HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

DISTRIBUTED SYSTEMS

CHAPTER 2: PROCESSES AND COMMUNICATION IN DISTRIBUTED SYSTEMS

Dr. Trần Hải Anh

Outline

1. Process and Thread
2. Overview of Communication in DS
3. RPC
4. Message-Oriented Communication
5. Stream-Oriented Communication

1. Process and Thread

1.1. Introduction

1.2. Threads in centralized systems

1.3. Threads in distributed systems

1.1. Introduction

- Process
  - A program in execution
  - Creating a process:
    - Create a complete independent address space
    - Allocation = initializing memory segments by zeroing a data segment, copying the associated program into a text segment, setup a stack for temporary data
  - Resources
    - Execution environment, memory space, registers, CPU...
    - Virtual processors
    - Virtual memory
  - Concurrency transparency
  - Switching the CPU between processes: Saving the CPU context + modify registers of MMU, ...

Thread

- A thread executes its own piece of code, independently from other threads.
- Process has several threads → multithreaded process
- Threads of a process use the process'context together
- Thread context: CPU context with some other info for thread management.
- Exchanging info by using shared variable (mutex variable)
- Protecting data against inappropriate access by threads within a single process is left to application developers.

1.2. Multi-threading and multi-processing

- Parallel processing benefits with sequential processing
- Multithreaded program vs multi-processes program
  - Programming cost
  - Switching context
  - Blocking system calls

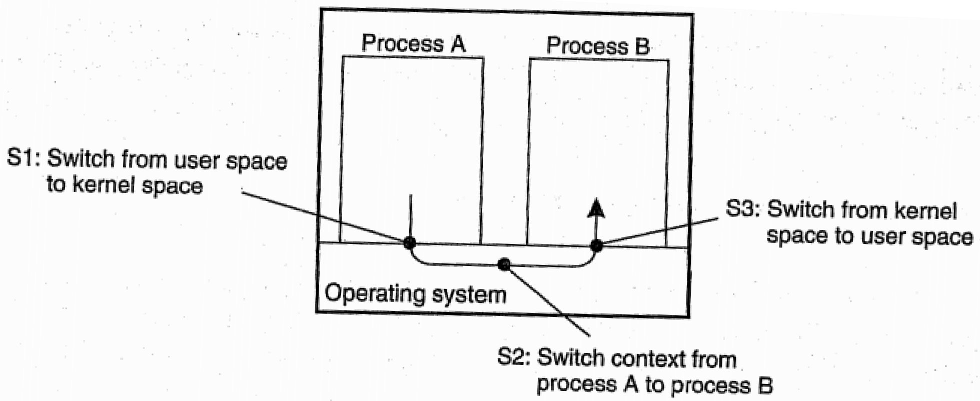


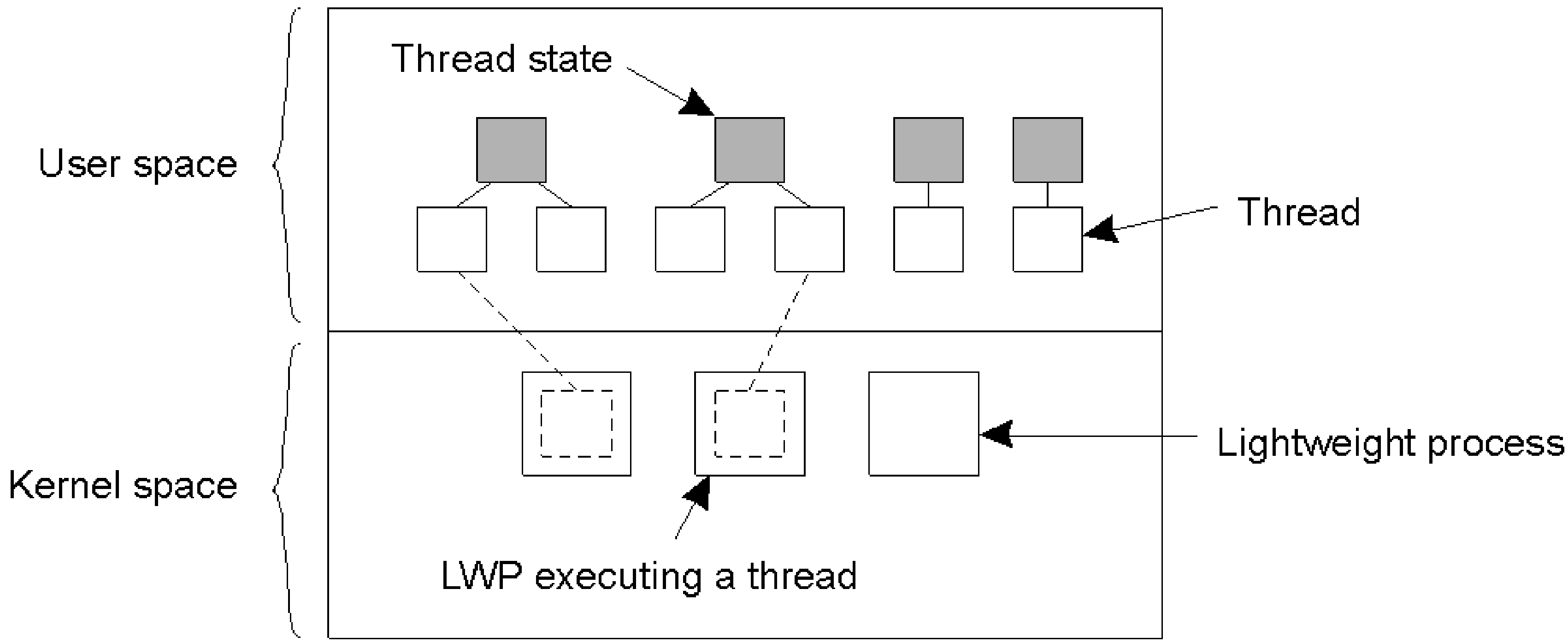
Figure 3-1. Context switching as the result of IPC.

Thread implementation

- Thread package:
  - Creating threads (1)
  - Destroying threads (2)
  - Synchronizing threads (3)
- (1), (2), (3) can be operated in user mode and kernel mode:
  - User mode:
    - Cheap to create and destroy threads
    - Easy to switch thread context
    - Invocation of a blocking system call will block the entire process
  - Kernel mode:

Lightweight processes (LWP) on Linux

- Combining kernel-level lightweight processes and user-level threads.

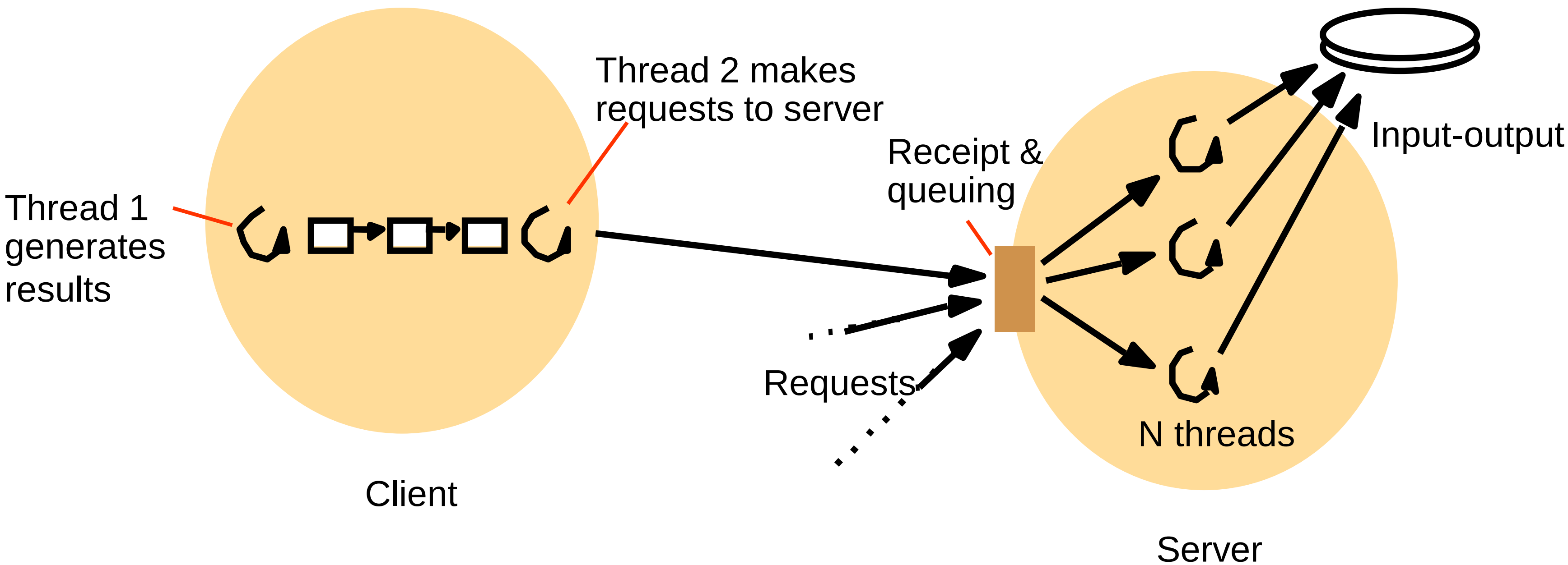


1.3. Threads in Distributed Systems

- Single-threaded server
  - One request at one moment
  - Sequentially
  - Do not guaranty the transparency

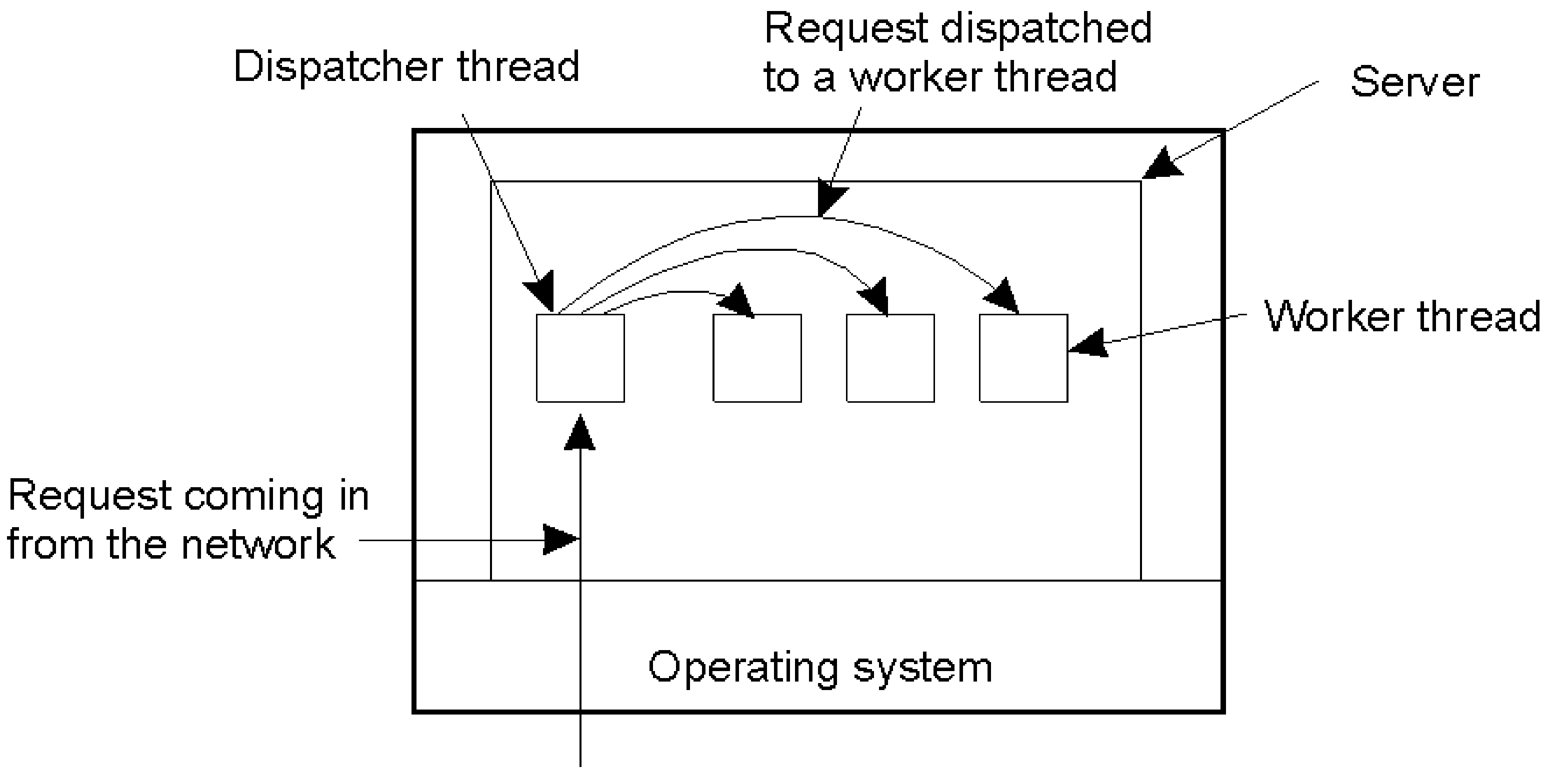
Multi-threaded server

Multithreaded Client and server

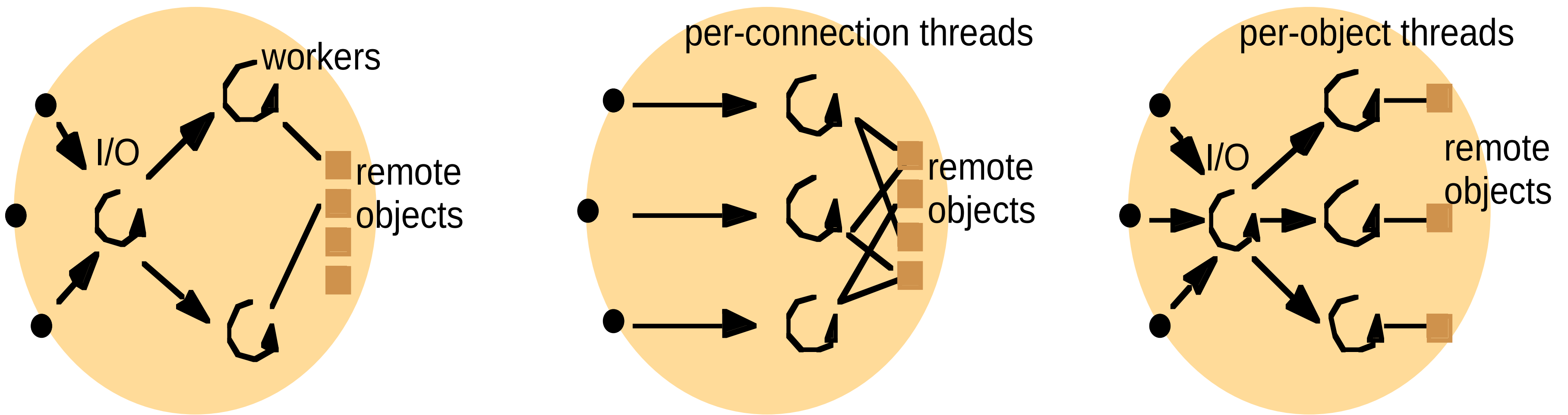


1.4. Server in Distributed Systems

Server dispatcher model



Multithreaded Server



a. Thread-per-request

b. Thread-per-connection

c. Thread-per-object

Finite-state machine

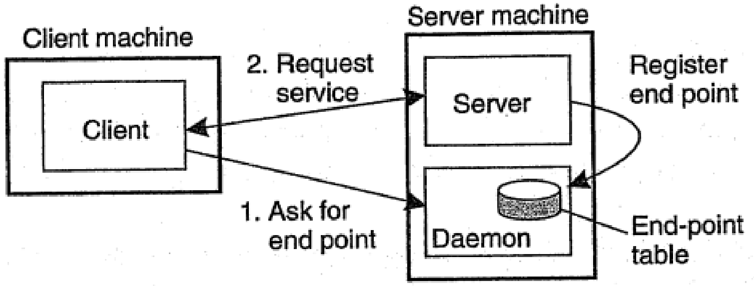
- Only one thread
- Non-blocking (asynchronous)
- Record the state of the current request in a table
- Simulating threads and their stacks
- Example: Node.js
  - Asynchronous and Event-driven
  - Single threaded but highly scalable

