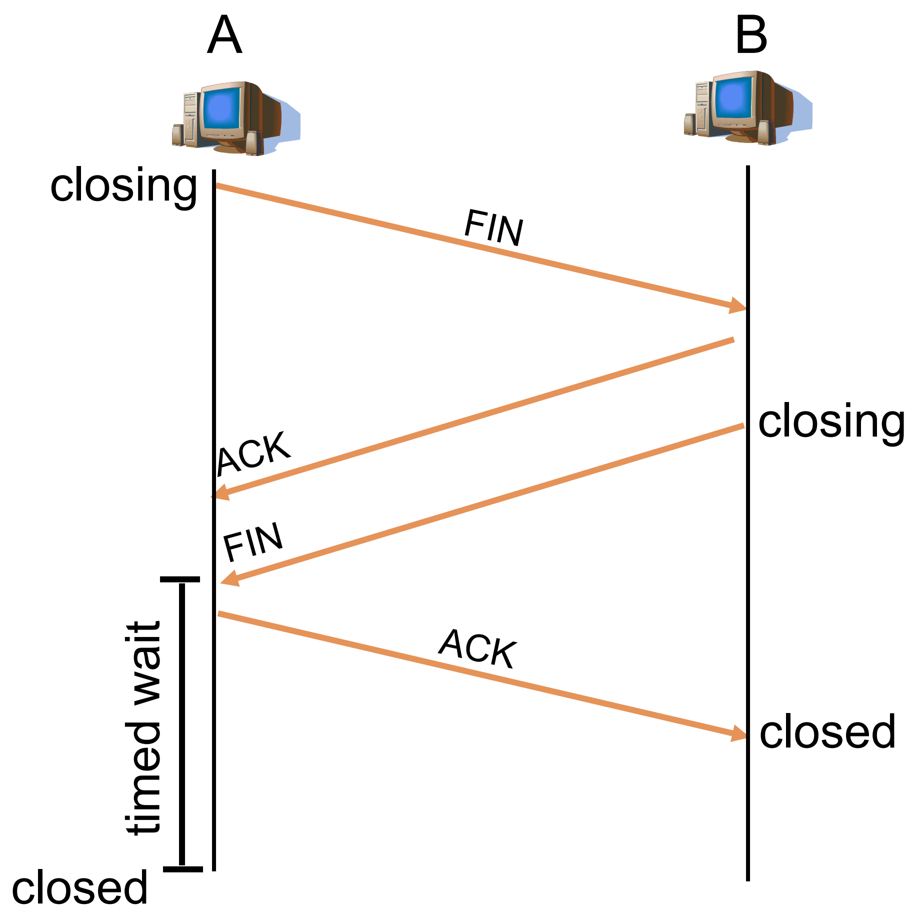
# 1.3. Communication with TCP-IP





# Closing the connection





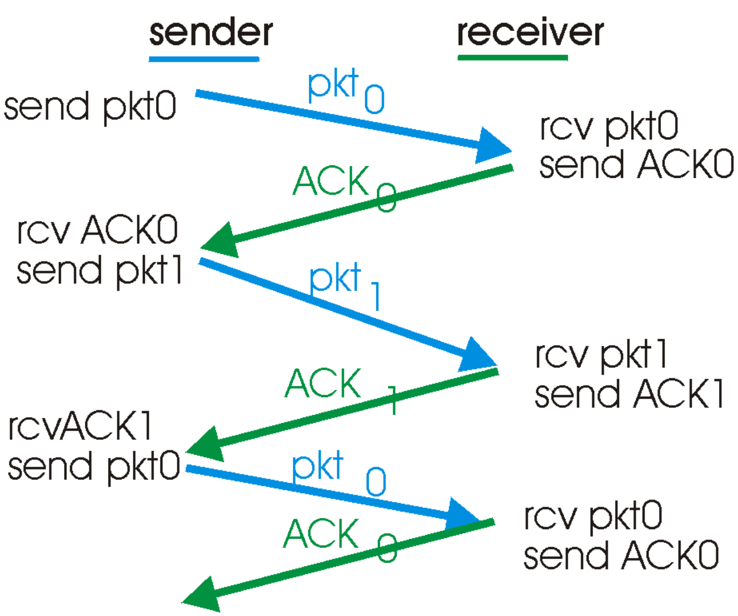
# Communication with TCP



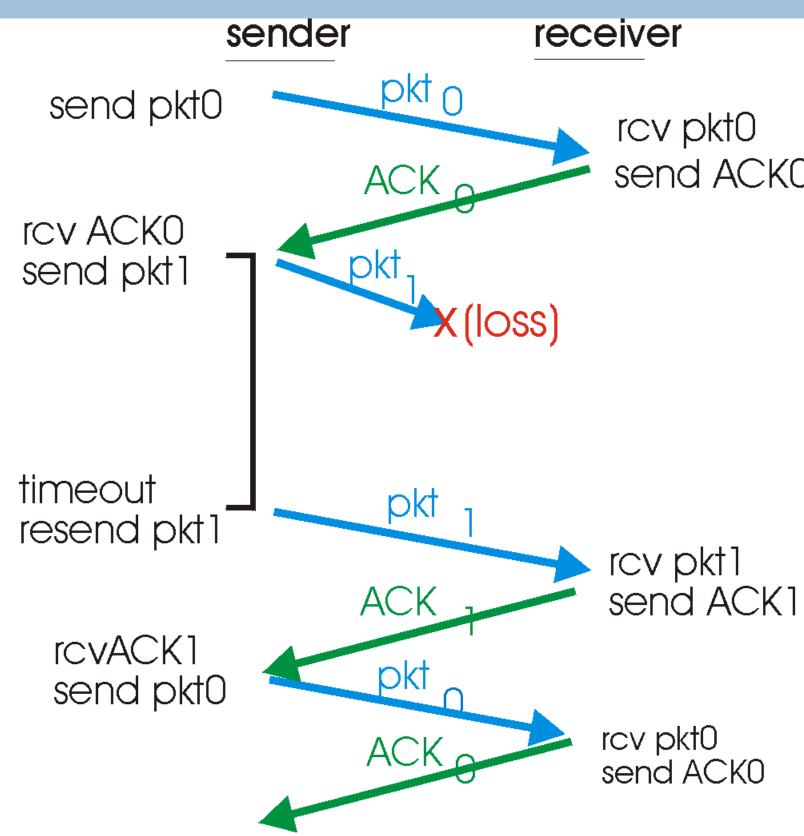
- Data type matching
- Synchronization
- Mutithreaded/Multi-processes server
- Reliability

# Solutions for some problems





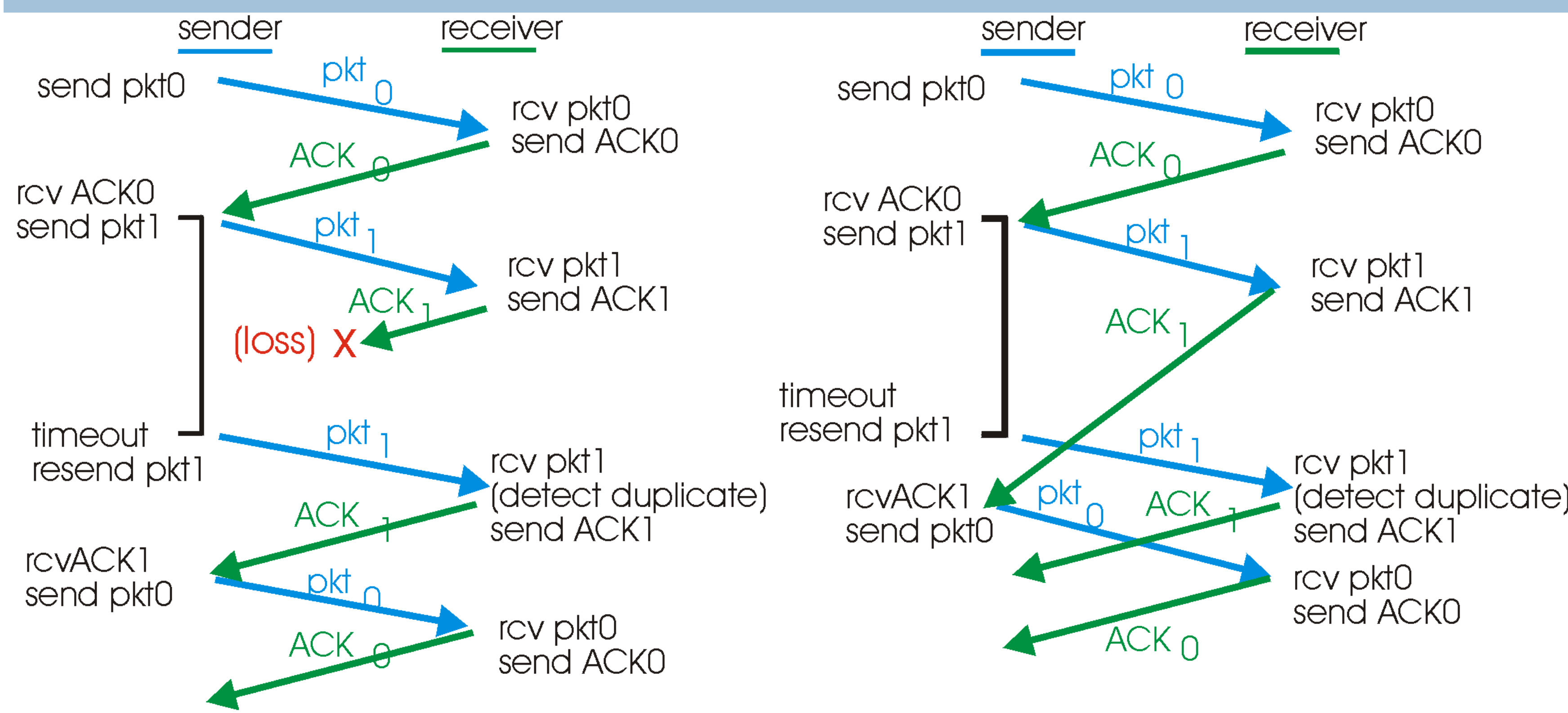

(a) operation with no loss




(b) lost packet

# Solutions for some problems






(c) lost ACK

(d) premature timeout

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

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

# 2. Remote Procedure Call

2.1. Request-reply protocol

2.2. RPC

2.3. RMI

# 2.1. Request-reply protocol

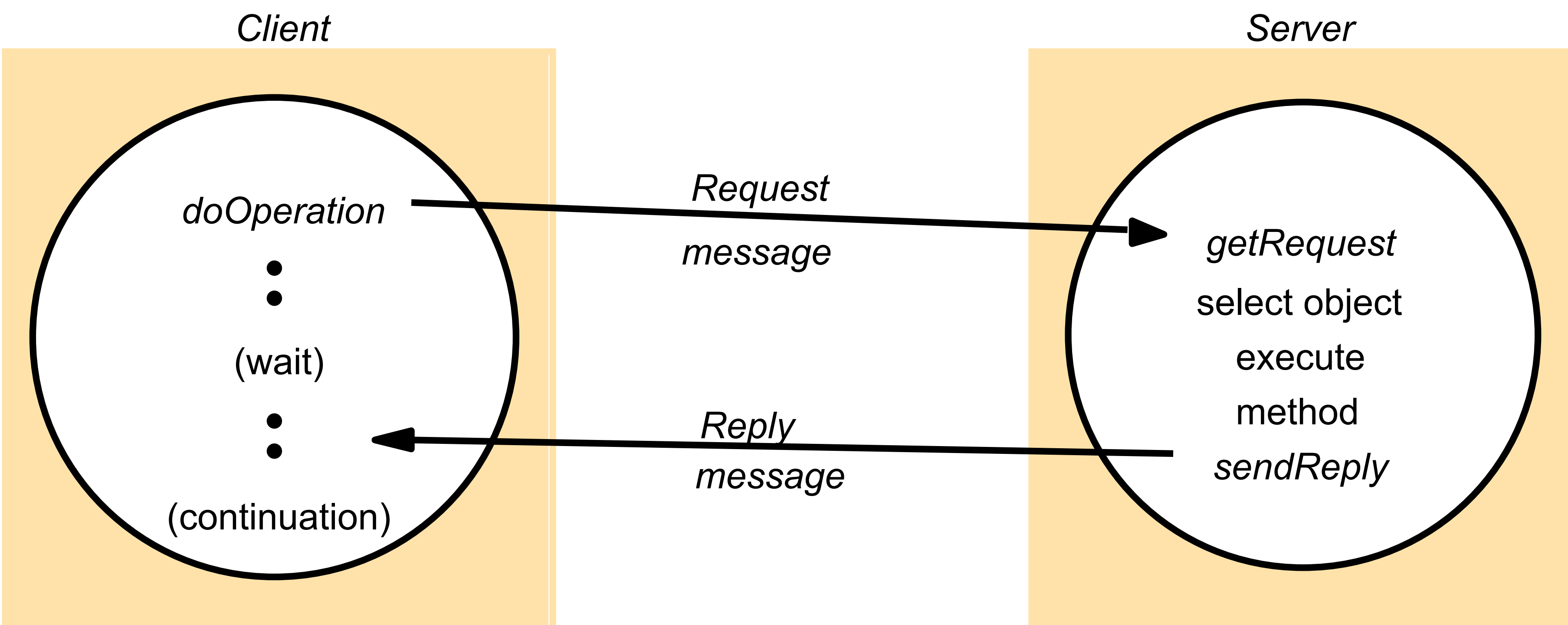


- a pattern on top of message passing
- support the two-way exchange of messages as encountered in client-server computing
- synchronous
- reliable