Comparison

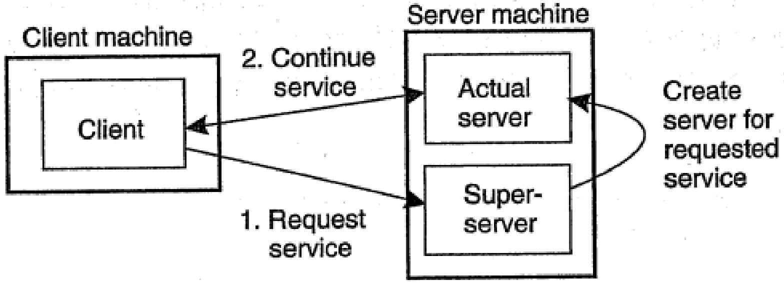
Model	Characteristics
Threads	Parallelism, Blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, Non-blocking system calls

General design issues

- Organize server
  - Iterative server
  - Concurrent server
- Find server:
  - End-point (port)
  - Deamon
  - Superserver
- Interrupt server
- Stateless & stateful server



(a)



(b)

Inetd

- Configuration info in the file */etc/inetd.conf*

```
telnet stream  tcp6  nowait root  /usr/sbin/telnetd   telnetd -a
```

service name

socket and protocol

do not wait

username

path

parameter

Example:

- A program *errorLogger.c*

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  const char *fn = argv[1];
  FILE *fp = fopen(fn, "a+");

  if(fp == NULL)
    exit(EXIT_FAILURE);

  char str[4096];
  //inetd passes its information to us in stdin.
  while(fgets(str, sizeof(str), stdin)) {
    fputs(str, fp);
    fflush(fp);
  }
  fclose(fp);
  return 0;
}
```

Configure inetd

- Insert info into */etc/services*

```
errorLogger 9999/udp
```

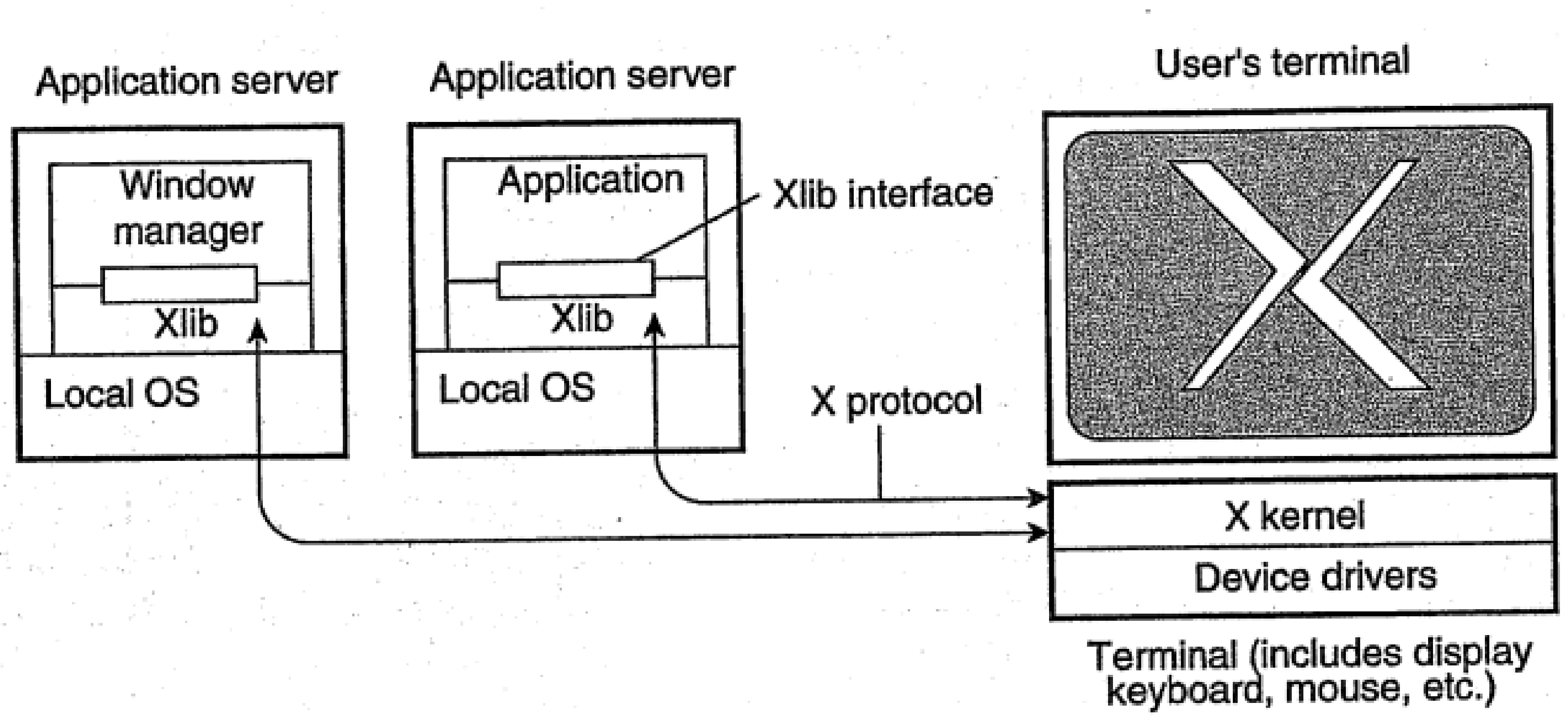
- Insert info into */etc/inetd.conf*