# Request-reply protocol



- Characteristics:
  - No need of Acknowledgement
  - No need of Flow control



# Trio of communication primitives



- *public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)*
- *public byte[] getRequest ();*
- *public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);*

# Message structure



| | |
|---|---|
| messageType | *int (0=Request, 1= Reply)* |
| requestId | *int* |
| remoteReference | *RemoteRef* |
| operationId | *int or Operation* |
| arguments | *array of bytes* |

# Example: HTTP

## 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 |

# Styles of exchange protocols

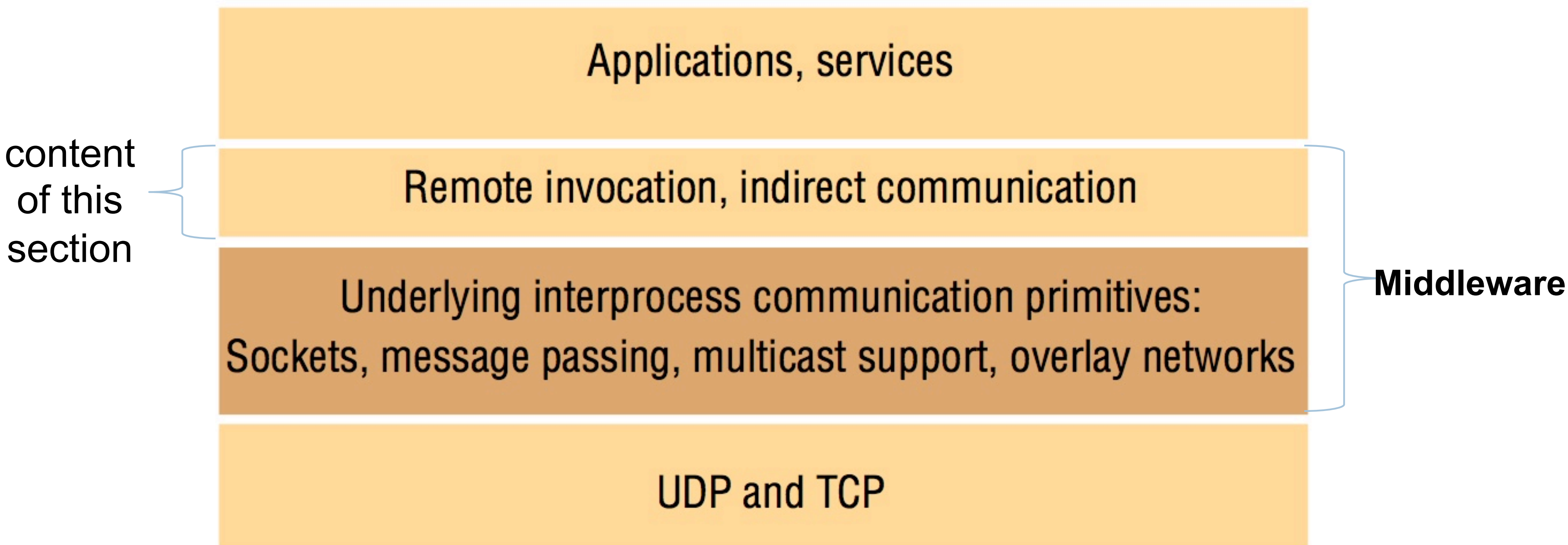


- R only protocol
- RR protocol
- RRA protocol

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

# 2.2. RPC (Remote Procedure Call)

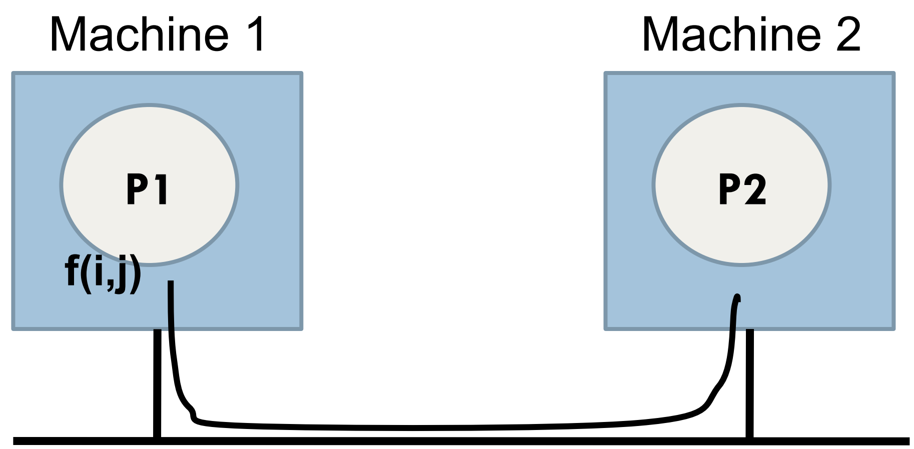




# 2.2. Remote Procedure Call



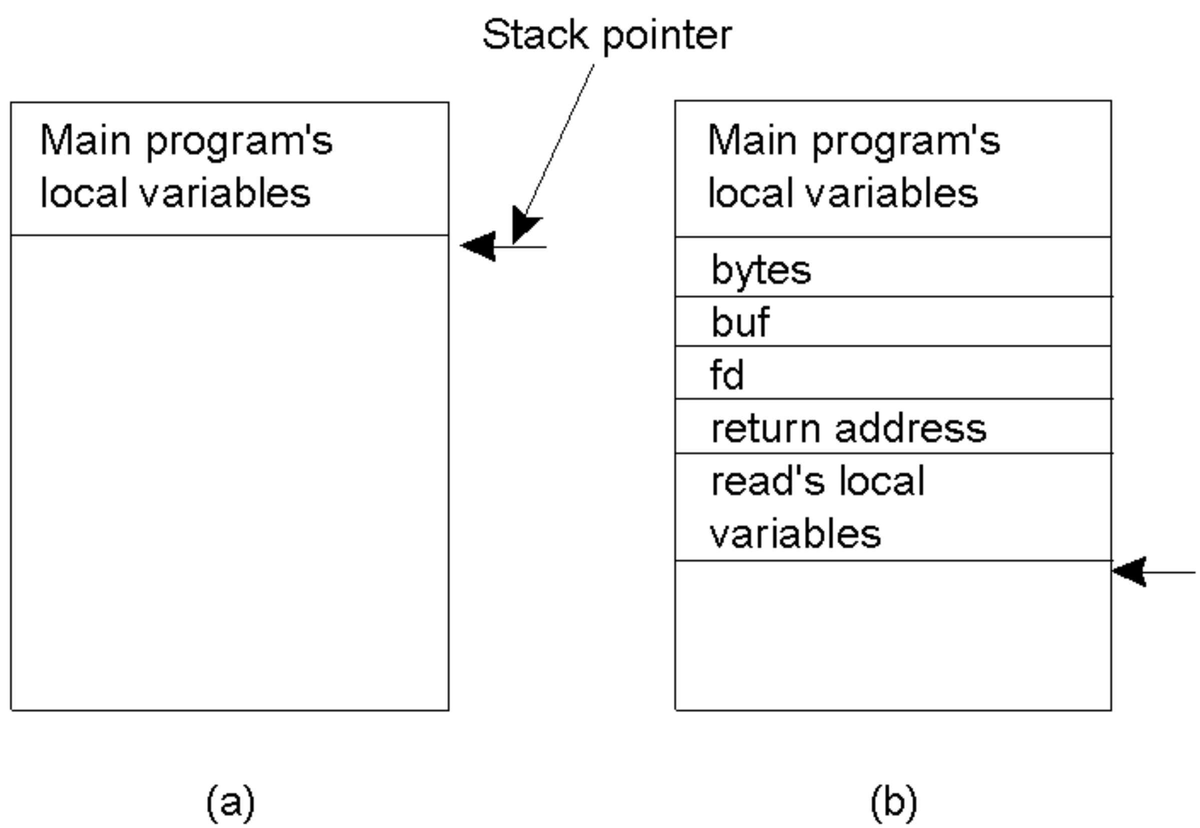
- Access transparency
- Issues:
  - Heterogenous system
    - Different memory space
    - Different information representation
  - Faults appear



# Call in C:



**count = read(fd, buf, nbytes)**



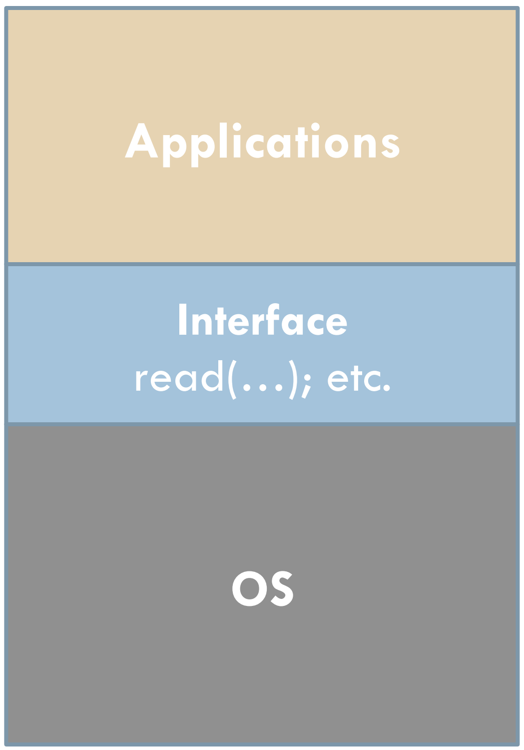
# Parameters

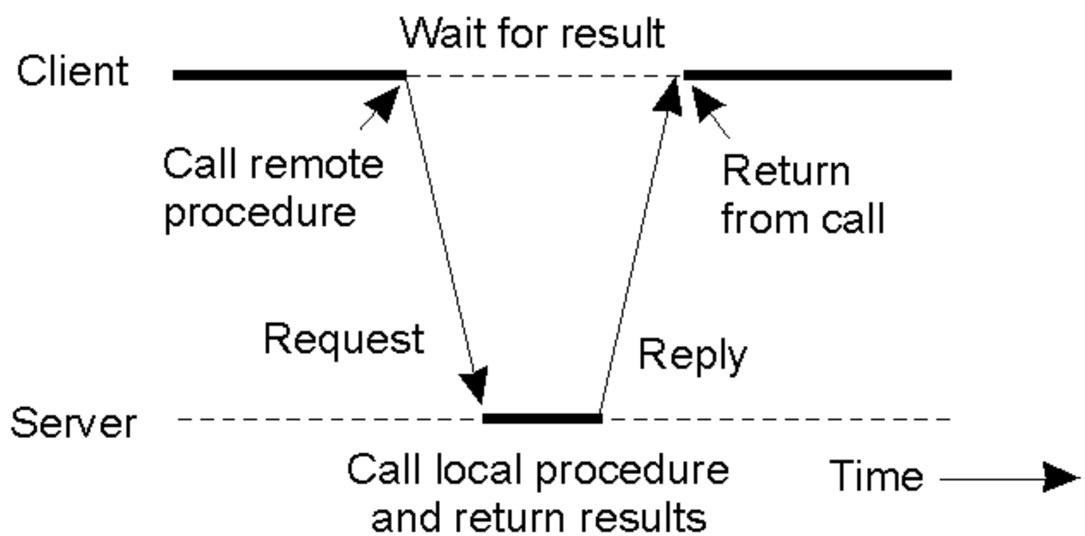


- Call-by-value
- Call-by-reference
- Call-by-copy/restore
  - Copy the variables to the stack
  - Copy back after the call, overwrite caller's the original value

# RPC mechanism

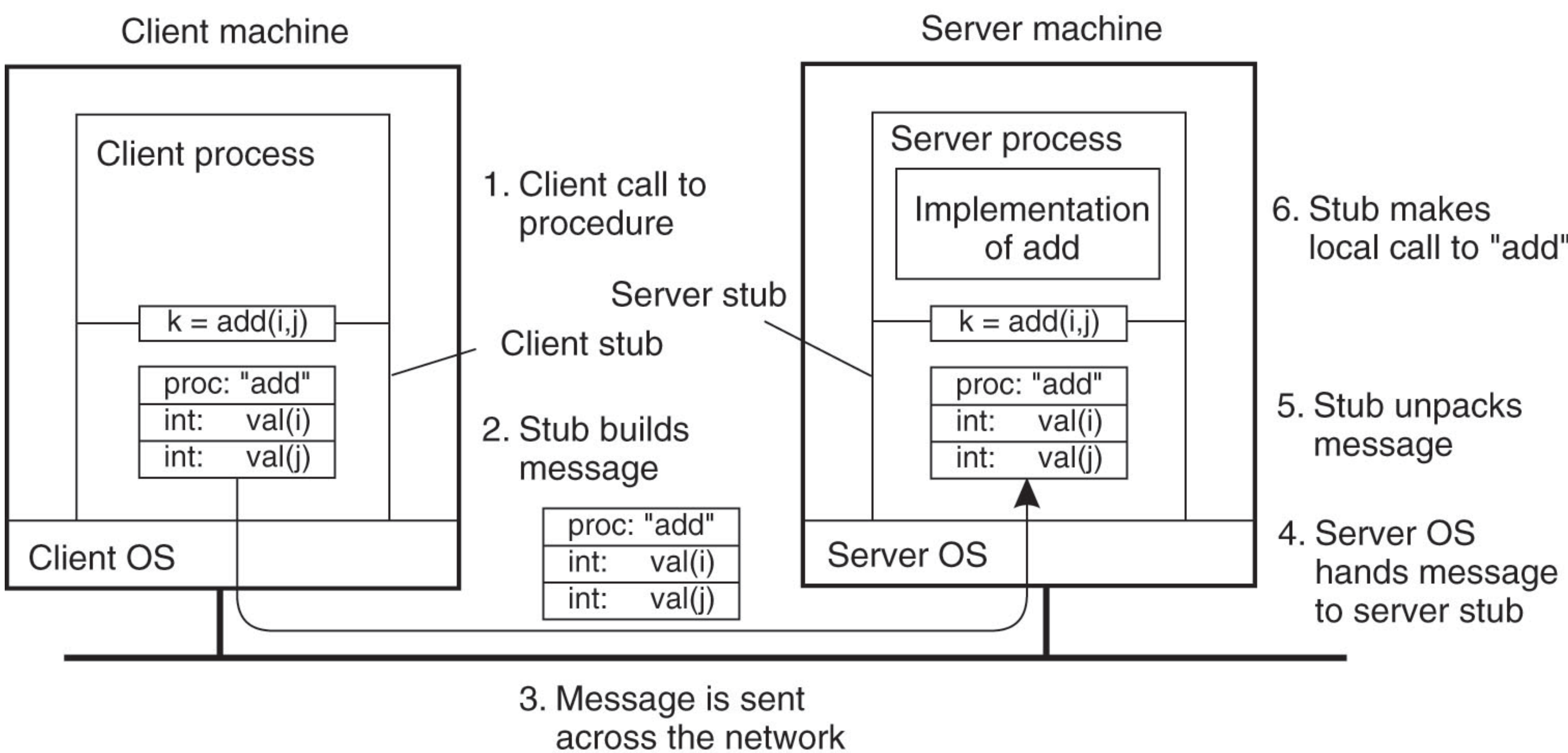




# RPC mechanism





# Problems with parameters passing



- Copy-by-value
  - Different value representation
- Copy-by-reference
  - Distributed memory
  - Copy the array into the message and send it → call-by-copy/restore

# Passing Value Parameters

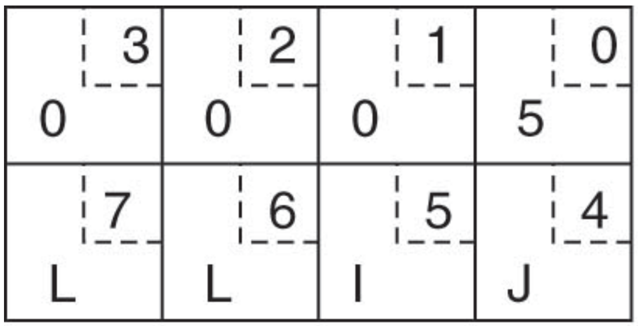


- Work well when the end-systems are uniform
- Problems:
  - Different of representation for numbers, characters, and other data items

# Issue: Different character format

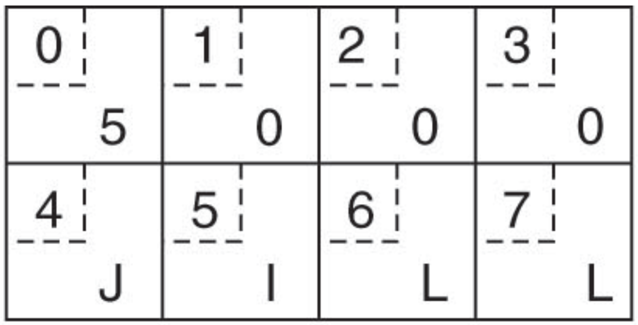


**Intel Pentium** (little endian)

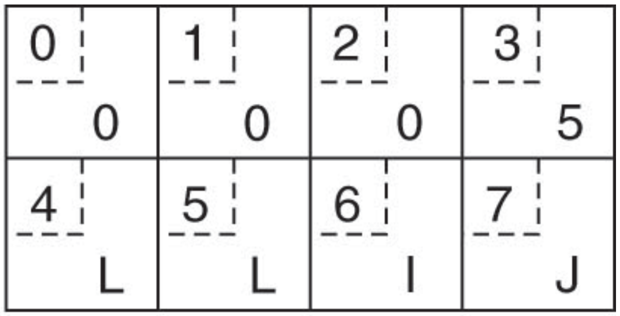



(a)

**SPARC** (big endian)




(b)




(c)

# Passing Reference Parameters

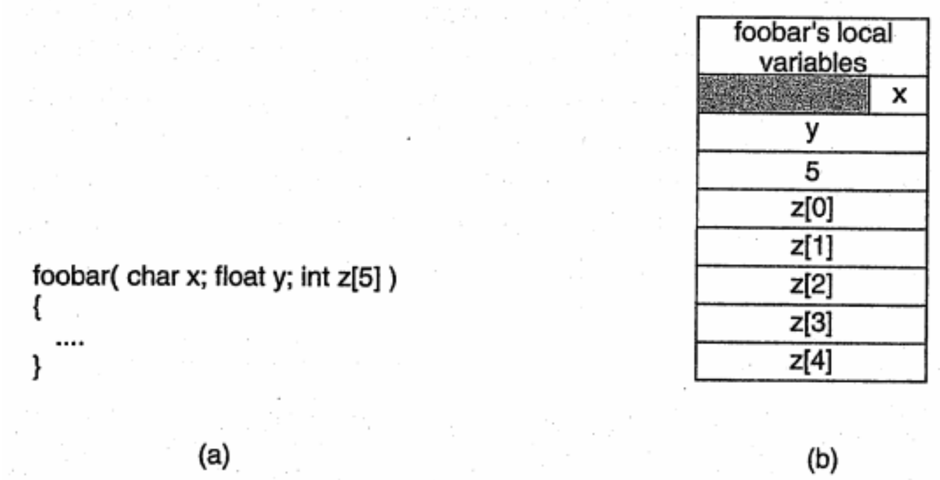


- Issue: a pointer is meaningful only within the address space of the process in which it is being used.
- Solutions:
  - Forbid pointers and reference parameters → undesirable
  - Copy/Restore
    - Issue: costly (bandwidth, store copies)
- Unfeasible for structured data

# Parameter specification



- The caller and the callee agree on the format of the messages they exchange.
- Agreements:
  - Message format
  - Representation of simple data structures (integers, characters, Booleans, etc.)
  - Method for exchanging messages.
  - Client-stub and server-stub need to be implemented.



# Example: CORBA specification



| *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 | 1984 | *unsigned long* |

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

# XML



```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

# IDL



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

# Sun specification