```
errorLogger dgram udp wait root
/usr/local/bin/errlogd errlogd
/tmp/logfile.txt
```

1.5. Client in Distributed Systems

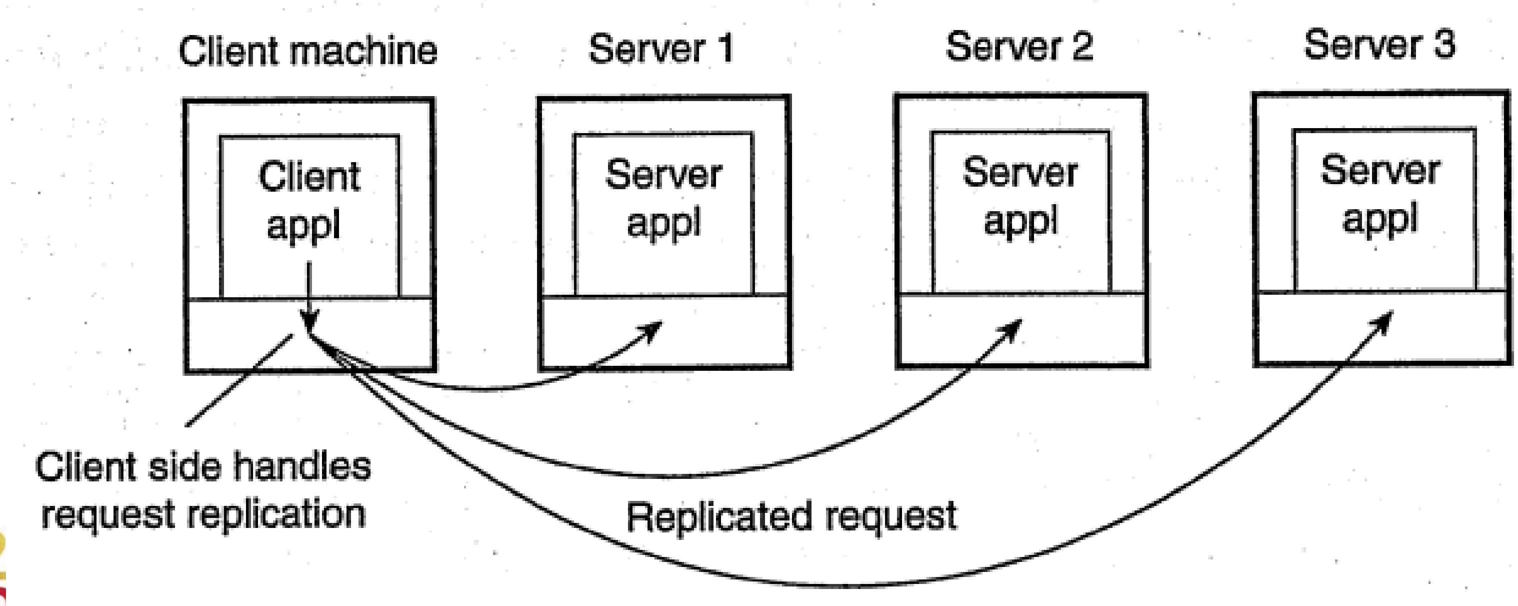
- Multi-threaded client:
  - Separate user interface and handle
  - Solve the problem of mutual exclusion
  - Enhance performance when working with many different servers
  - Example: Website loading

Solution for thin-client model: X Window System



Client-side software for distribution transparency

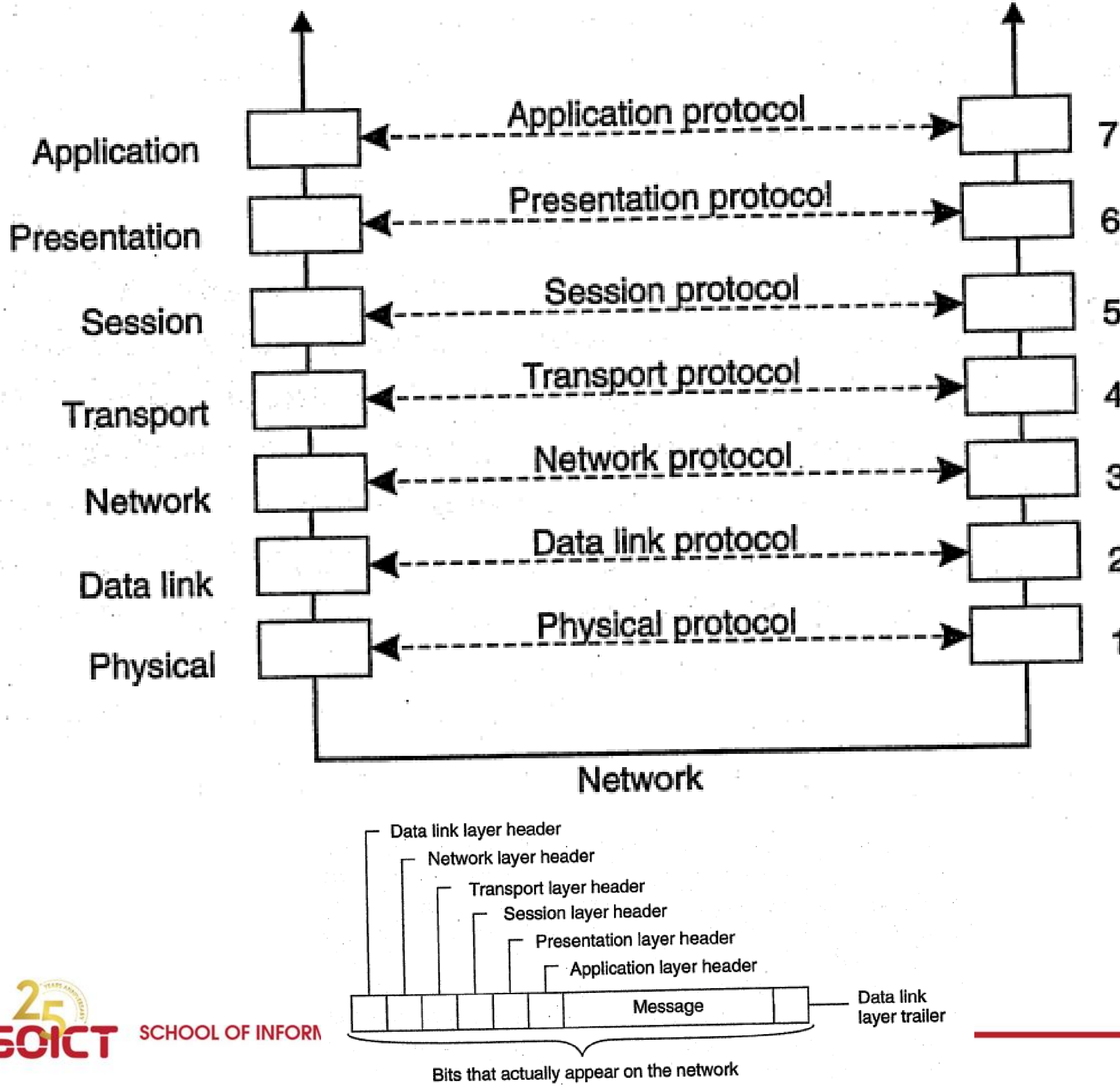
❖Transparent distribution:
- ❖Transparent access
- ❖Transparent migration
- ❖Transparent replication
- ❖Transparent faults



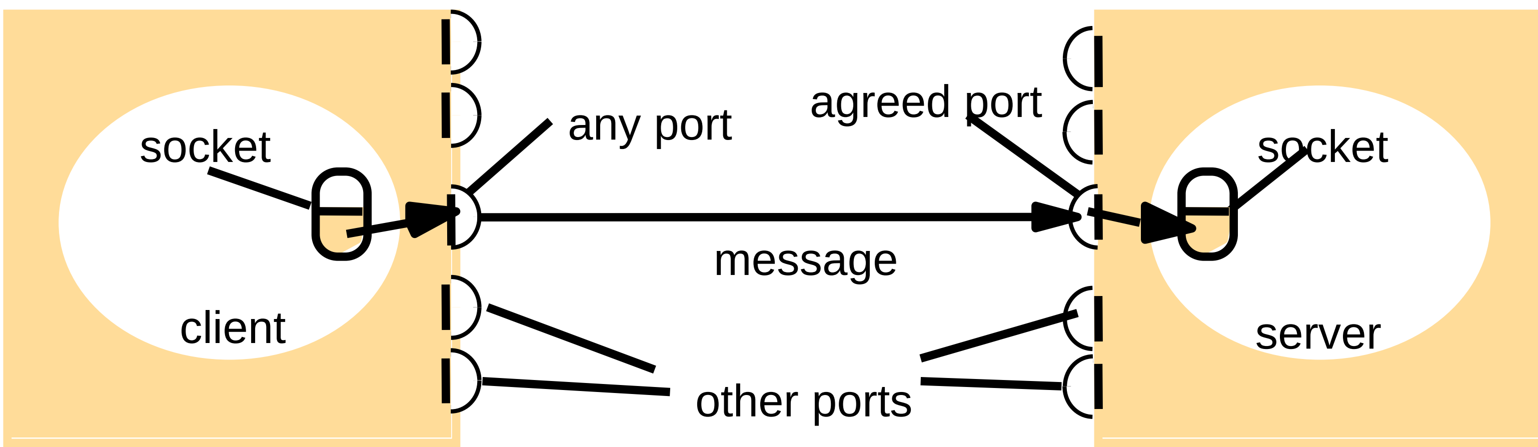
2. Communication

2.1. Definition

- 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

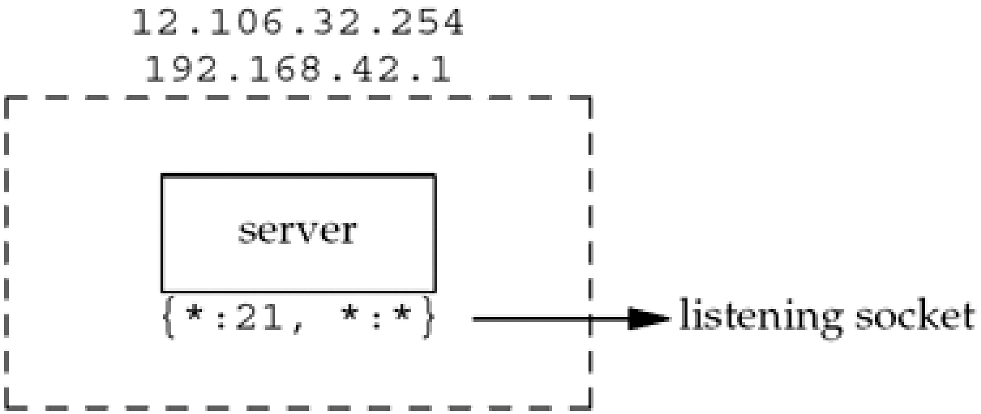
Socket-port



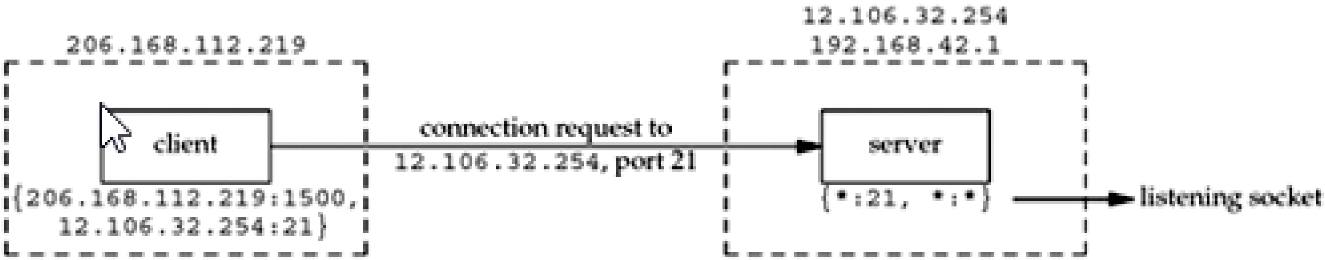
Internet address = 138.37.94.248

Internet address = 138.37.88.249

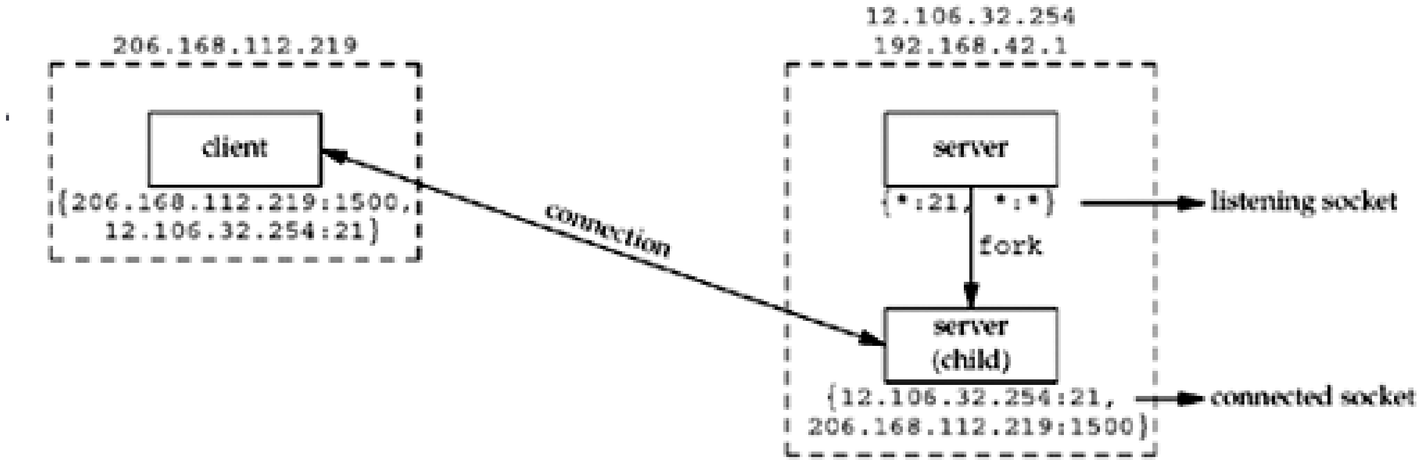
TCP Port Numbers and Concurrent Servers (1)



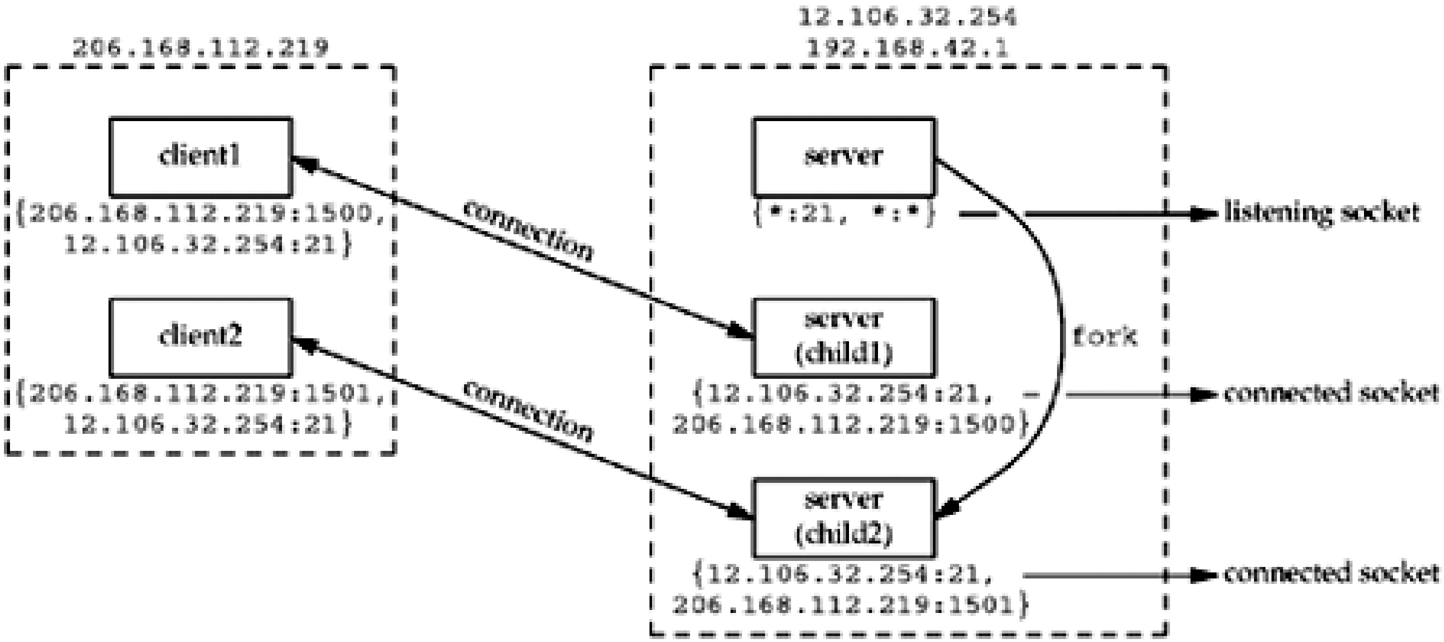
TCP Port Numbers and Concurrent Servers (2)



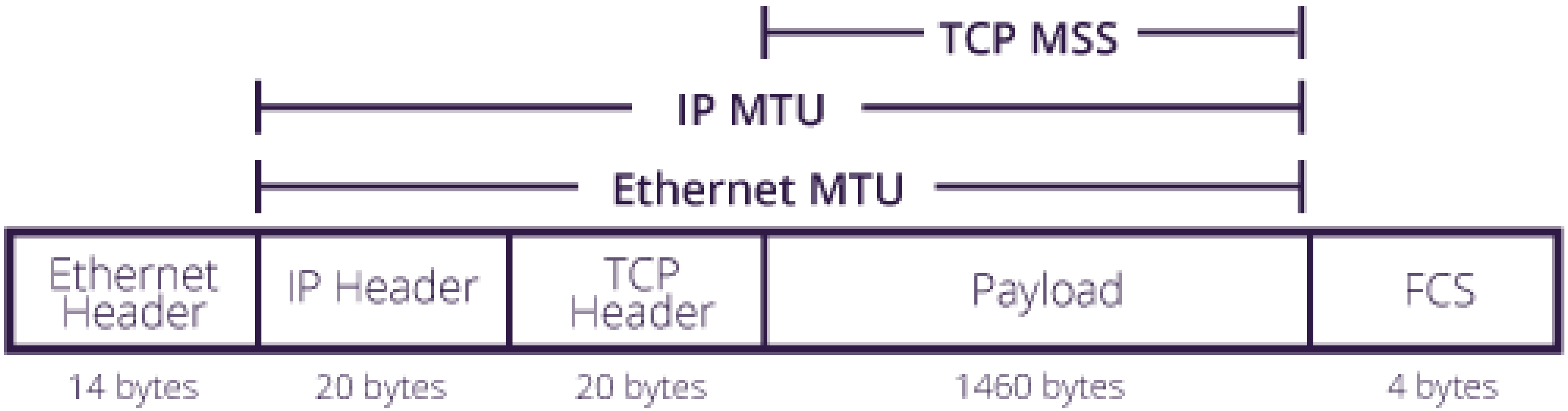
TCP Port Numbers and Concurrent Servers (3)



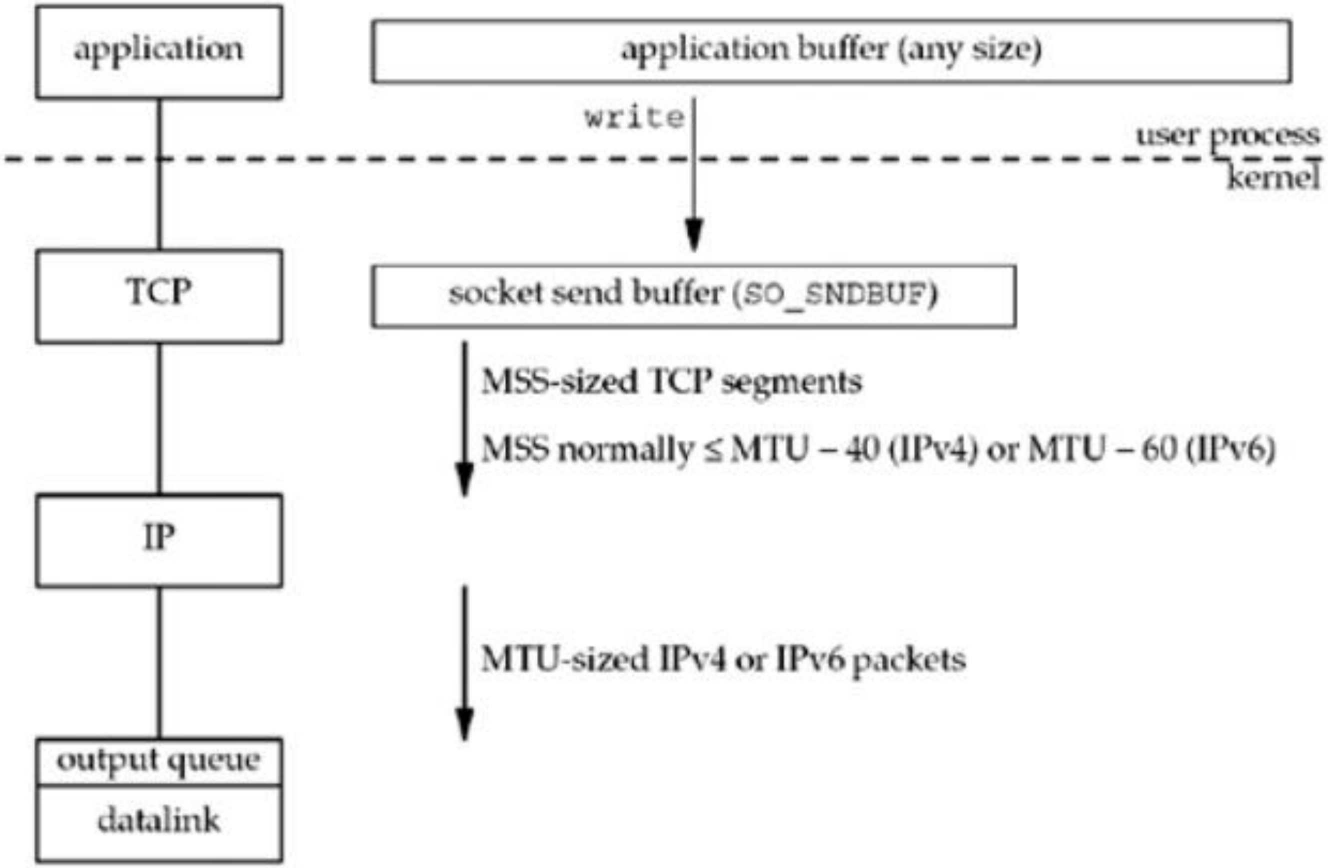
TCP Port Numbers and Concurrent Servers (4)



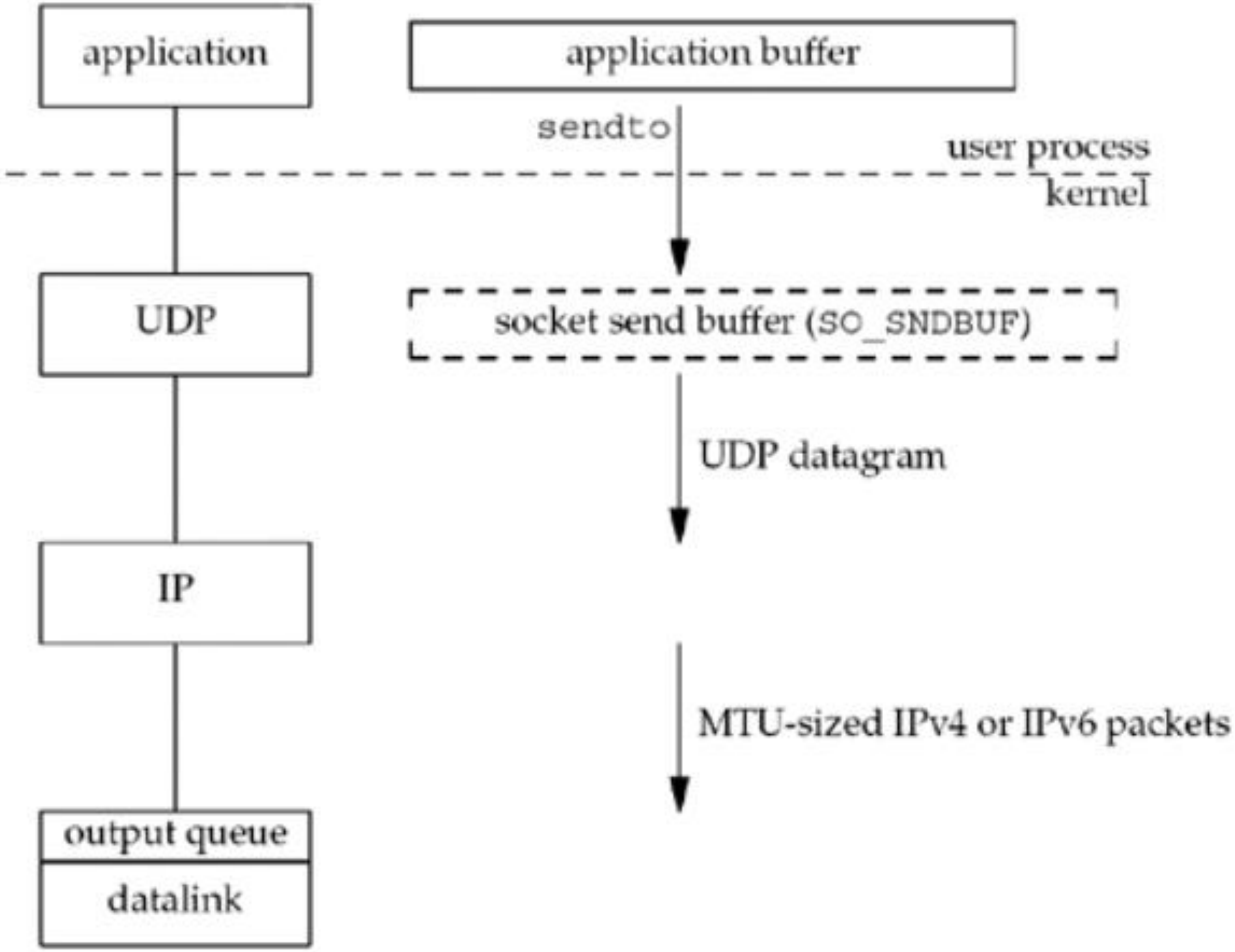
Buffer Sizes and Limitations



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

Communication with UDP and TCP

- UDP
  - connectionless protocol
    - No handshake protocol
  - Unreliable protocol
  - Asynchronous protocol
- TCP
  - connection-oriented protocol
    - With handshake protocol (SYN, SYN-ACK, ACK)
  - Reliable protocol
    - error recovery
    - acknowledgment segments
  - Synchronous protocol

3. Remote Procedure Call

3.1. Request-reply protocol

3.2. RPC

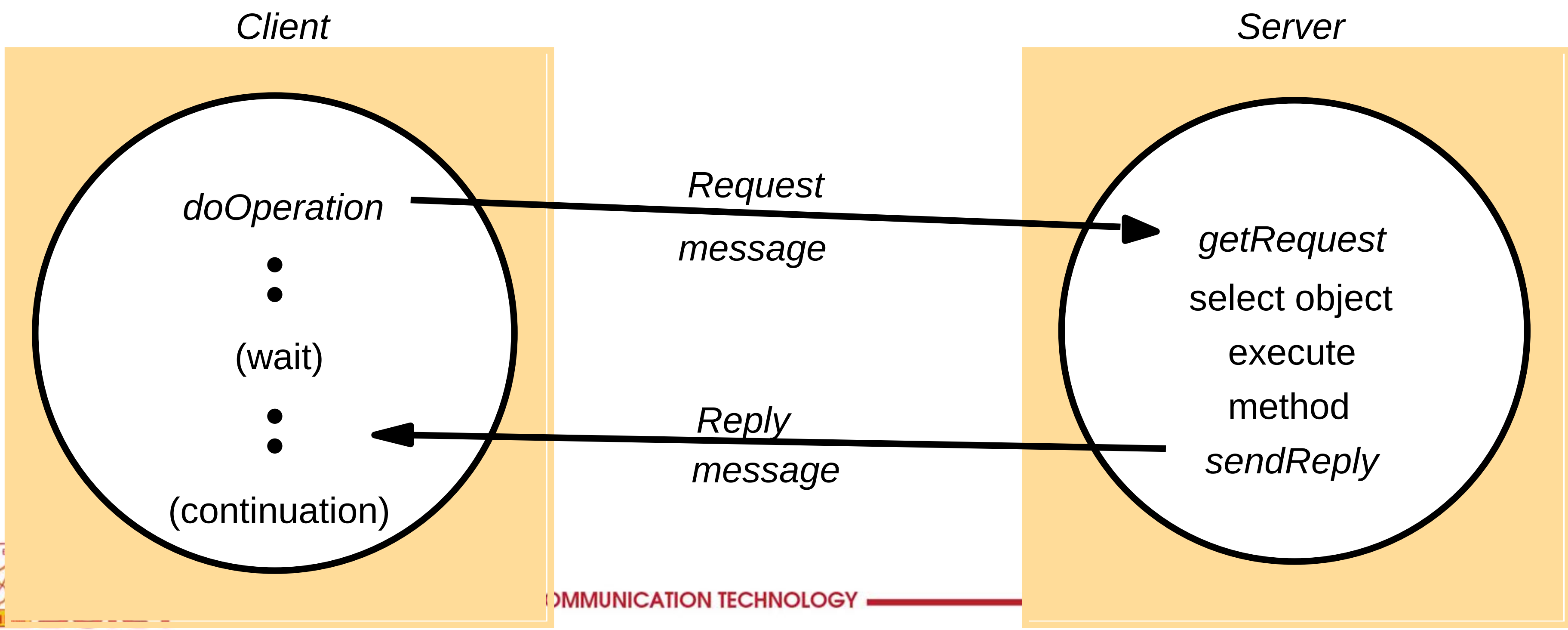
3.3. RMI

3.1. 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



Message structure

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

Example: HTTP

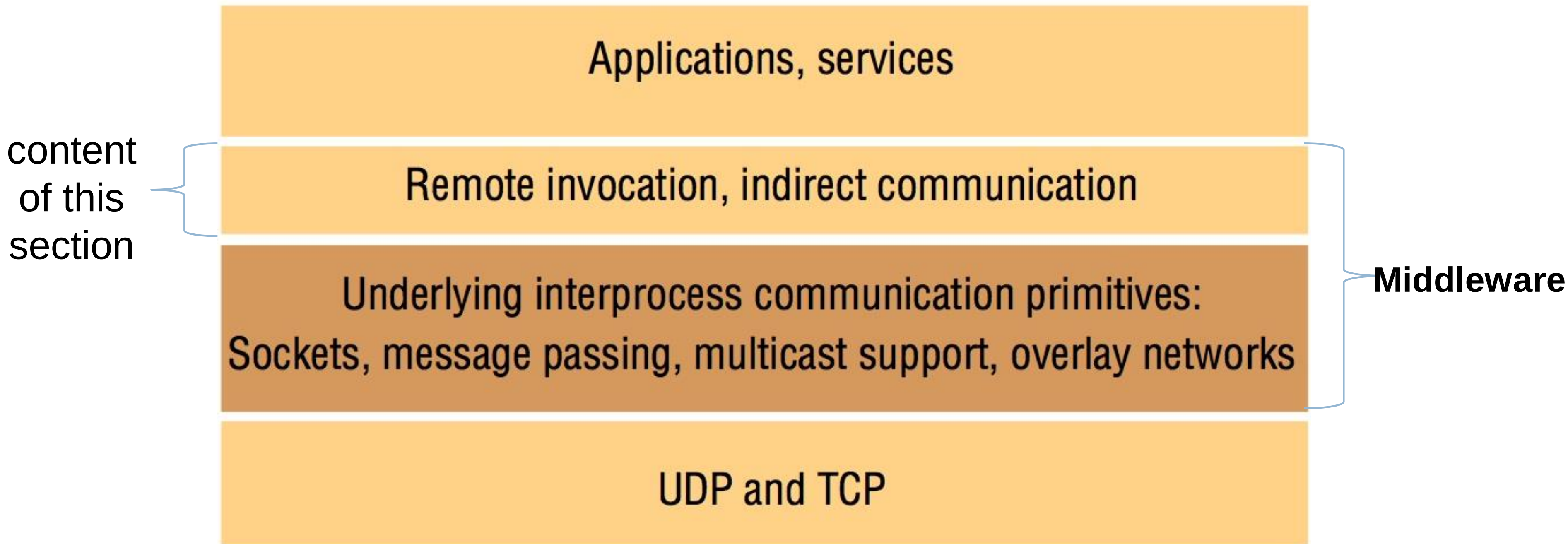
HTTP *request* message

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

HTTP *reply* message

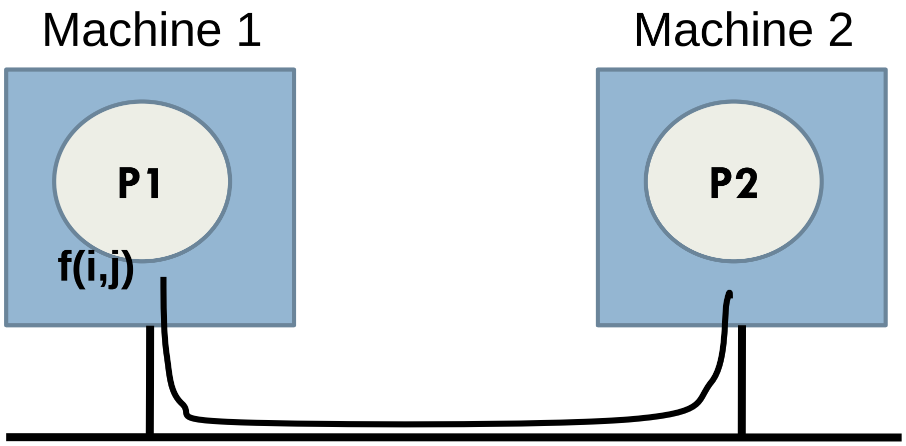
HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

3.2. RPC (Remote Procedure Call)



Remote Procedure Call

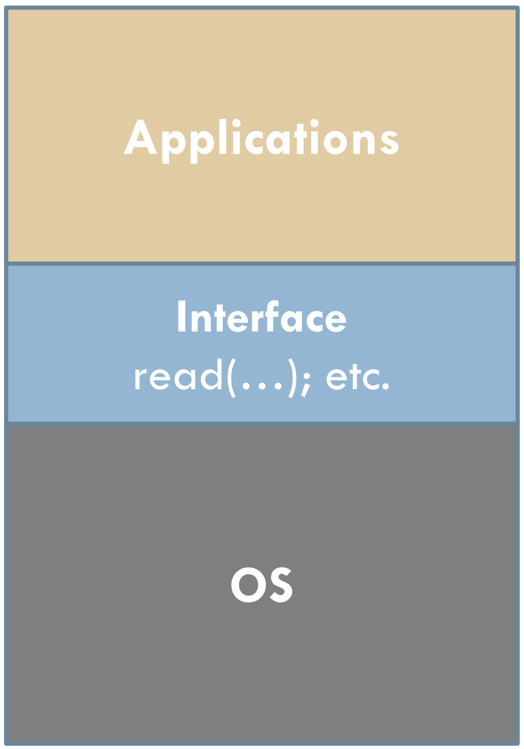
- Access transparency
- Issues:
  - Heterogenous system
    - Different memory space
    - Different information representation
  - Faults appear

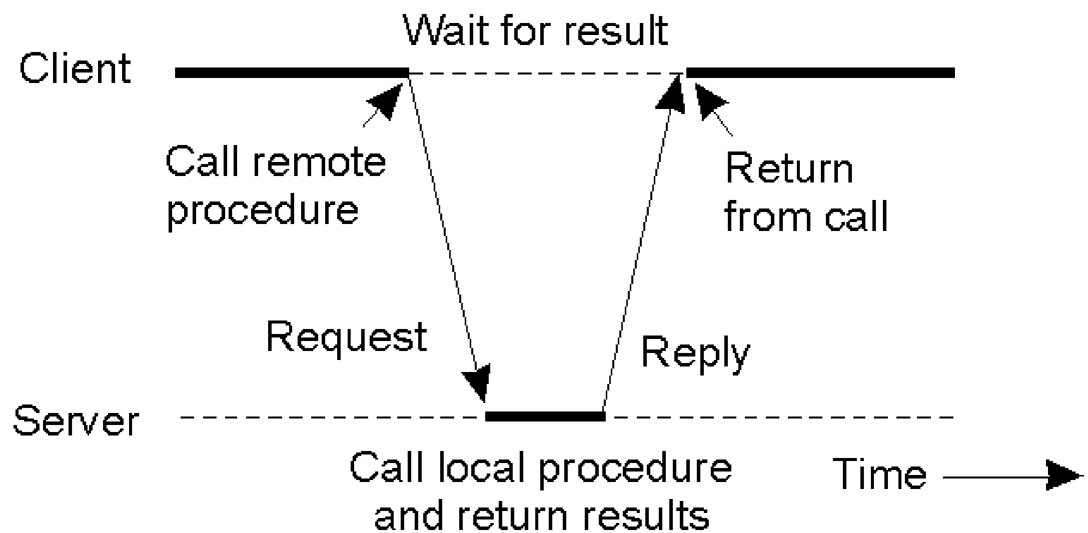


Parameters

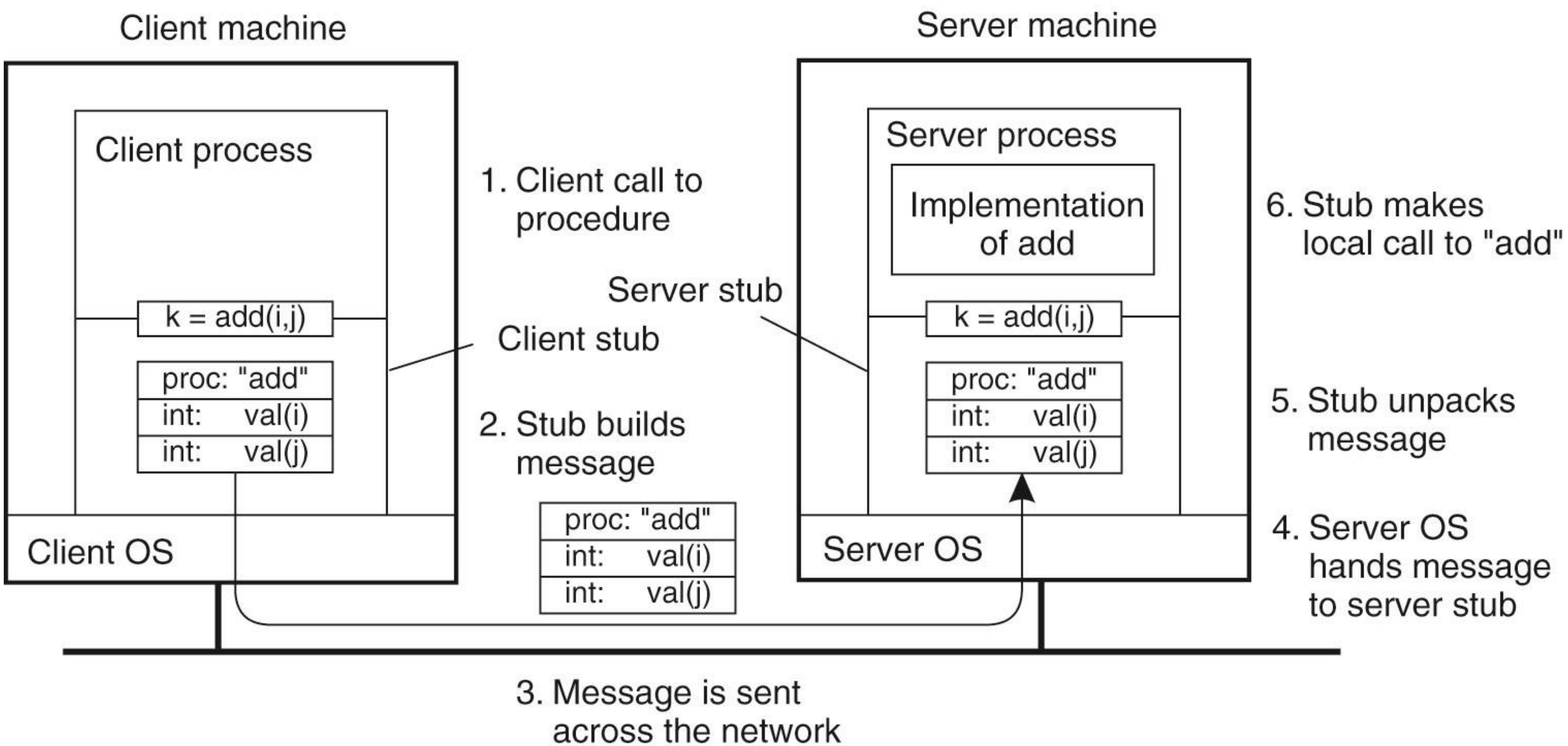
- Call-by-value
- Call-by-reference
- Call-by-copy/restore
  - Copy the variables to the stack
  - Copy back after the call, overwrite caller's the original value

RPC mechanism





RPC mechanism

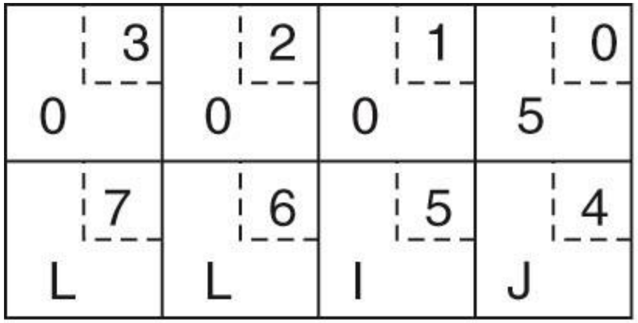


Passing Value Parameters

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

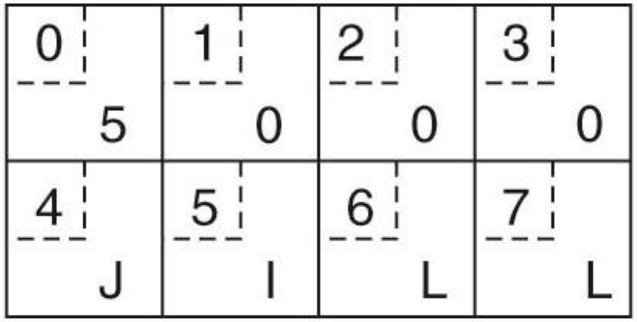
Issue: Different character format

Intel Pentium (little endian)

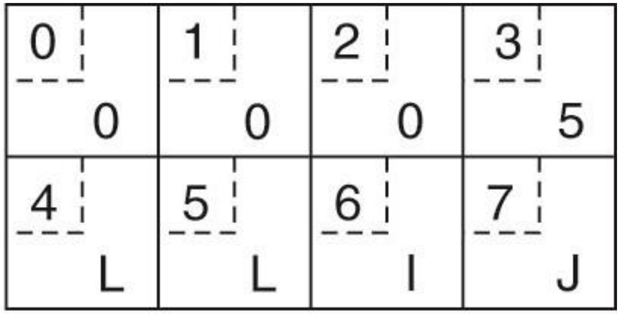


(a)

SPARC (big endian)