```
/*
 * date.x  Specification of the remote date and time server
 */
/*
 * Define two procedures
 *      bin_date_1() returns the binary date and time (no arguments)
 *      str_date_1() takes a binary time and returns a string
 *
 */
program DATE_PROG {
    version DATE_VERS {
        long BIN_DATE(void) = 1;    /* procedure number = 1 */
        string STR_DATE(long) = 2;  /* procedure number = 2 */
    } = 1;                          /* version number = 1 */
} = 0x31234567;                     /* program number = 0x31234567 */
```

# Openness of RPC



- Client and Server are installed by different providers.
- Common interface between client and server
  - Programming language independence
  - Full description and neutral
  - Using IDL

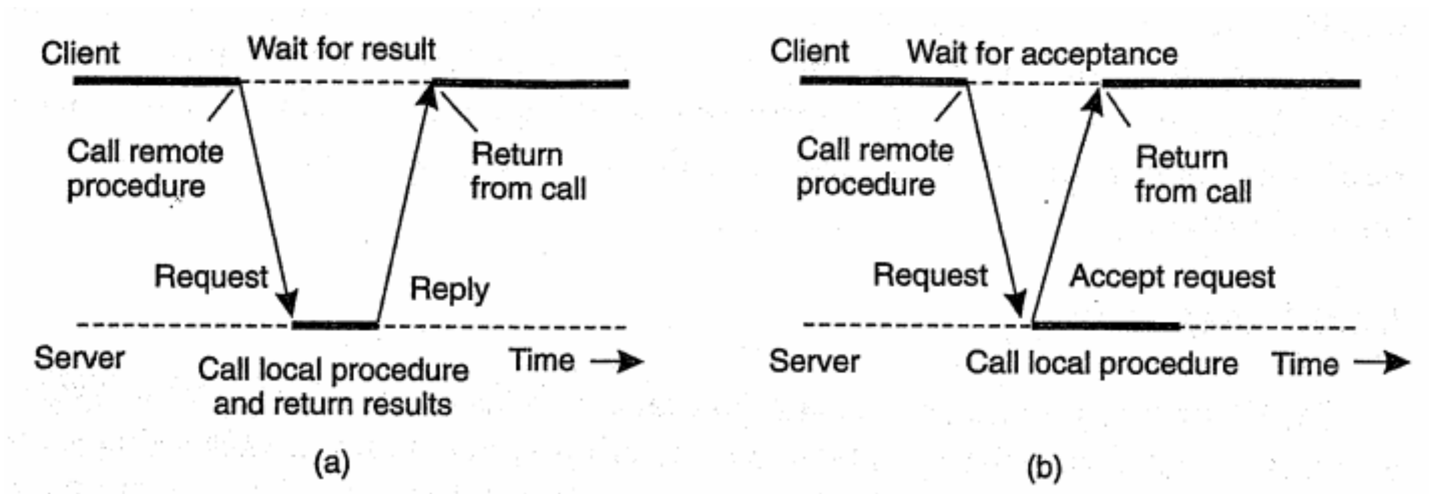
# Asynchronous RPC

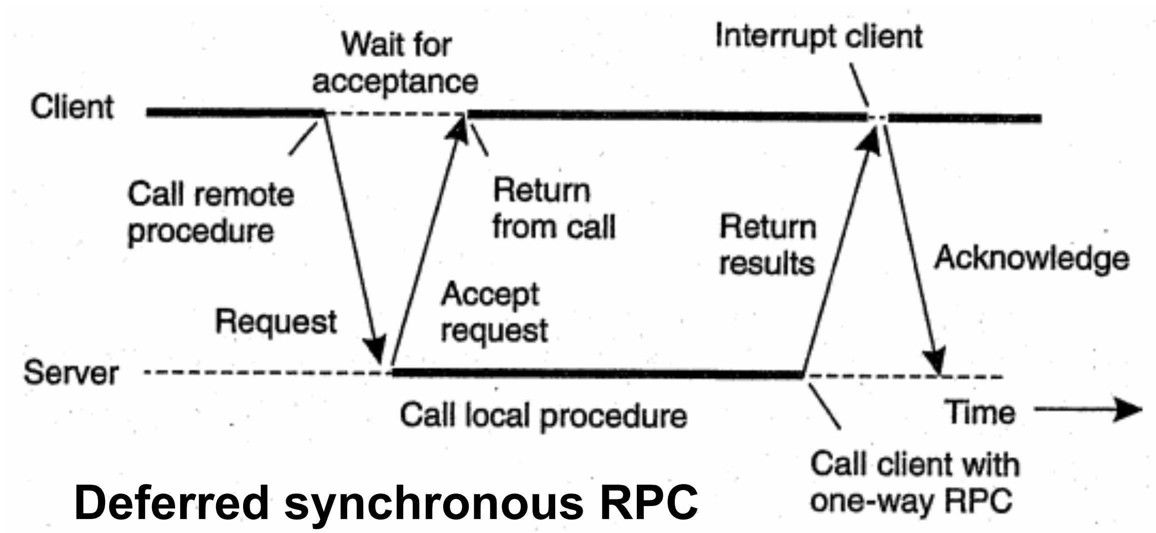


- Sometimes there is no result to return
- After requesting the remote procedure, the caller continue to do useful work without beeing blocked.

# Asynchronous RPC



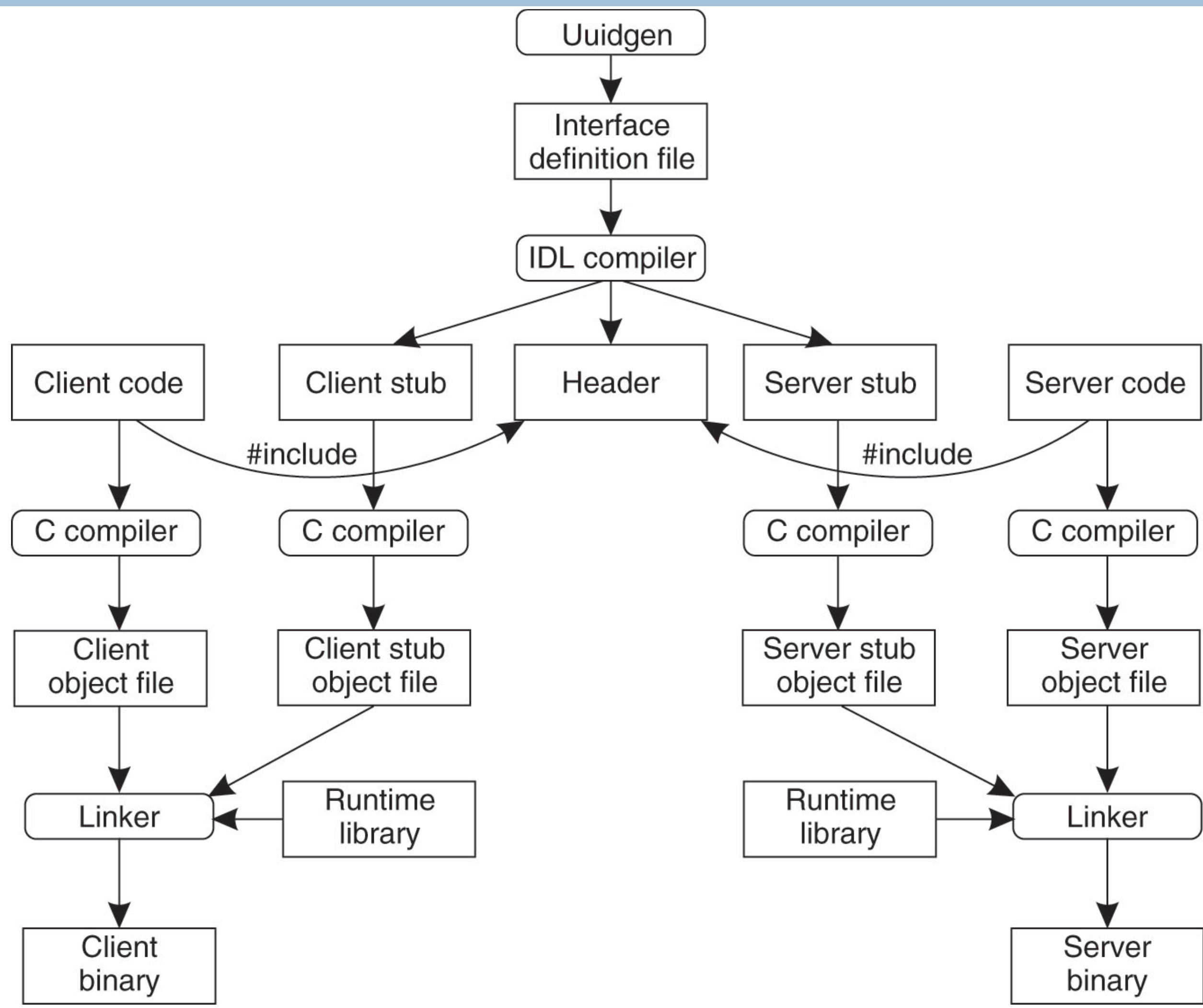



**Deferred synchronous RPC**

# Implementing RPC in using DCE-RPC





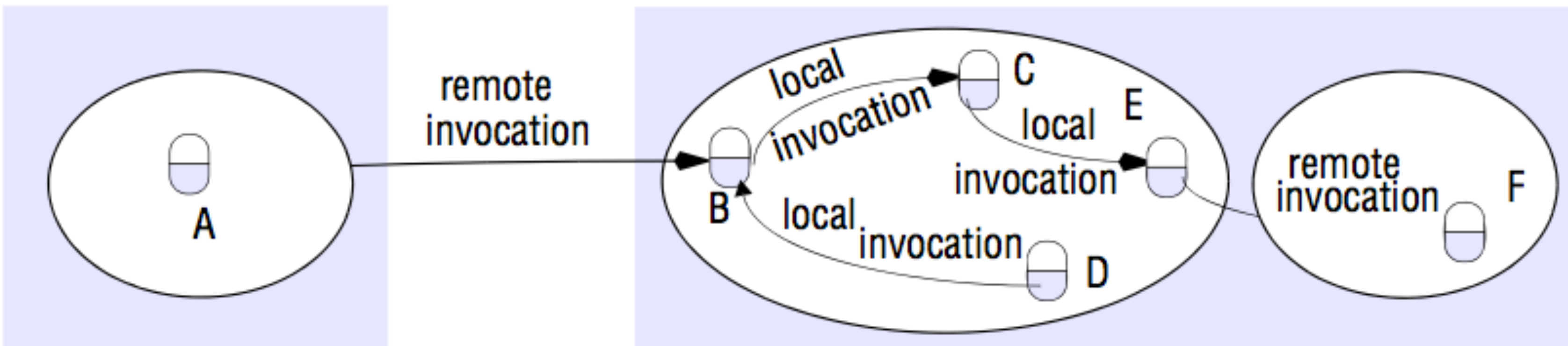
# 2.4. RMI (Remote Method Invocation)



- RMI vs. RPC
  - Common points:
    - Support programming with interface
    - Based on request-reply protocol
    - Transparency
  - Different point:
    - Benefits of OOP

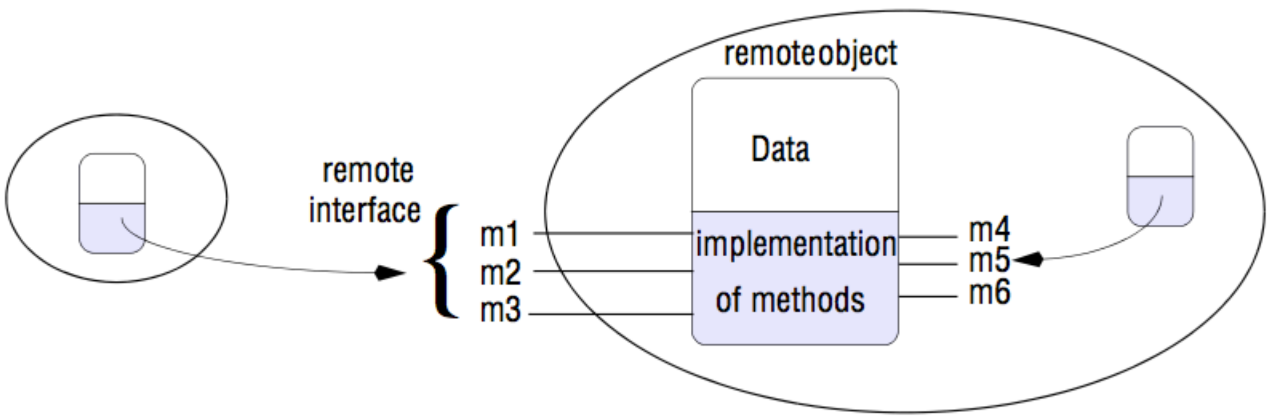
# Distributed objects model





# Remote object and Remote interface





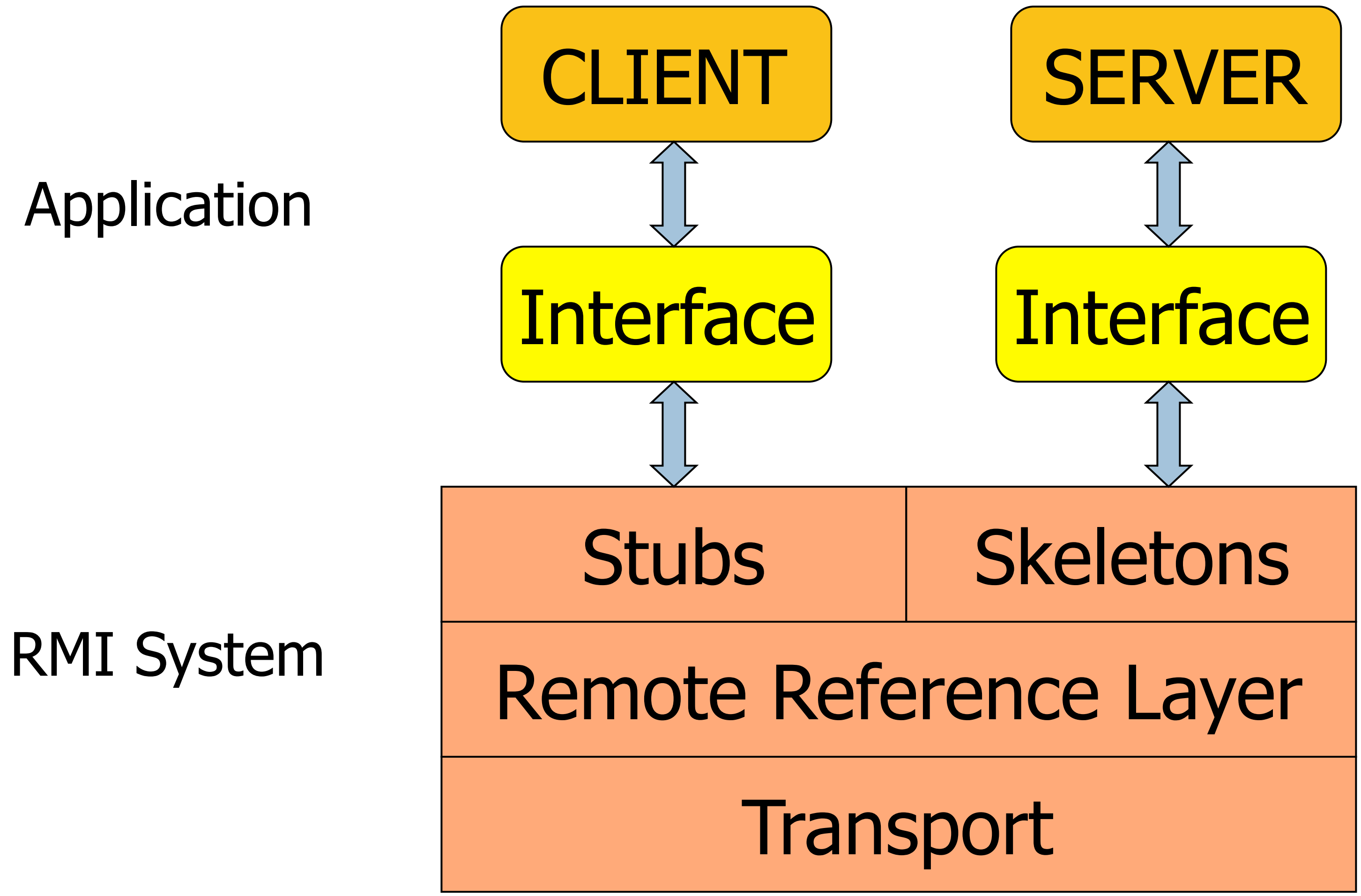
# Characteristics



- Benefits
  - Simplicity
  - Transparency
  - Reliability
  - Security (supported by Java)
- Drawbacks:
  - Only support java

# RMI Architecture





# 3. Message-oriented communication

3.1. Message-oriented transient communication

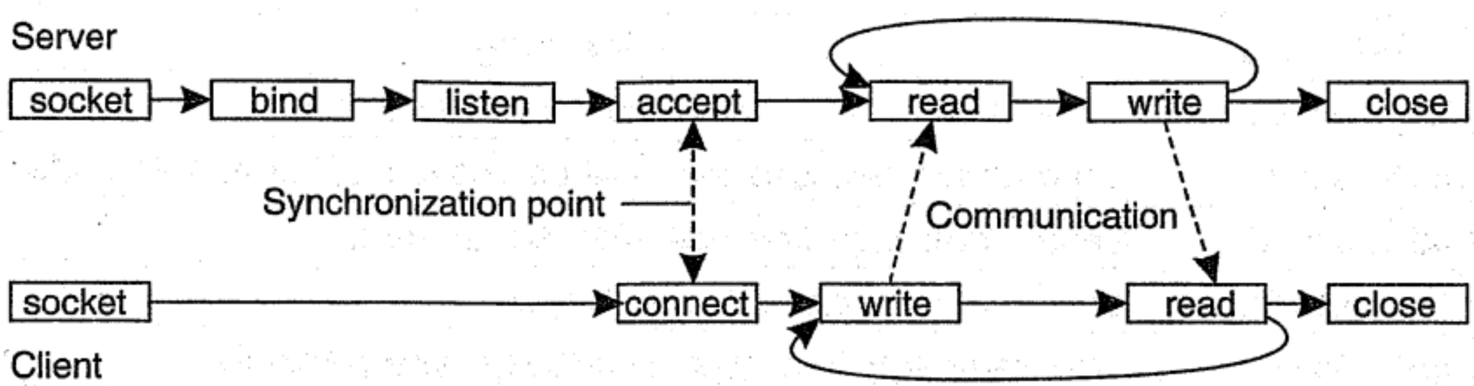
3.2. Message-oriented persistent communication

# 3.1. Message-oriented transient communication

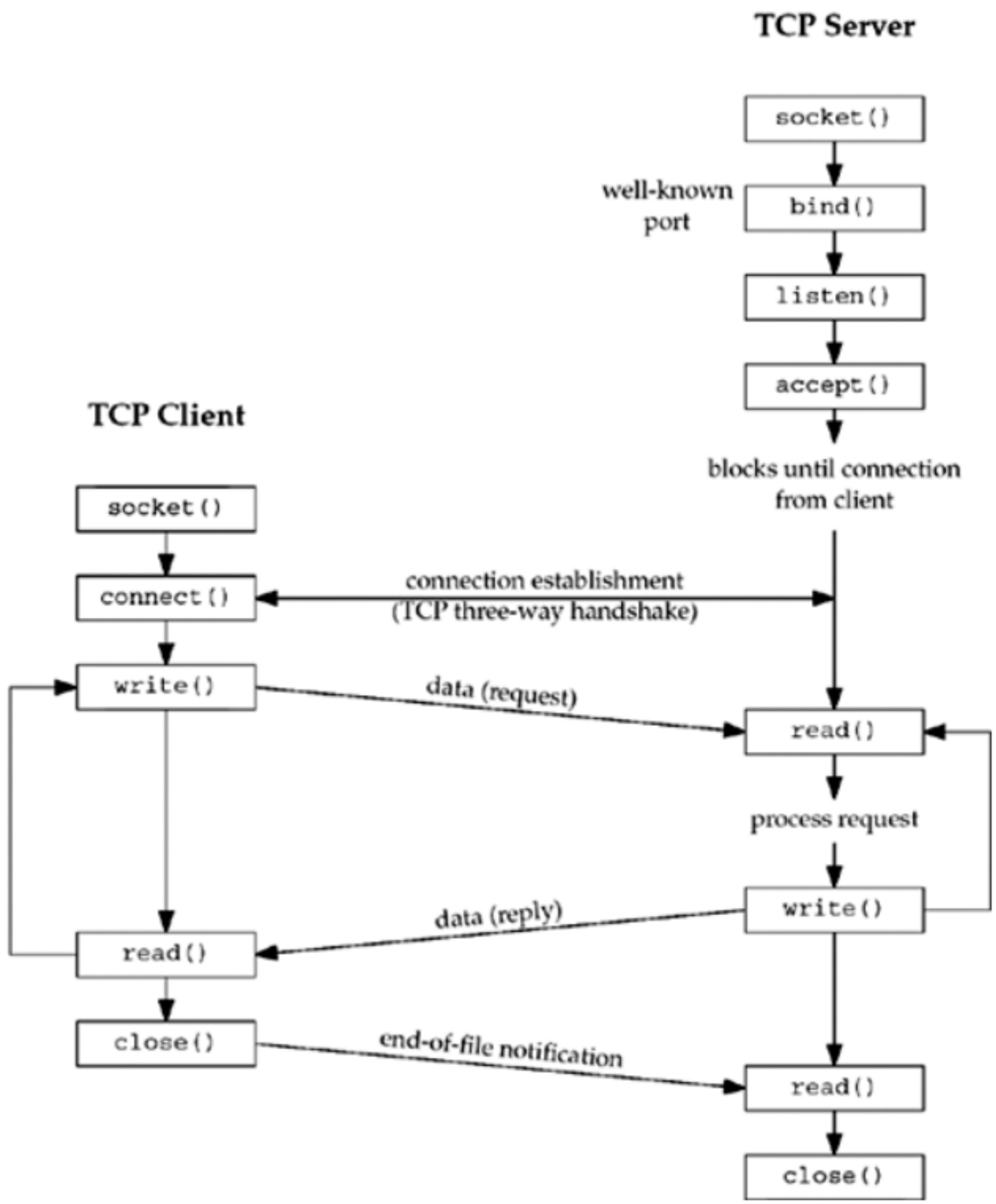


- Berkeley Sockets

| Primitive | Meaning |
|---|---|
| Socket | Create a new communication end point |
| Bind | Attach a local address to a socket |
| Listen | Announce willingness to accept connections |
| Accept | Block caller until a connection request arrives |
| Connect | Actively attempt to establish a connection |
| Send | Send some data over the connection |
| Receive | Receive some data over the connection |
| Close | Release the connection |



# Introduction



# *socket* function

- To perform network I/O, the first thing a process must do is call the *socket* function

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

- Returns: non-negative descriptor if OK, -1 on error

| *family* | Description |
|---|---|
| AF_INET | IPv4 protocols |
| AF_INET6 | IPv6 protocols |
| AF_LOCAL | Unix domain protocols (Chapter 15) |
| AF_ROUTE | Routing sockets (Chapter 18) |
| AF_KEY | Key socket (Chapter 19) |

**family**

| *type* | Description |
|---|---|
| SOCK_STREAM | stream socket |
| SOCK_DGRAM | datagram socket |
| SOCK_SEQPACKET | sequenced packet socket |
| SOCK_RAW | raw socket |

**socket**

| *Protocol* | Description |
|---|---|
| IPPROTO_TCP | TCP transport protocol |
| IPPROTO_UDP | UDP transport protocol |
| IPPROTO_SCTP | SCTP transport protocol |

**protocol**

# *connect* Function

- The connect function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr,
   socklen_t addrlen);
```

- Returns: 0 if OK, -1 on error
- *sockfd* is a socket descriptor returned by the *socket* function
- The second and third arguments are a pointer to a socket address structure and its size.
- The client does not have to call *bind* before calling *connect*: the kernel will choose both an ephemeral port and the source IP address if necessary.

# *connect* Function (2)

- Problems with *connect* function:
  1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned. (If no response is received after a total of 75 seconds, the error is returned).
  2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (i.e., the server process is probably not running). Error: ECONNREFSED.
  3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a soft error. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH.

# *bind* Function

- The bind function assigns a local protocol address to a socket.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr,
  socklen_t addrlen);
```

- Returns: 0 if OK,-1 on error
- Example:

```
struct sockaddr_in address;

/* type of socket created in socket() */
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = INADDR_ANY;
/* 7000 is the port to use for connections */
  address.sin_port = htons(7000);
/* bind the socket to the port specified above */
```

# *listen* Function

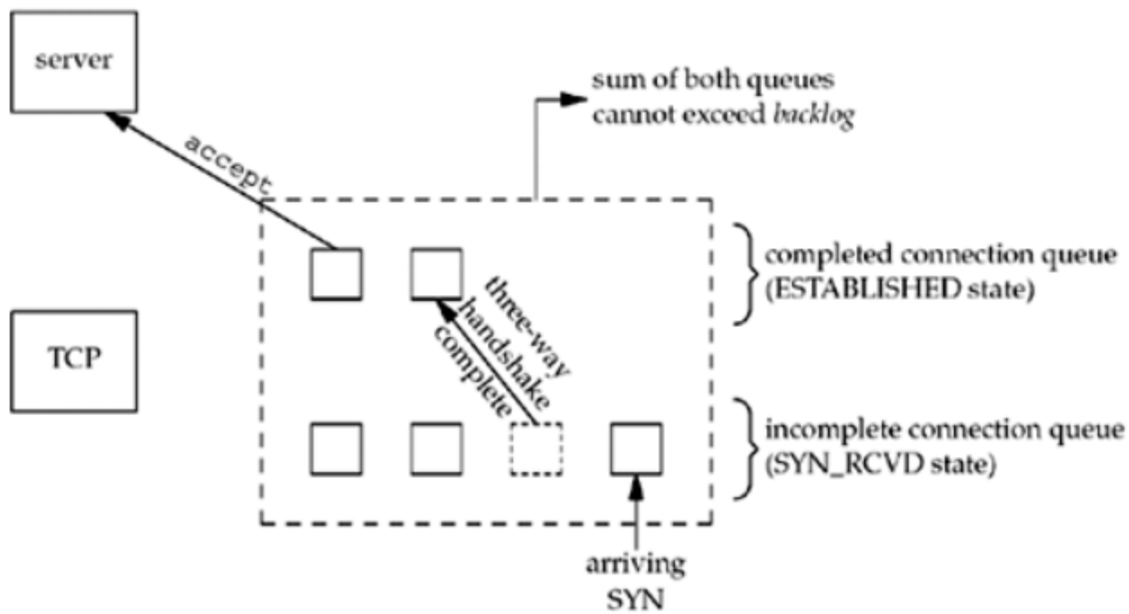
- The listen function is called only by a TCP server.
- When a socket is created by the *socket* function, it is assumed to be an active socket, that is, a client socket that will issue a *connect*.
- The *listen* function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.
- Move the socket from the CLOSED state to the LISTEN state.

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

- Returns: 0 if OK, -1 on error

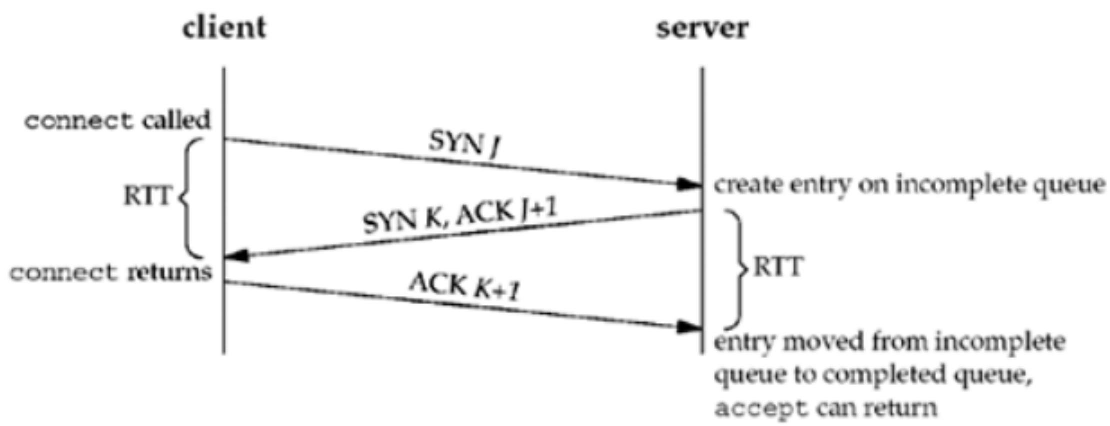
# *listen* Function (2)

- The second argument (*backlog*) to this function specifies the maximum number of connections the kernel should queue for this socket.




**The two queues maintained by TCP for a listening socket**

# *listen* Function (3)




**TCP three-way handshake and the two queues for a listening socket.**

# *accept* Function

- *accept* is called by a TCP server to return the next completed connection from the front of the completed connection queue.
- If the completed connection queue is empty, the process is put to sleep.

```
#include <sys/socket.h>

int accept (int sockfd, struct sockaddr *cliaddr, socklen_t
   *addrlen);
```

- Returns: non-negative descriptor if OK, -1 on error
- The *cliaddr* and addrlen arguments are used to return the protocol address of the connected peer process (the client).
- *addrlen* is a value-result argument

# *accept* Function

- Example

```
int addrlen;
struct sockaddr_in address;

  addrlen = sizeof(struct sockaddr_in);
  new_socket = accept(socket_desc, (struct sockaddr *)&address, &addrlen);
  if (new_socket<0)
    perror("Accept connection");
```

# *fork* and *exec* Functions

```
#include <unistd.h>
pid_t fork(void);
```

- Returns: 0 in child, process ID of child in parent, -1 on error
- *fork* function (including the variants of it provided by some systems) is the only way in Unix to create a new process.
- It is called once but it returns twice.
- It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0.
- The reason fork returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling *getppid*.

# Example

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    printf("--beginning of program\n");

    int counter = 0;
    pid_t pid = fork();

    if (pid == 0)
    {
        // child process
        int i = 0;
        for (; i < 5; ++i)
        {
            printf("child process: counter=%d\n", ++counter);
        }
    }
    else if (pid > 0)
    {
        // parent process
        int j = 0;
        for (; j < 5; ++j)
        {
            printf("parent process: counter=%d\n", ++counter);
        }
    }
    else
    {
        // fork failed
        printf("fork() failed!\n");
        return 1;
    }

    printf("--end of program--\n");

    return 0;
}
```

- 2 typical uses of fork:
  - A process makes a copy of itself so that one copy can handle one operation while the other copy does another task.
  - A process wants to execute another program.

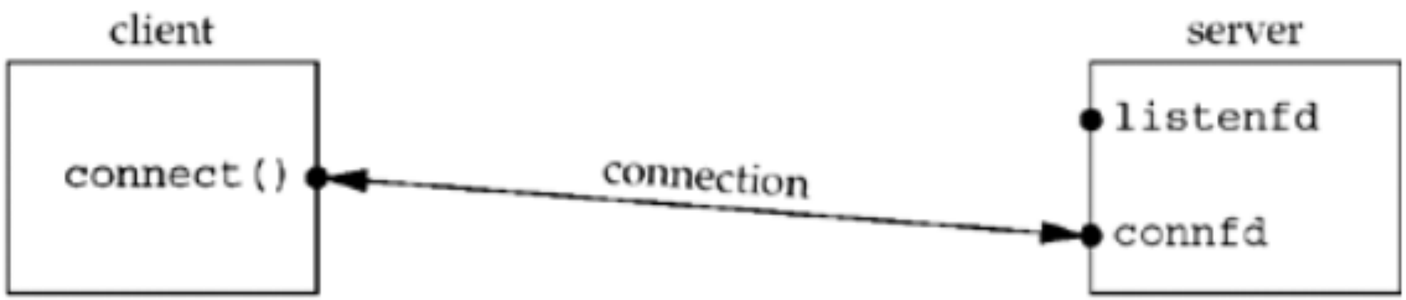
# Concurrent Servers

- *fork* a child process to handle each client

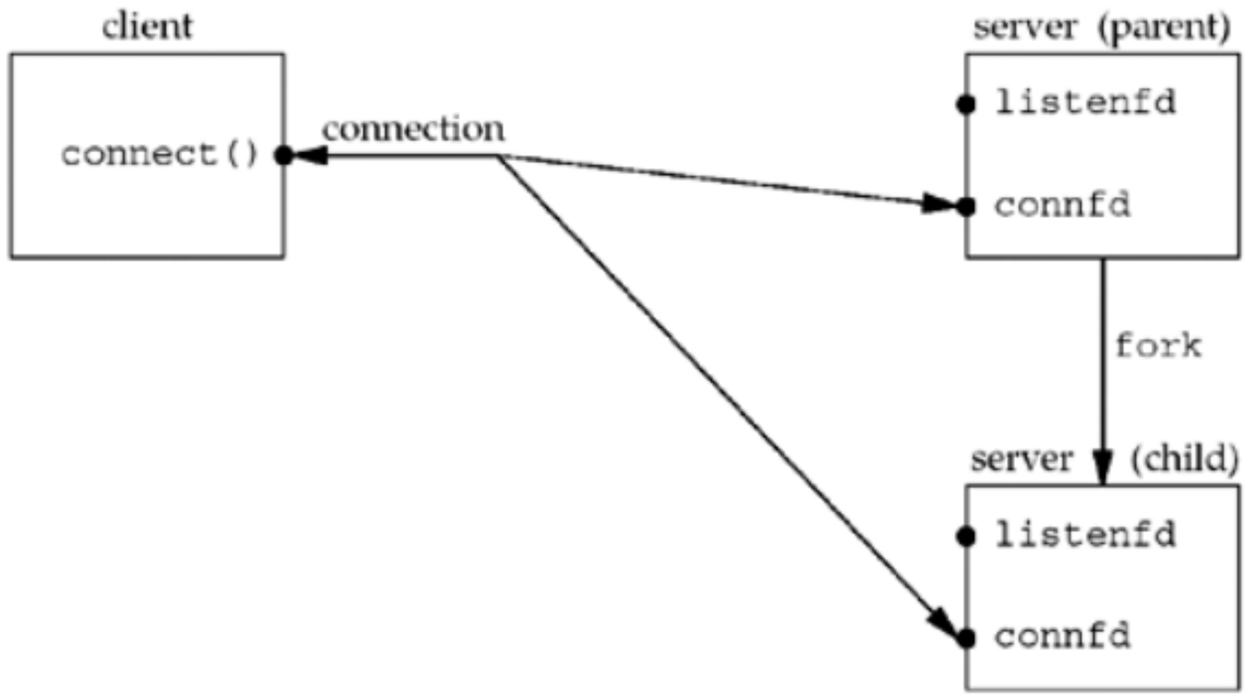
```
pid_t pid;
int   listenfd,  connfd;

listenfd = Socket( ... );

    /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
    connfd = Accept (listenfd, ... );    /* probably blocks */

    if( (pid = Fork()) == 0) {
       Close(listenfd);    /* child closes listening socket */
       doit(connfd);       /* process the request */
       Close(connfd);      /* done with this client */
       exit(0);            /* child terminates */
    }

    Close(connfd);         /* parent closes connected socket */
}
```