(b)



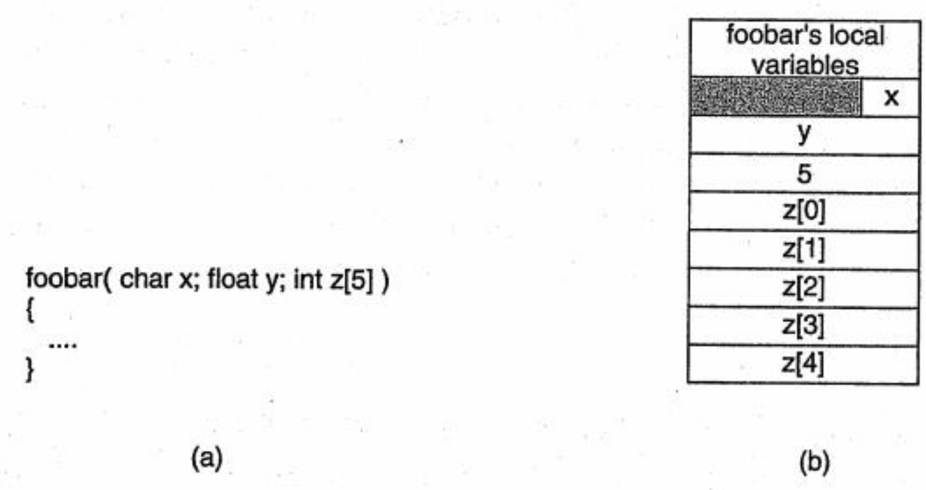
(c)

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

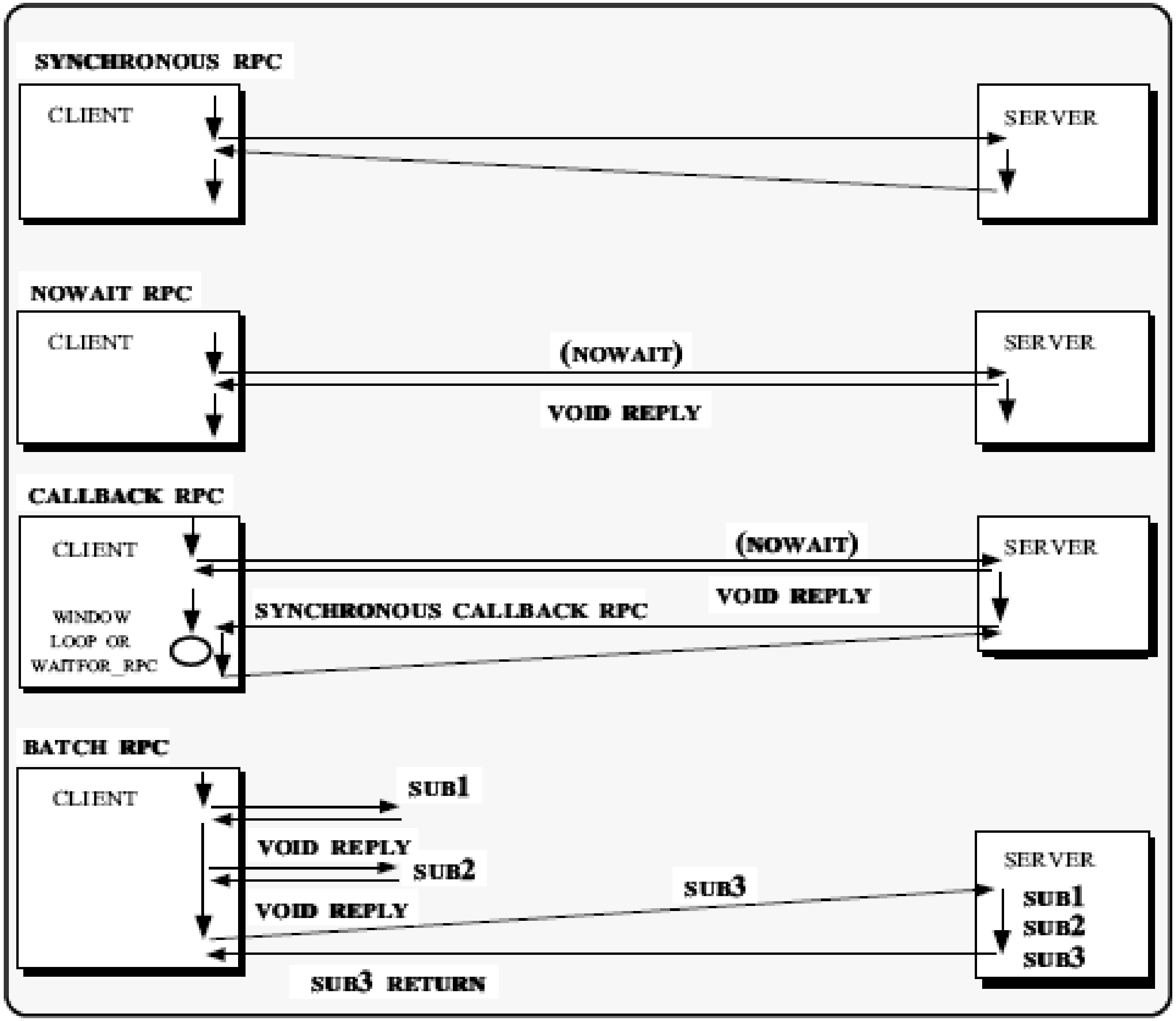
- The caller and the callee agree on the format of the messages they exchange.
- Agreements:
  - Message format
  - Representation of simple data structures (integers, characters, Booleans, etc.)
  - Method for exchanging messages.
  - Client-stub and server-stub need to be implemented.



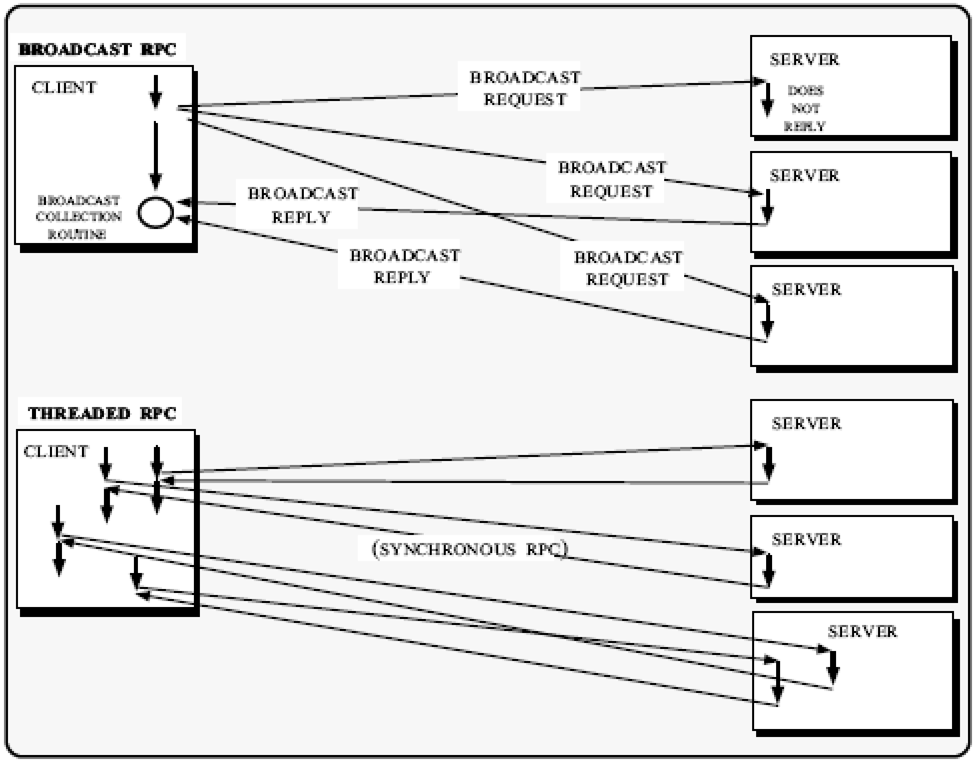
Openness of RPC

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

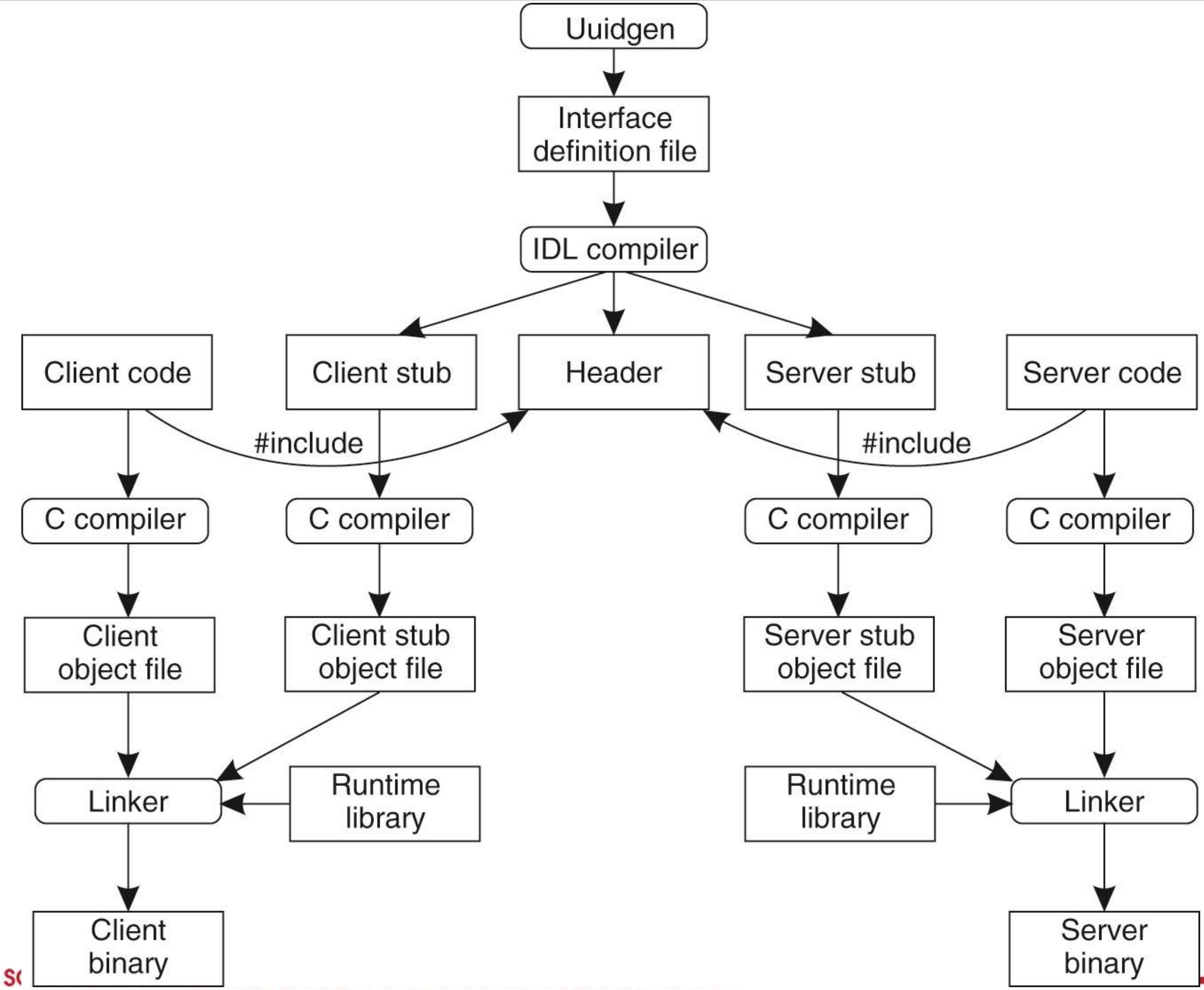
RPC models


SYNCHRONOUS RPC
CLIENT
SERVER
NOWAIT RPC
CLIENT
(NOWAIT)
SERVER
VOID REPLY
CALLBACK RPC
CLIENT
(NOWAIT)
SERVER
VOID REPLY
SYNCHRONOUS CALLBACK RPC
WINDOW
LOOP OR
WAITFOR_RPC
BATCH RPC
CLIENT
SUB1
VOID REPLY
SUB2
VOID REPLY
SUB3
SERVER
SUB1
SUB2
SUB3
SUB3 RETURN

RPC models


BROADCAST RPC
CLIENT
BROADCAST COLLECTION ROUTINE
BROADCAST REQUEST
BROADCAST REQUEST
BROADCAST REQUEST
BROADCAST REPLY
BROADCAST REPLY
SERVER
DOES NOT REPLY
SERVER
SERVER
THREADED RPC
CLIENT
(SYNCHRONOUS RPC)
SERVER
SERVER
SERVER

Implementing RPC



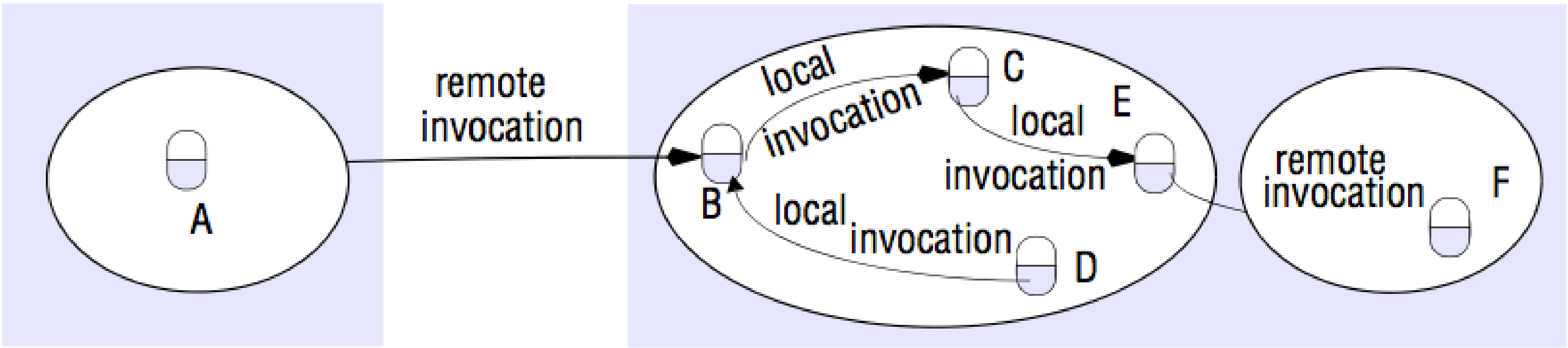
Mini tutorial: RPC on Windows

- Link: https://docs.microsoft.com/en-us/windows/win32/rpc/tutorial
- Create interface definition and application configuration files.
- Use the MIDL compiler to generate C-language client and server stubs and headers from those files.
- Write a client application that manages its connection to the server.
- Write a server application that contains the actual remote procedures.
- Compile and link these files to the RPC run-time library to produce the distributed application.

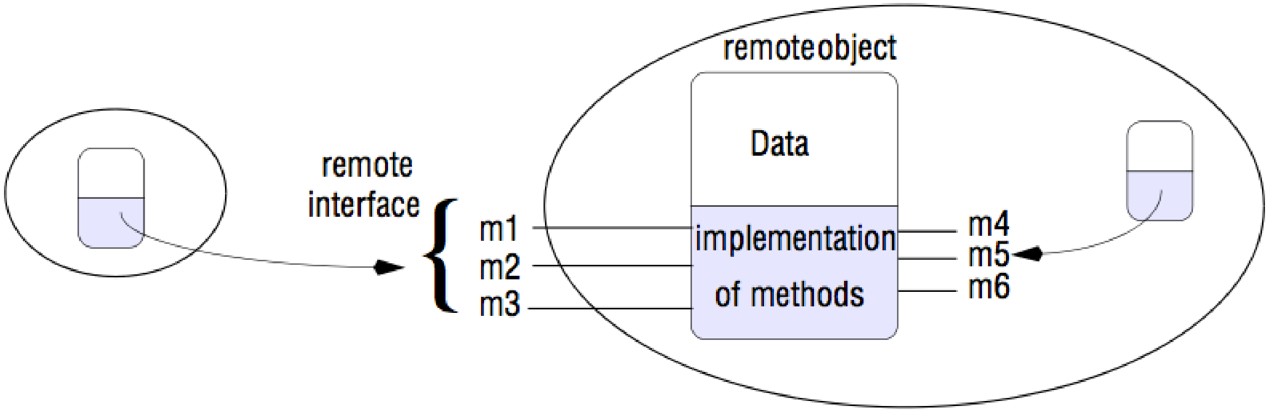
2.4. RMI (Remote Method Invocation)

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

Distributed objects model



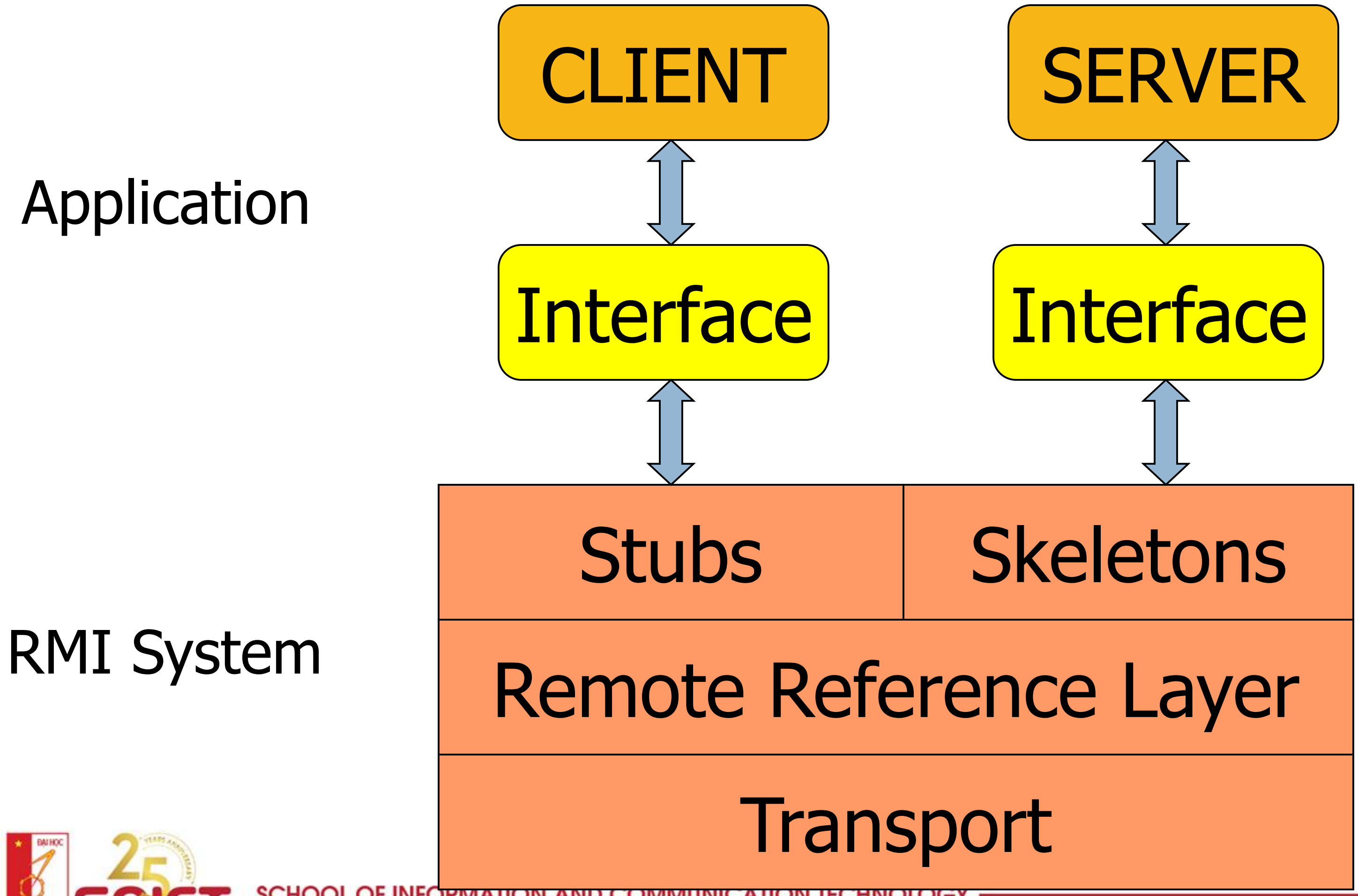
Remote object and Remote interface



Characteristics

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

RMI Architecture



Mini tutorial:

- Link: https://docs.oracle.com/javase/tutorial/rmi/index.html
- Overview of RMI
- Writing RMI server
- Creating RMI client
- Compiling & Running

4. Message-oriented communication

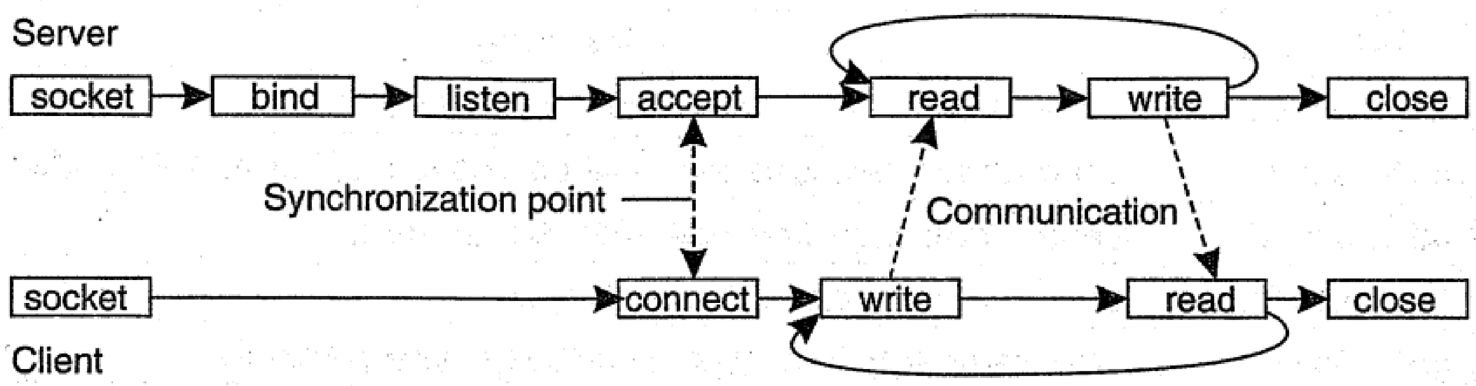
4.1. Message-oriented transient communication

4.2. Message-oriented persistent communication

4.1. 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



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

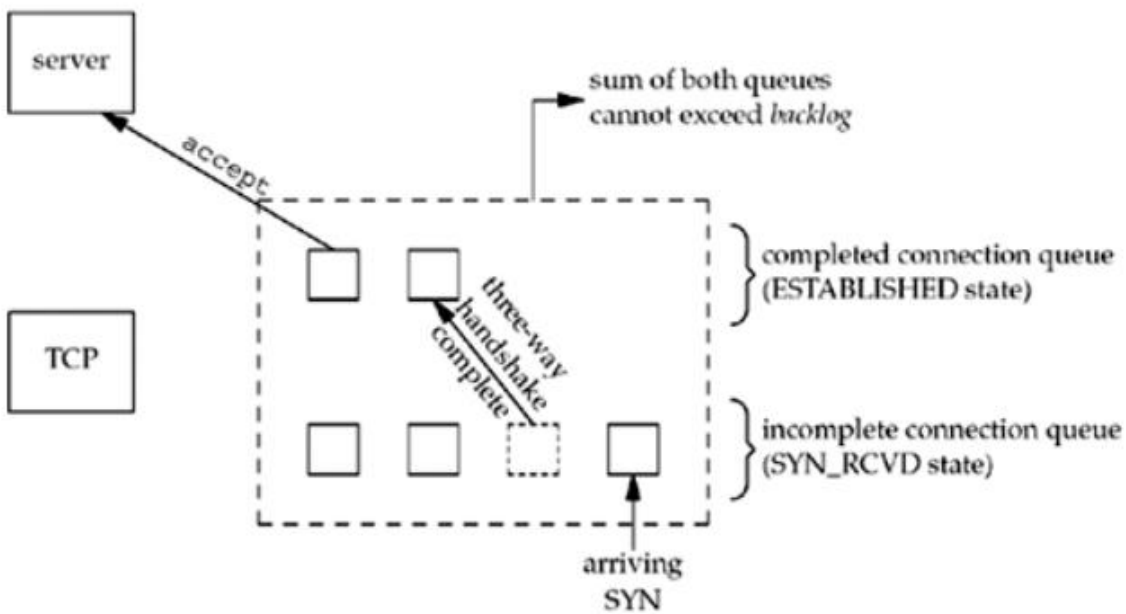
- The listen function is called only by a TCP server.
- When a socket is created by the *socket* function, it is assumed to be an active socket, that is, a client socket that will issue a *connect*.
- The *listen* function converts an unconnected socket into a passive socket, indicating that the kernel 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

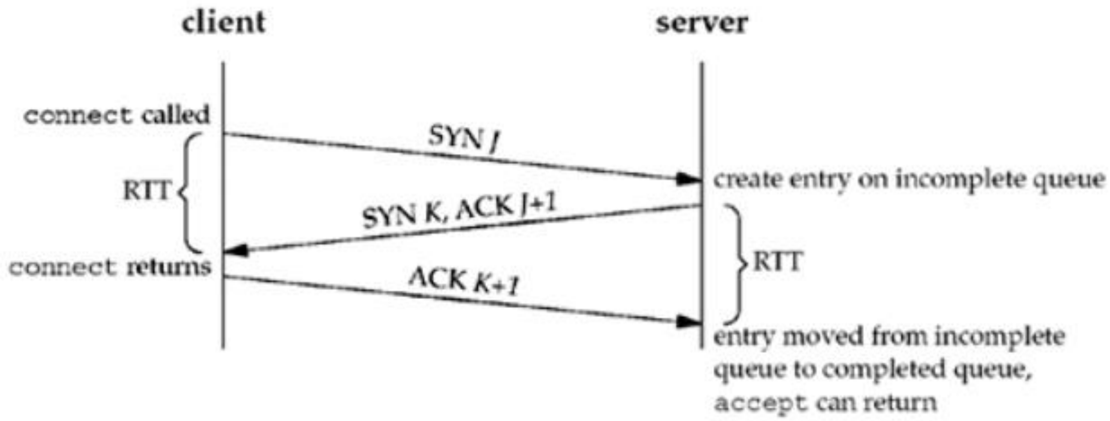
listen Function (2)

- 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

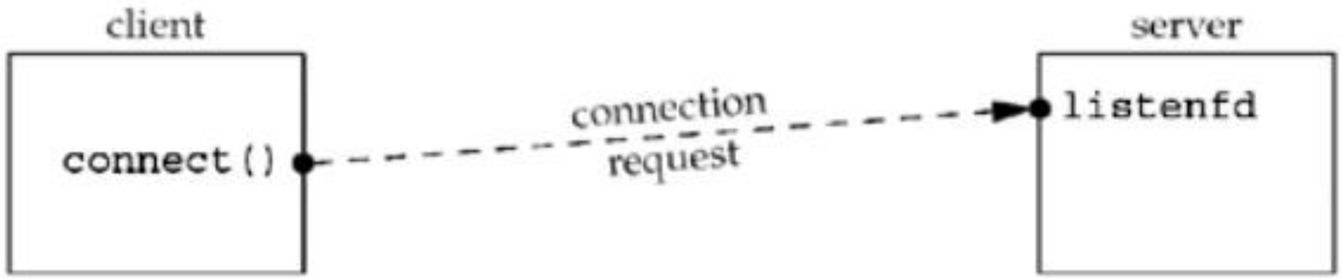
```c
#include <stdio.h>
#include <unistd.h>