# Status of client/server before call to *accept* returns.

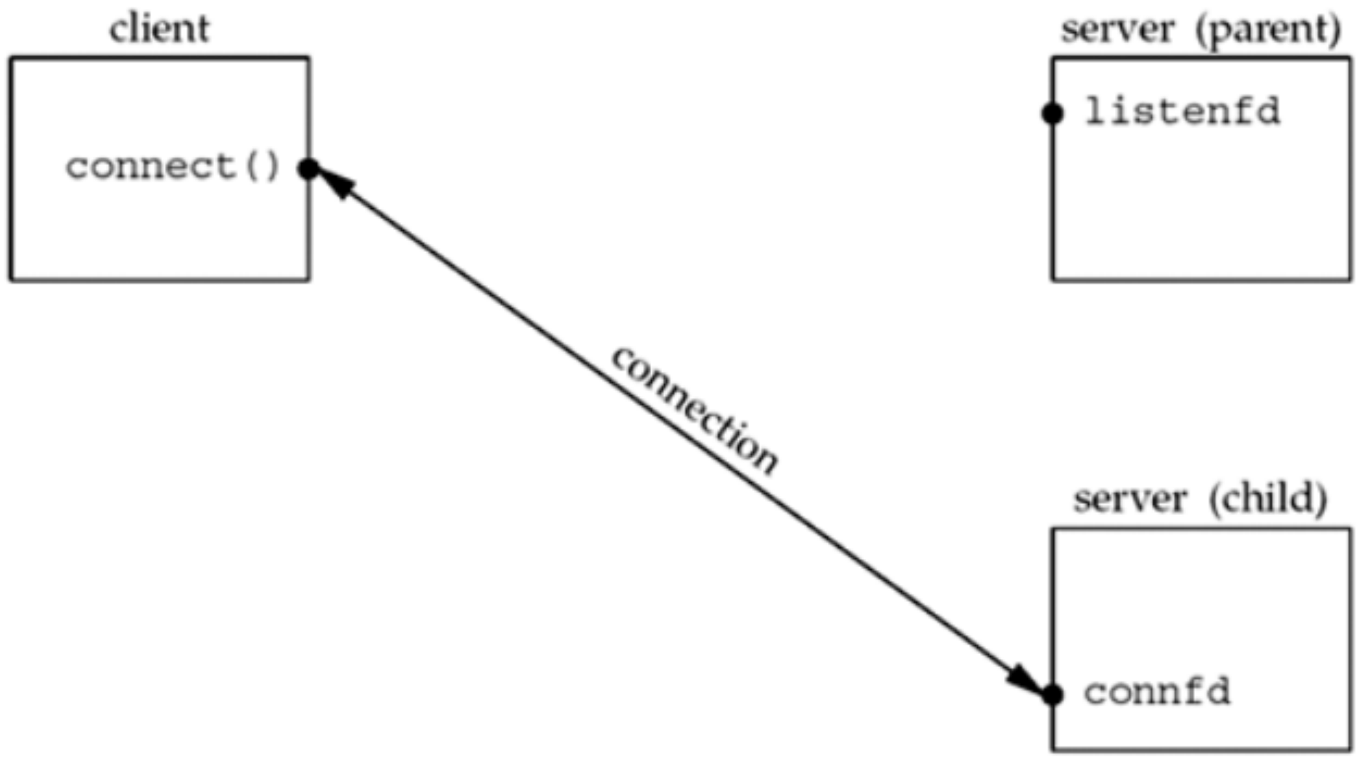
# Status of client/server after return from *accept*.