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;
}
```

Concurrent Servers

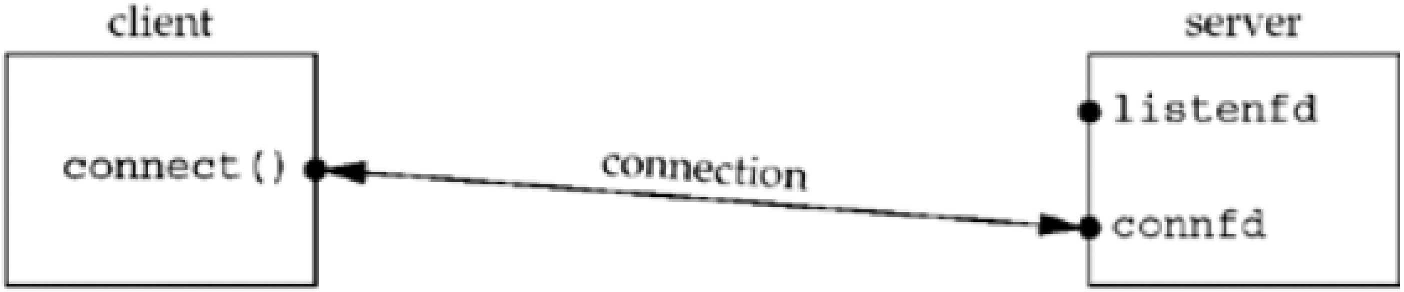
- *fork* a child process to handle each client