# Status of client/server after fork returns.



# Status of client/server after parent and child close appropriate sockets.



# *close* Function

- The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>
int close (int sockfd);
```

- Returns: 0 if OK, -1 on error
- If the parent doesn’t close the socket, when the child closes the connected socket, its reference count will go from 2 to 1 and it will remain at 1 since the parent never closes the connected socket. This will prevent TCP's connection termination sequence from occurring, and the connection will remain open.

# Message-Passing Interface



| Primitive | Meaning |
|---|---|
| MPI_bsend | Append outgoing message to a local send buffer |
| MPI_send | Send a message and wait until copied to local or remote buffer |
| MPI_ssend | Send a message and wait until receipt starts |
| MPI_sendrecv | Send a message and wait for reply |
| MPI_isend | Pass reference to outgoing message, and continue |
| MPI_issend | Pass reference to outgoing message, and wait until receipt starts |
| MPI_recv | Receive a message; block if there is none |
| MPI_irecv | Check if there is an incoming message, but do not block |

**The socket primitives for TCP/IP**

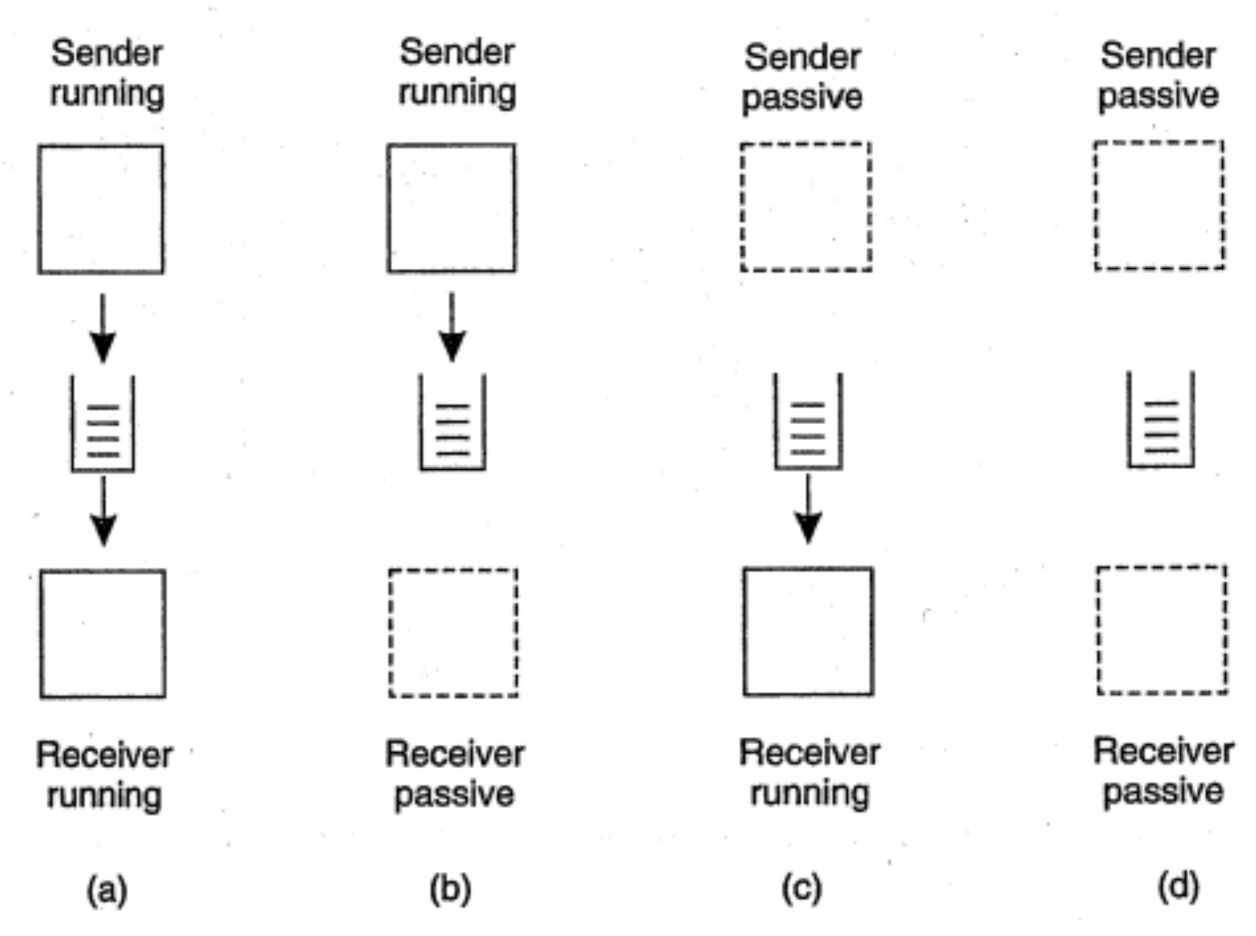
# 3.2. Message-Oriented Persistent Communication



- Very important class of message-oriented middleware services: Message-Queuing Systems, or MOM (Message-Oriented Middleware).
- Message-Queuing Systems provide extensive support for persistent asynchronous communication.
- Offer intermediate-term storage capacity for messages
- Latency tolerance
- Example: Email system

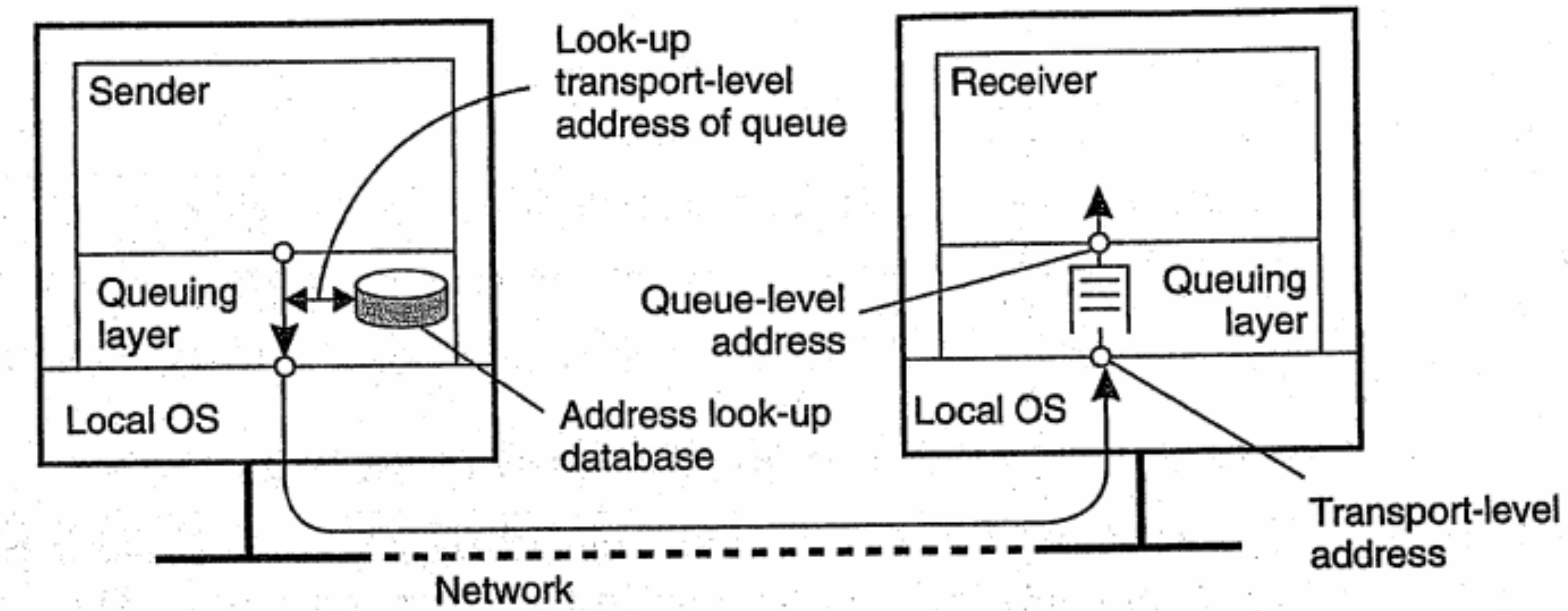
# Message-Queuing System





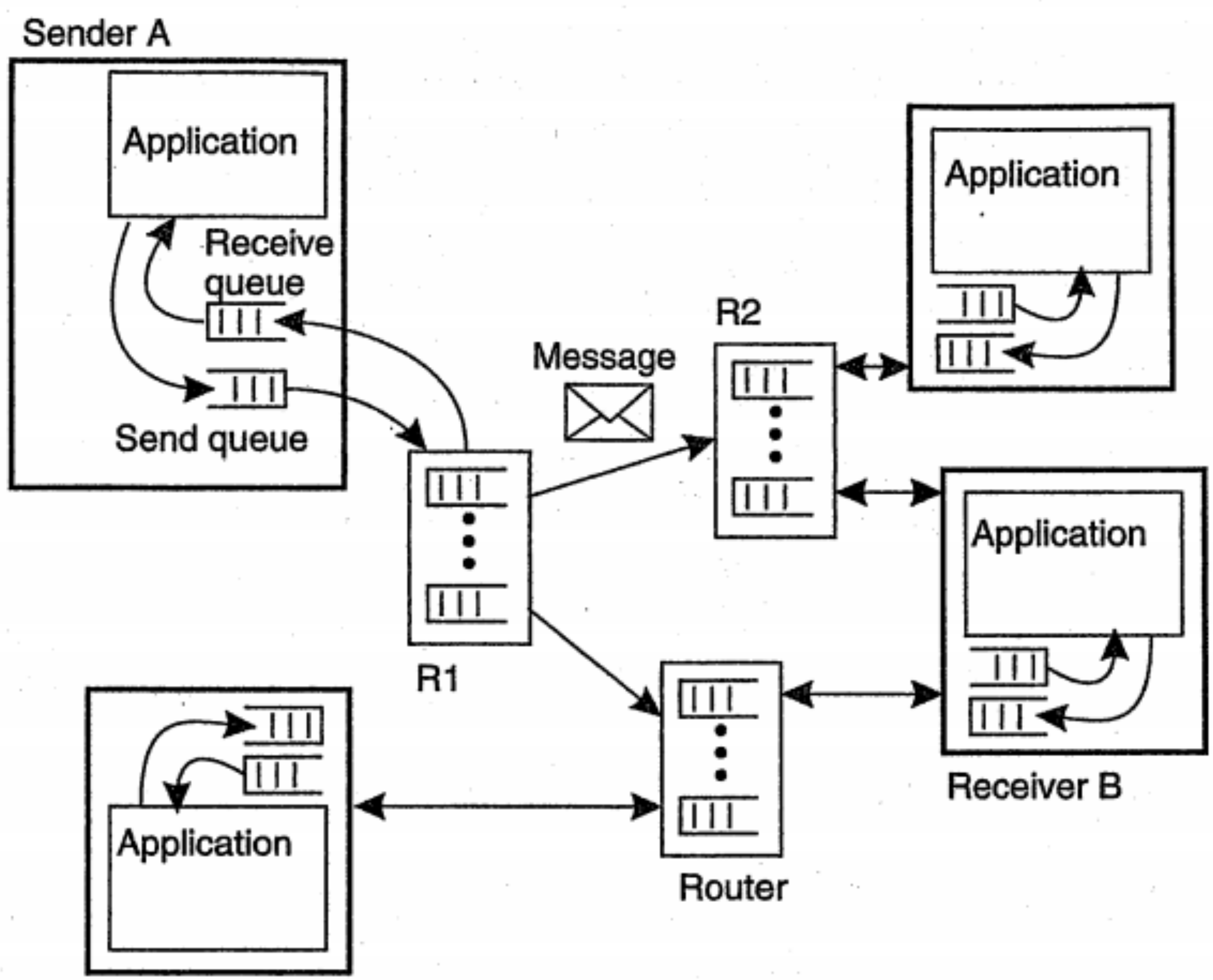
# The relationship between queue-level addressing and network-level addressing





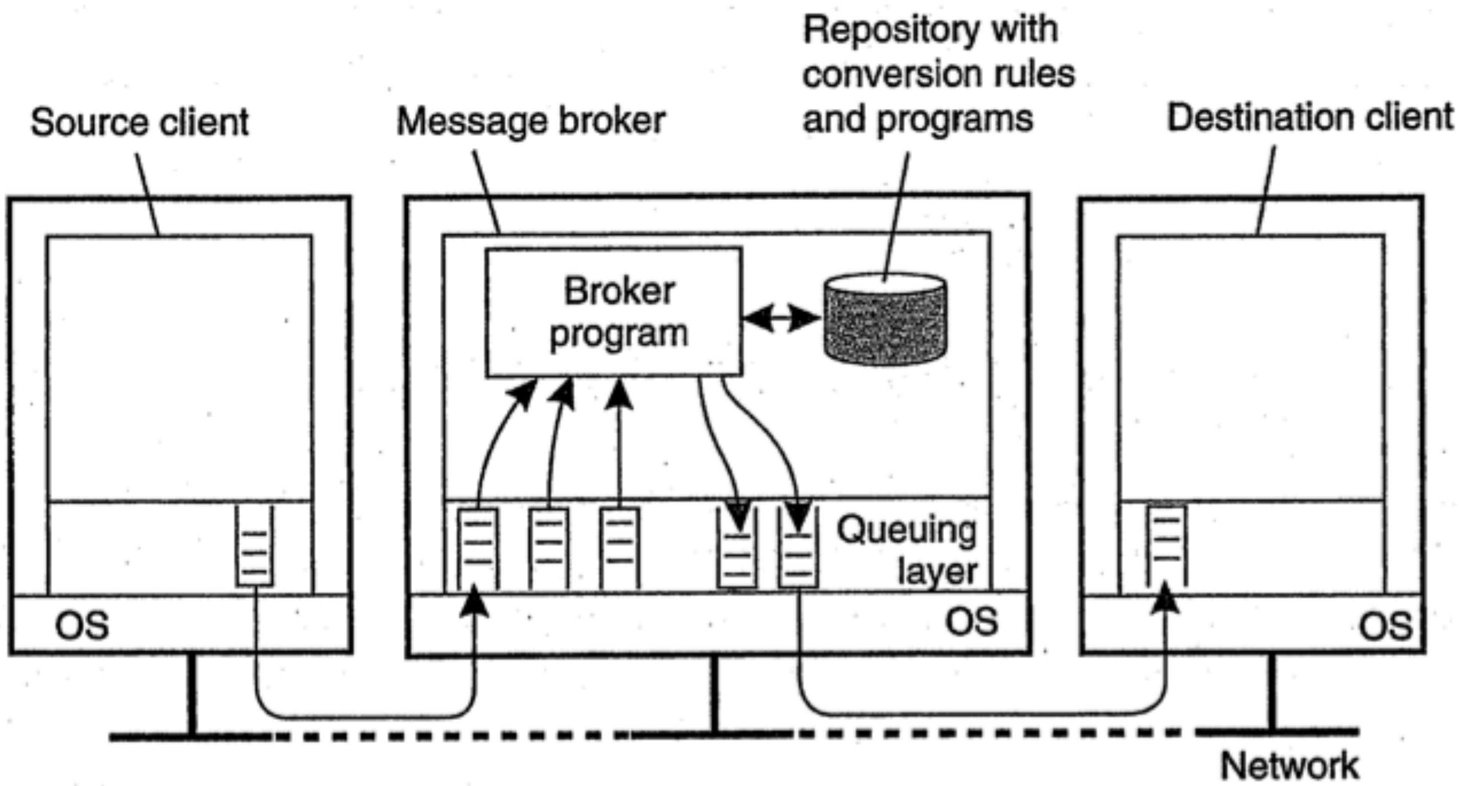
# Routing with Queueing system





# Message Broker





# RabbitMQ

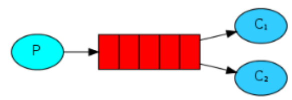
## 1 "Hello World!"

The simplest thing that does *something*


- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C

## 2 Work queues

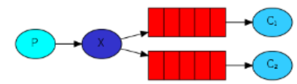
Distributing tasks among workers




- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C

## 3 Publish/Subscribe

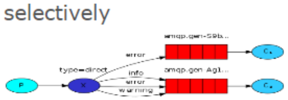
Sending messages to many consumers at once



- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C

## 4 Routing

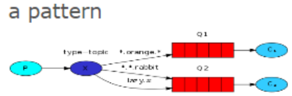
Receiving messages selectively



- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C

## 5 Topics

Receiving messages based on a pattern



- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir
- Objective-C

## 6 RPC

Remote procedure call implementation

- Python
- Java
- Ruby
- PHP
- C#
- Javascript
- Go
- Elixir

# 4. Stream-oriented Communication

4.1. Support for Continuous Media

4.2. Streams and QoS

4.3. Stream synchronization

# 4.1. Support for Continuous Media



- The medium of communication
  - Storage
  - Transmission
  - Representation (screen, etc.)
- Continuous/discrete media

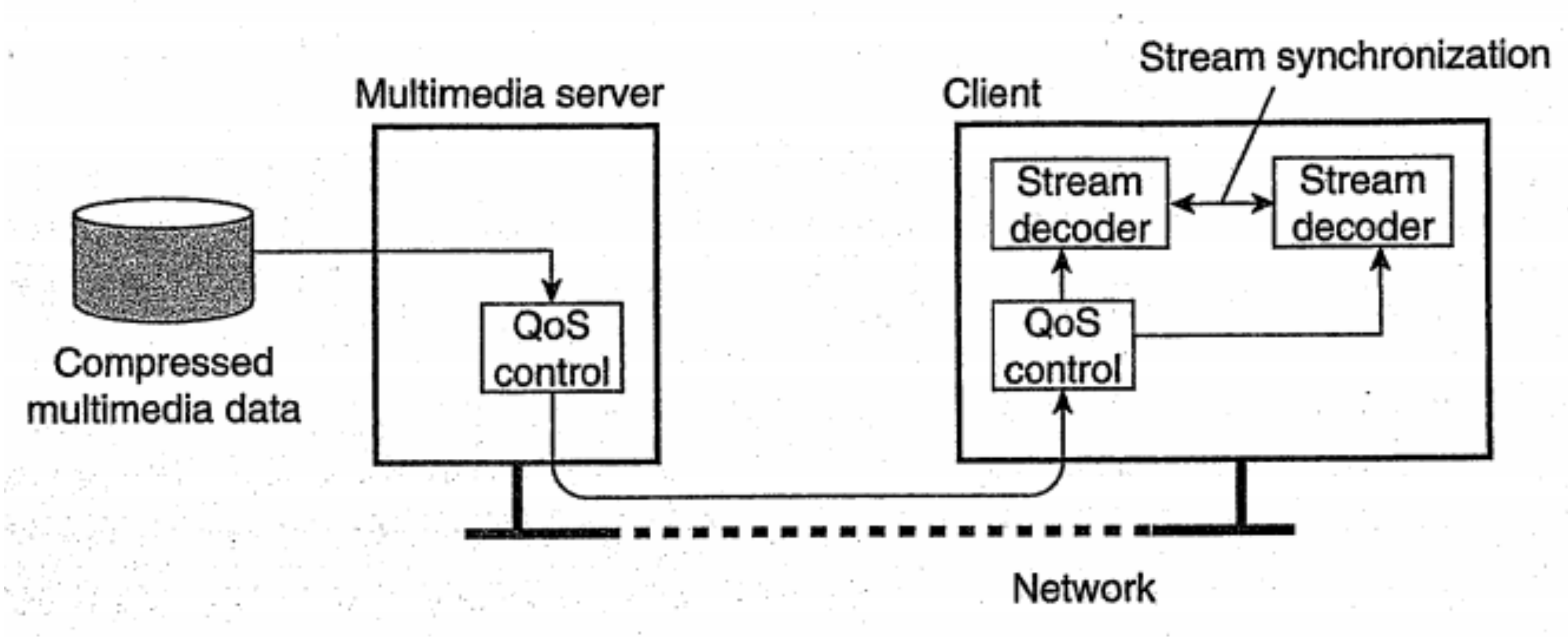
# Data stream



- Sequence of data units
- Can be applied to discrete and continuous media
- Timing aspects
- A simple stream: only a single sequence of data
- A complex stream: several related simple streams
- Issues:
  - Data compression
  - QoS
  - Synchronization

# Data stream (cont.)






**A general architecture for streaming stored multimedia data over a network**

# 4.2. Streams and QoS

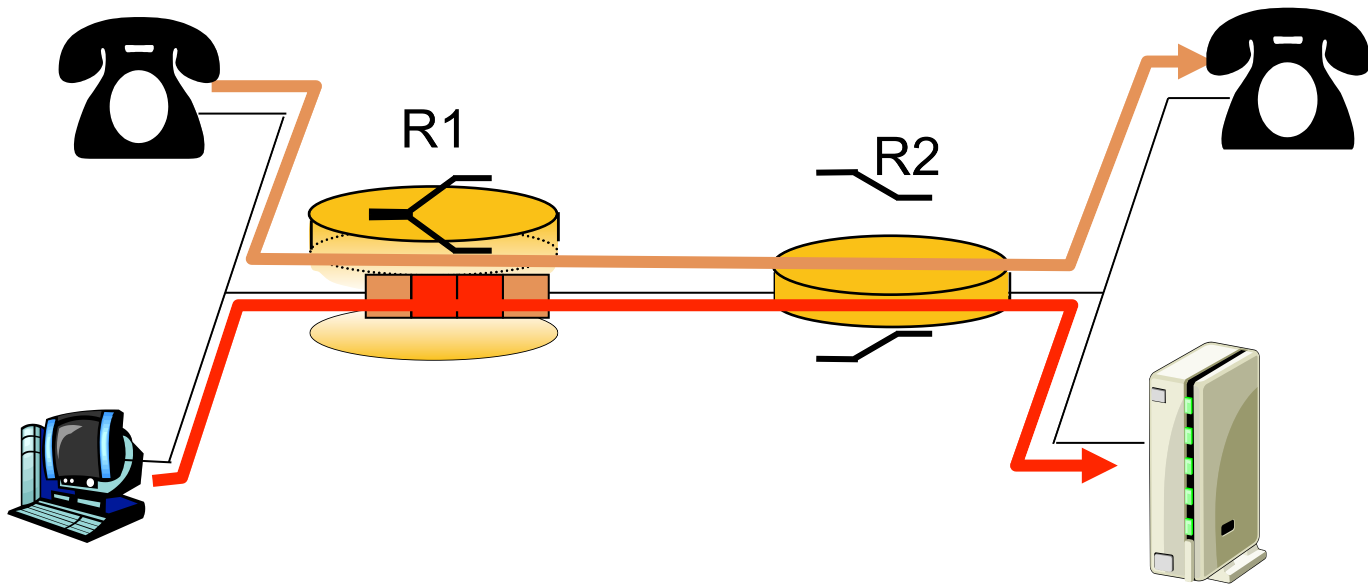


- Quality of Service (QoS):
  - bit-rate,
  - delay
  - e2e delay
  - jitter
  - round-trip delay
- Based on IP layer
  - Simple in using best-effort policy

# Enforcing QoS



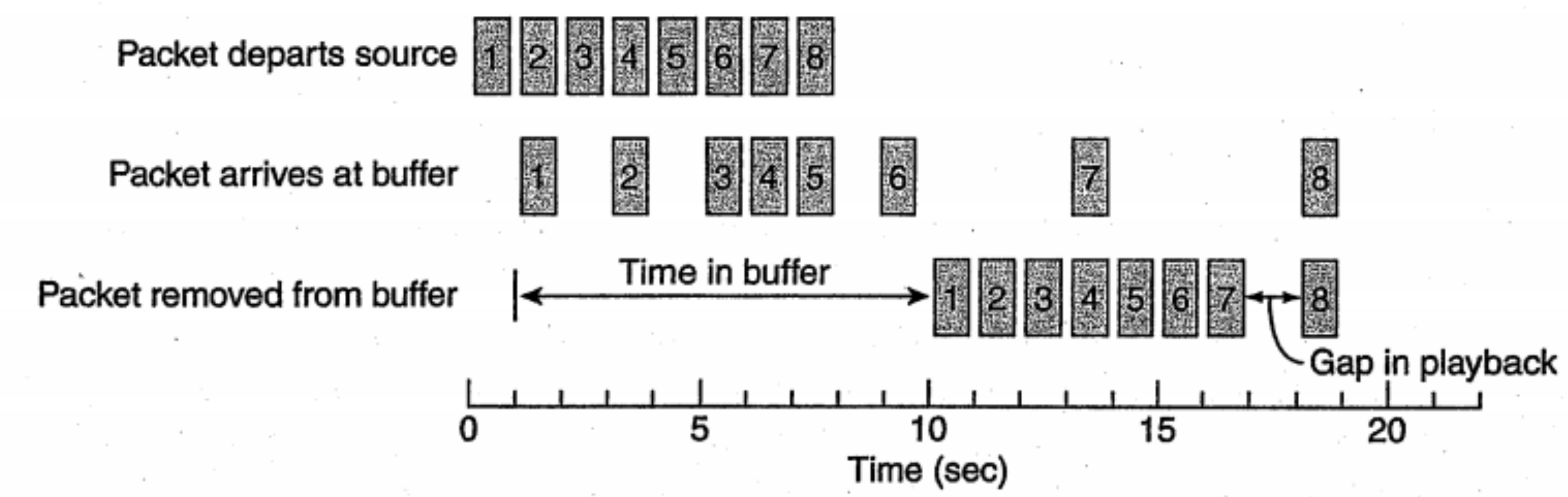
- Differentiated services



# Enforcing QoS (cont.)



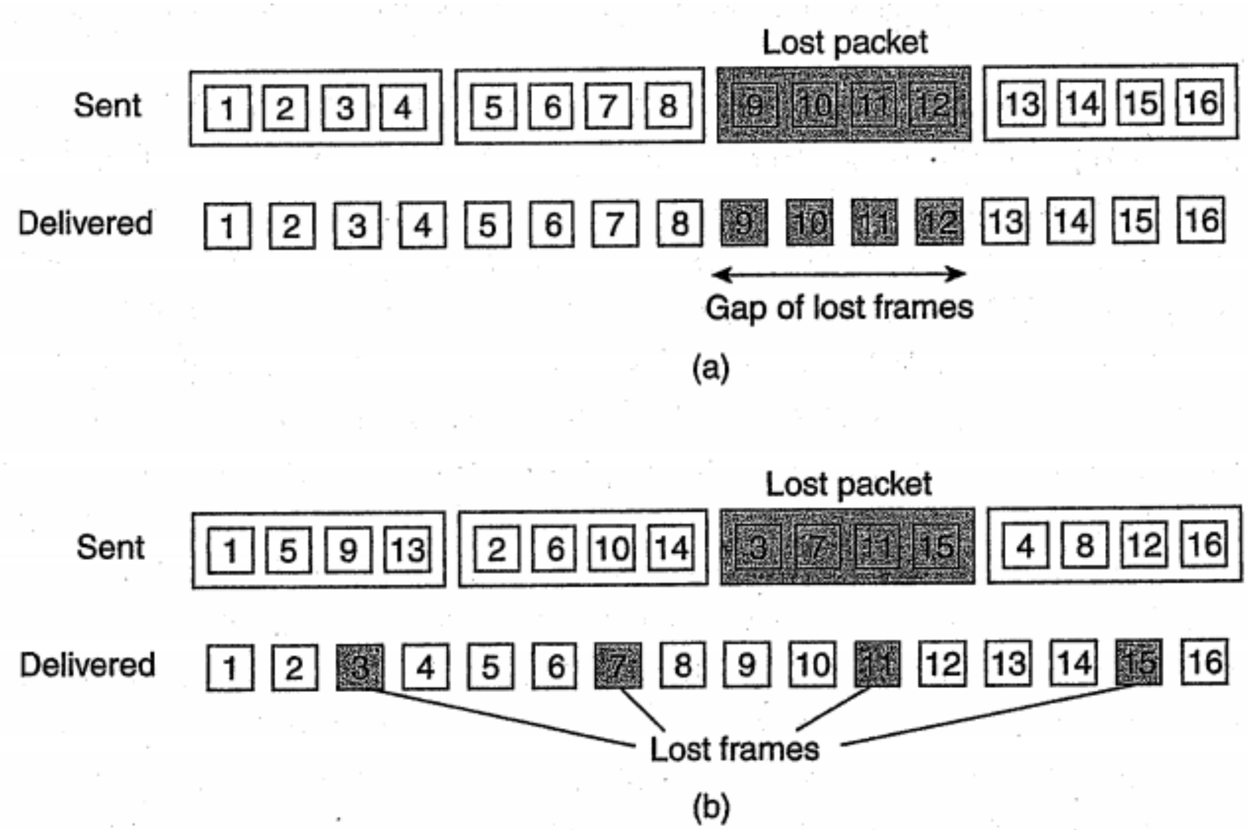
- Using a buffer to reduce jitter



# Enforcing QoS (cont.)

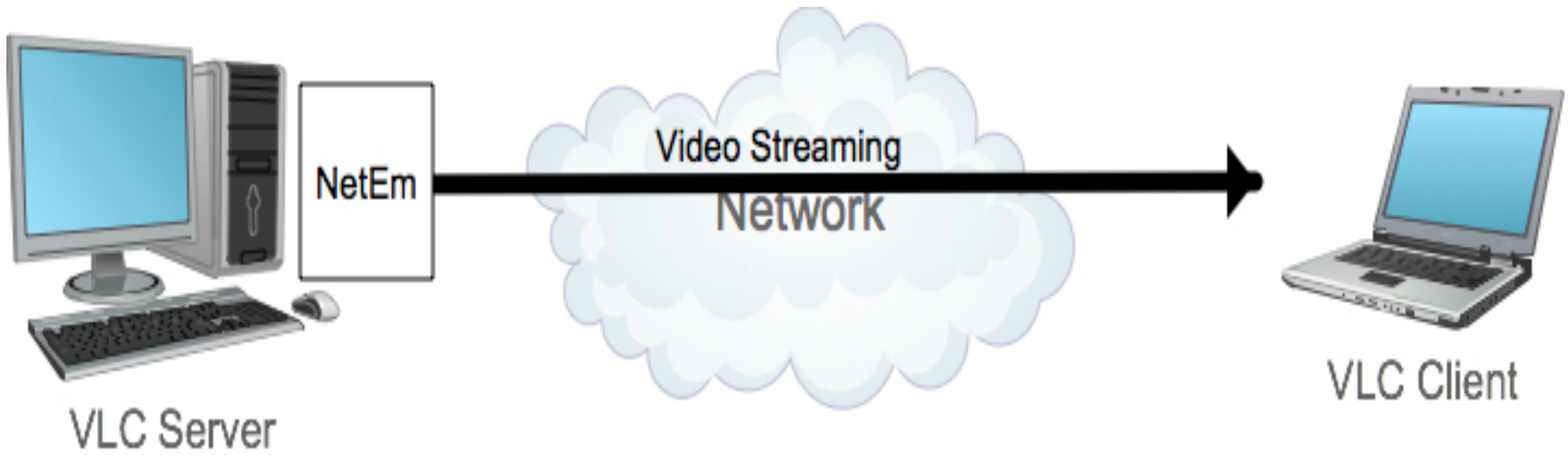


- Forward error correction (FEC)
  - Interleaved transmission



# Labwork





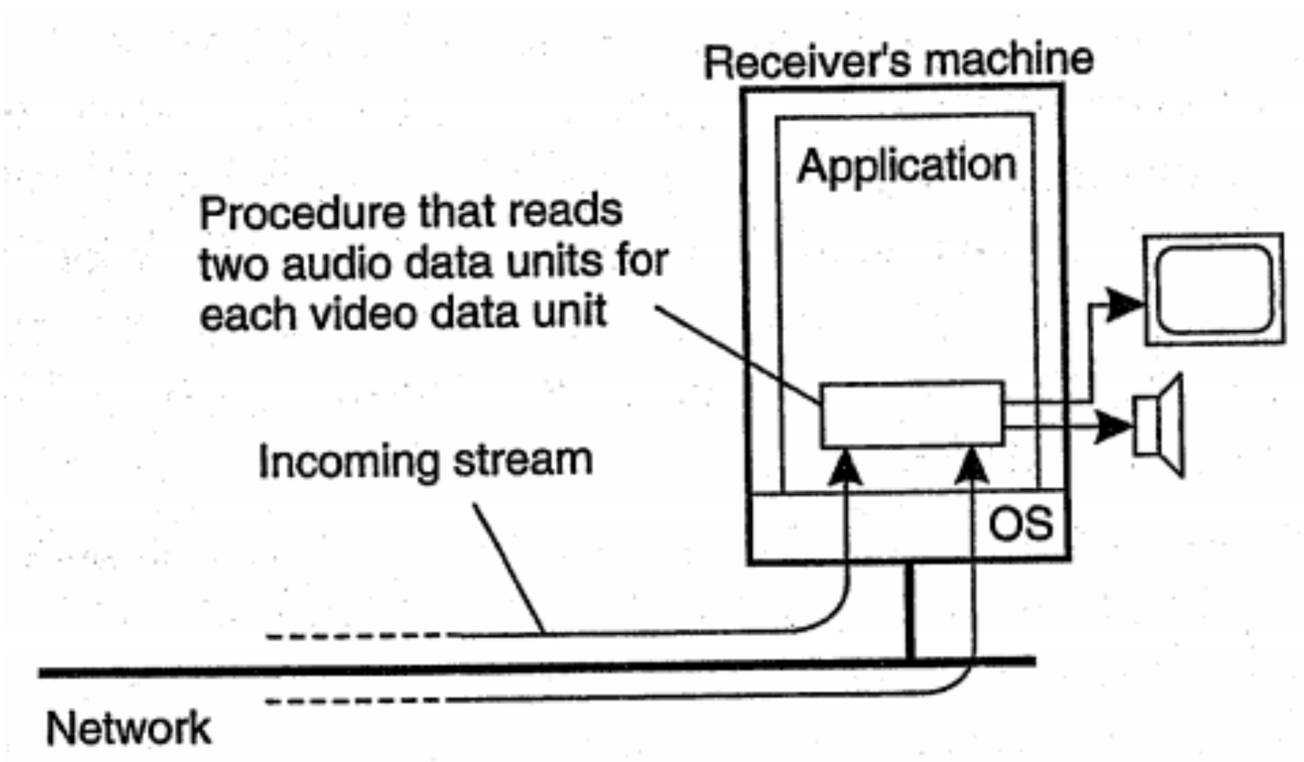
# 4.3. Stream Synchronization



- Needs of stream synchronization
- 2 types:
  - Synchronize *discrete data stream* and *continuous data stream*.
  - Synchronize 2 *continuous data streams.*

# Explicit synchronization on the level data units





# Synchronization as supported by high-level interfaces