```
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 */
}
```

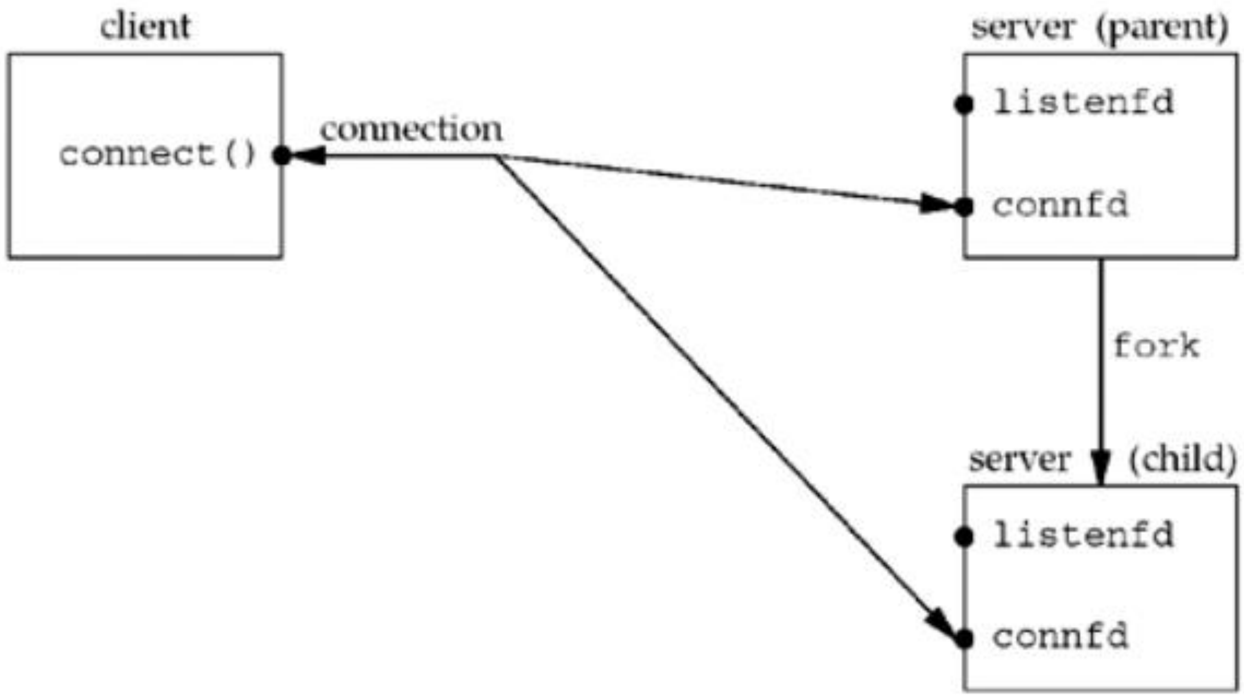
Status of client/server before call to *accept* returns.



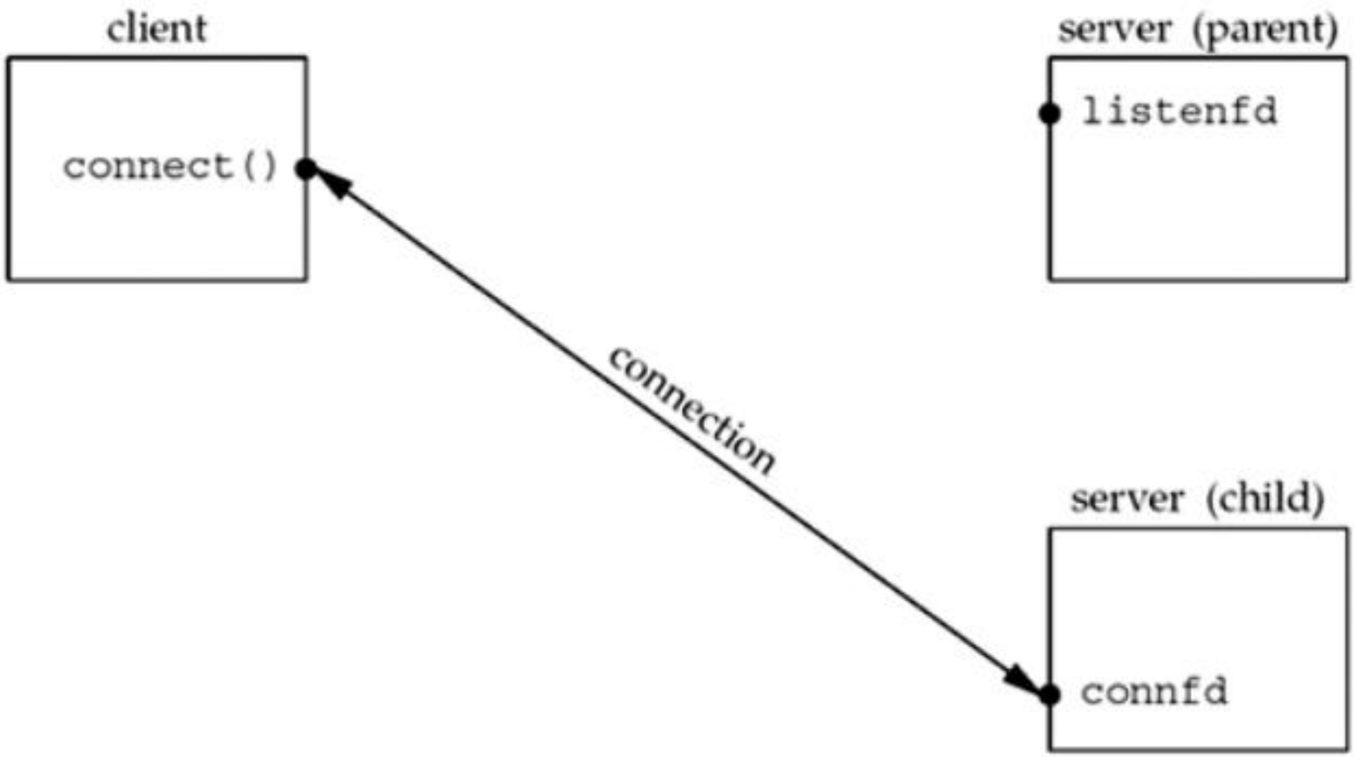
Status of client/server after return from *accept*.



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

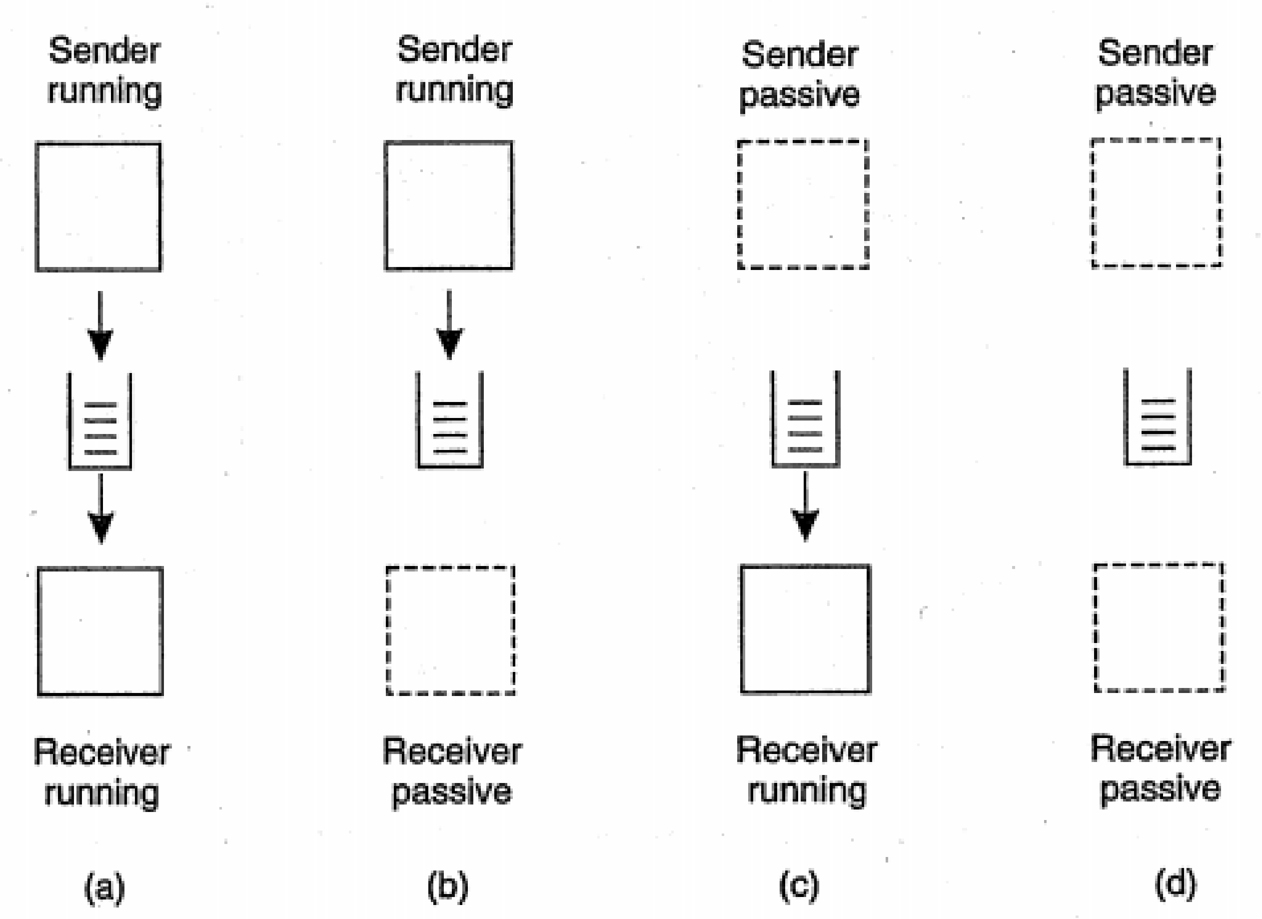
Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
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

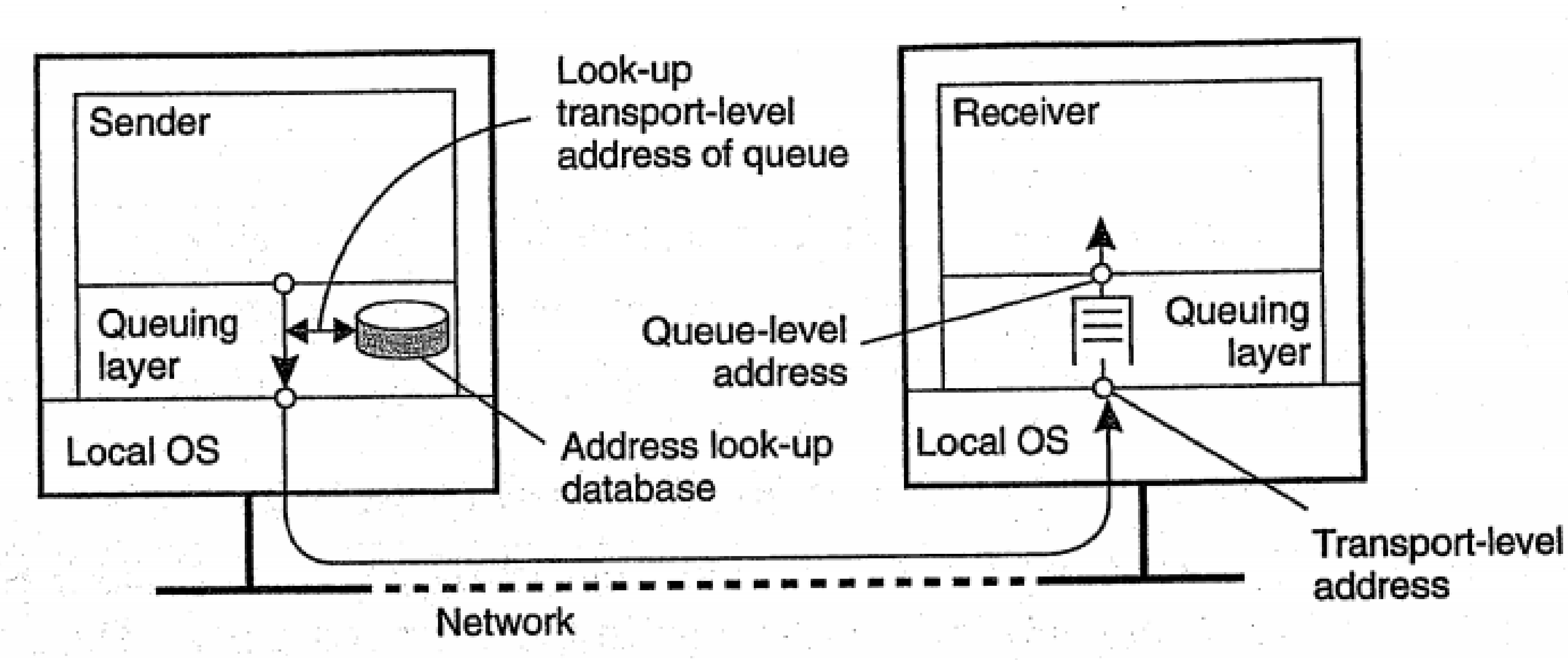
4.2. Message-Oriented Persistent Communication

- 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

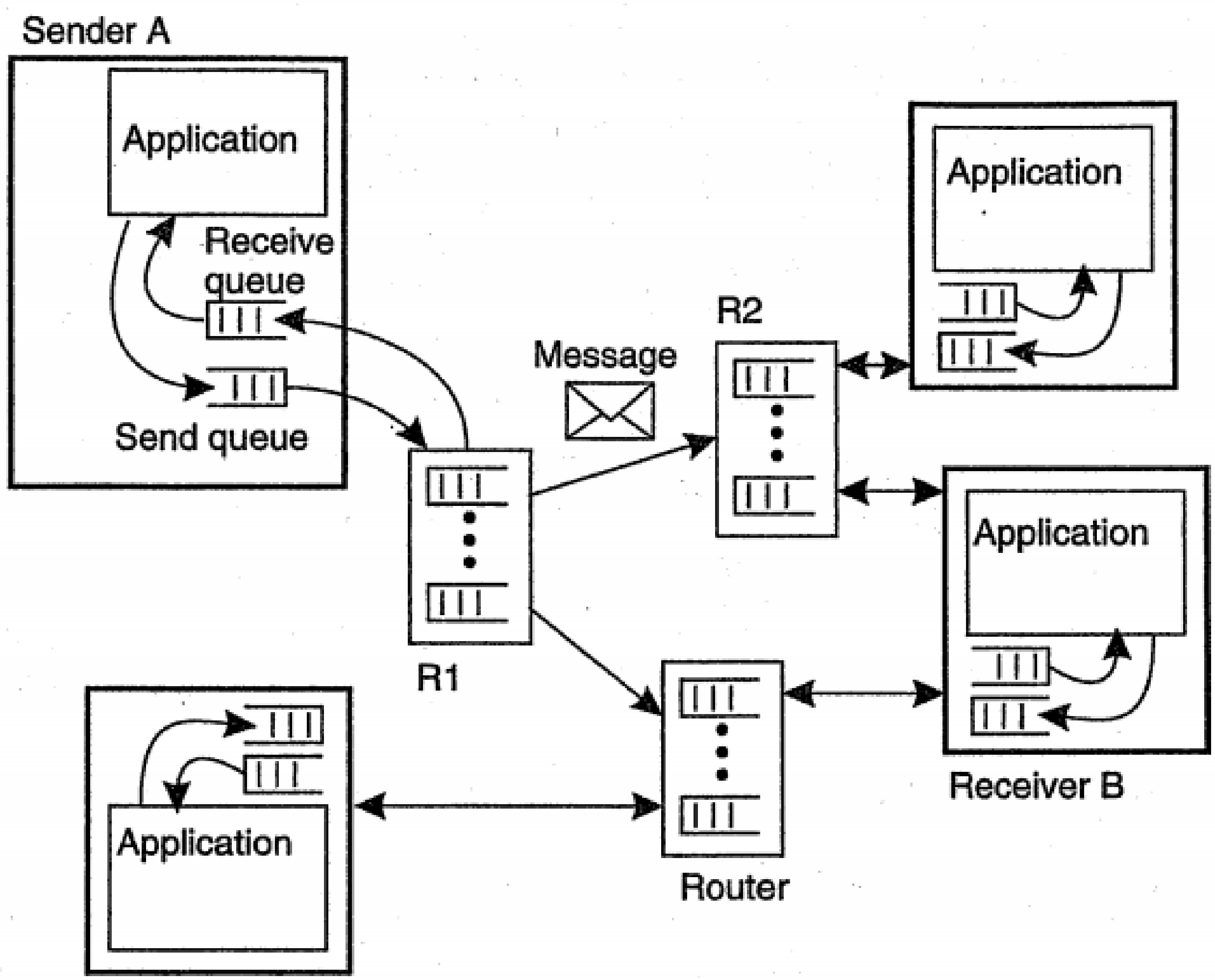
Message-Queuing System



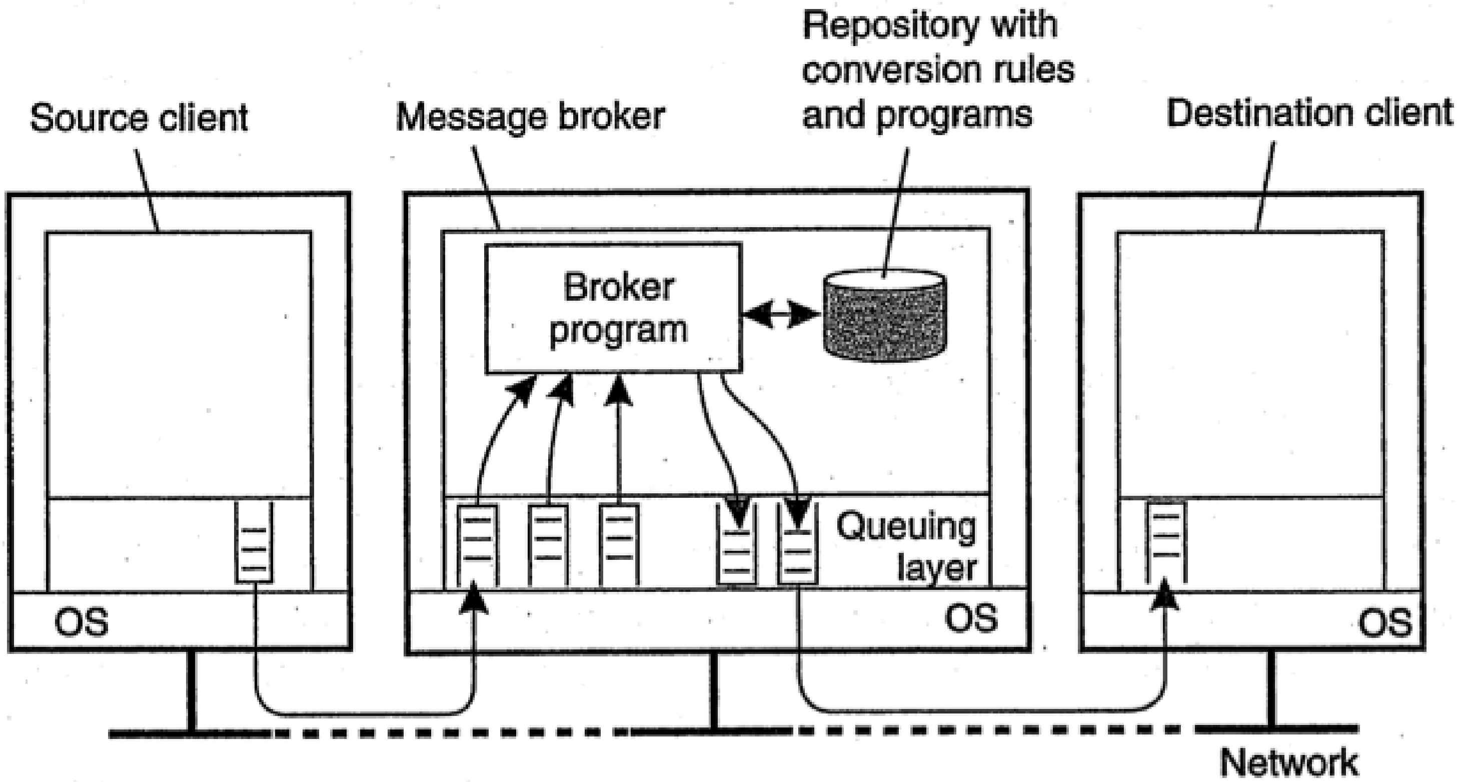
The relationship between queue-level addressing and network-level addressing



Routing with Queueing system



Message Broker



RabbitMQ

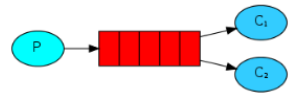
1 "Hello World!"

The simplest thing that does *something*

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

2 Work queues

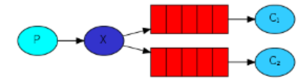
Distributing tasks among workers



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

3 Publish/Subscribe

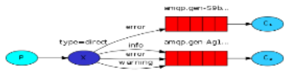
Sending messages to many consumers at once



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

4 Routing

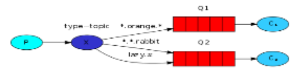
Receiving messages selectively



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

5 Topics

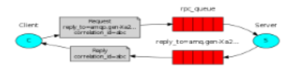
Receiving messages based on a pattern



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

6 RPC

Remote procedure call implementation



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

5. Stream-oriented Communication

5.1. Support for Continuous Media

5.2. Streams and QoS

5.3. Stream synchronization

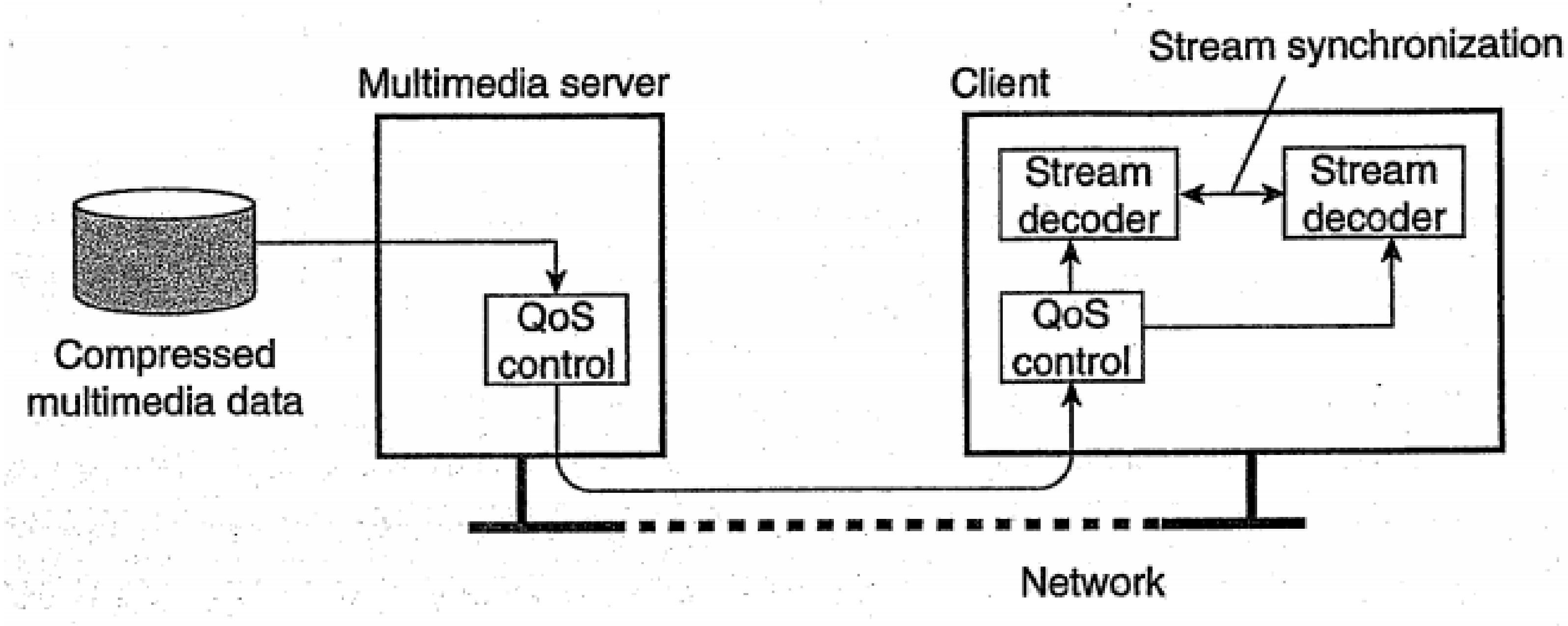
4.1. Support for Continuous Media

- 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
  - asynchronous
  - synchronous
  - isochronous
- A simple stream: only a single sequence of data
- A complex stream: several related simple streams
- Issues:
  - Data compression
  - QoS
  - Synchronization

Data stream (cont.)



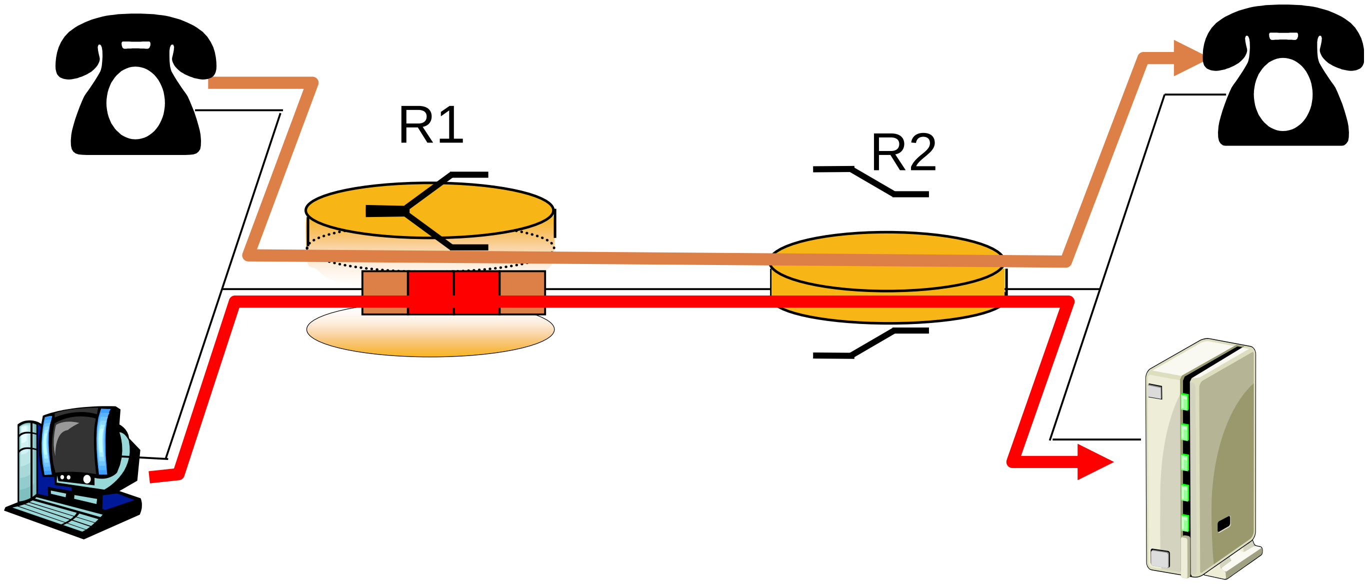
A general architecture for streaming stored multimedia data over a network

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

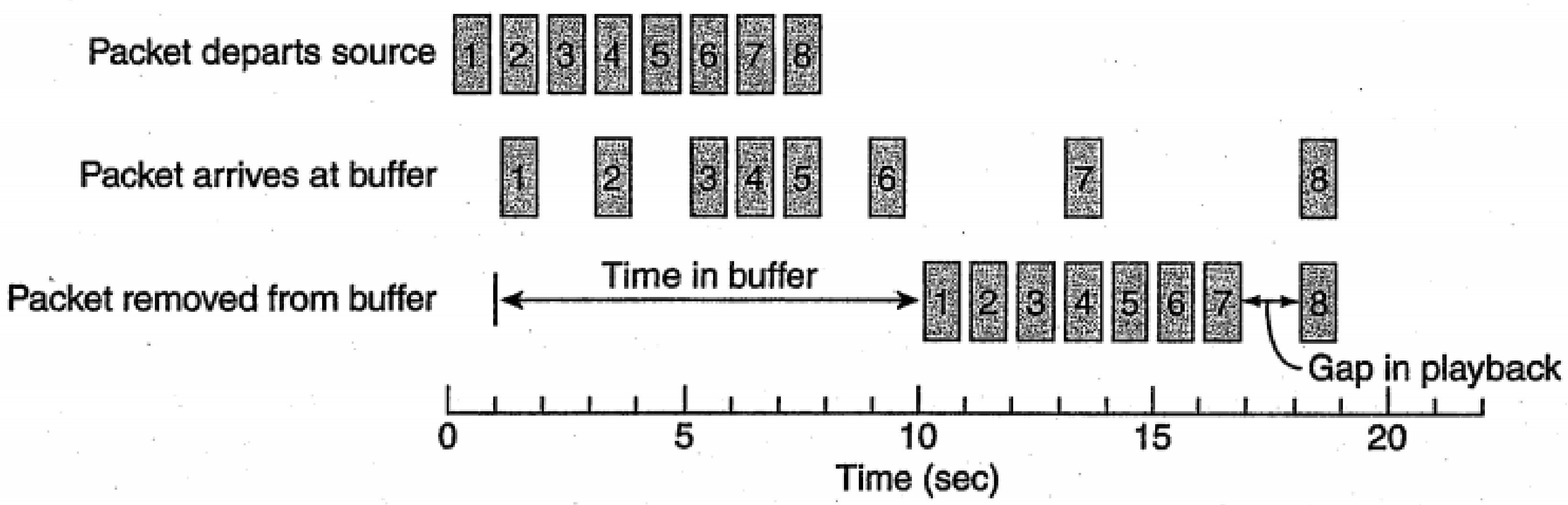
Enforcing QoS

- Differentiated services



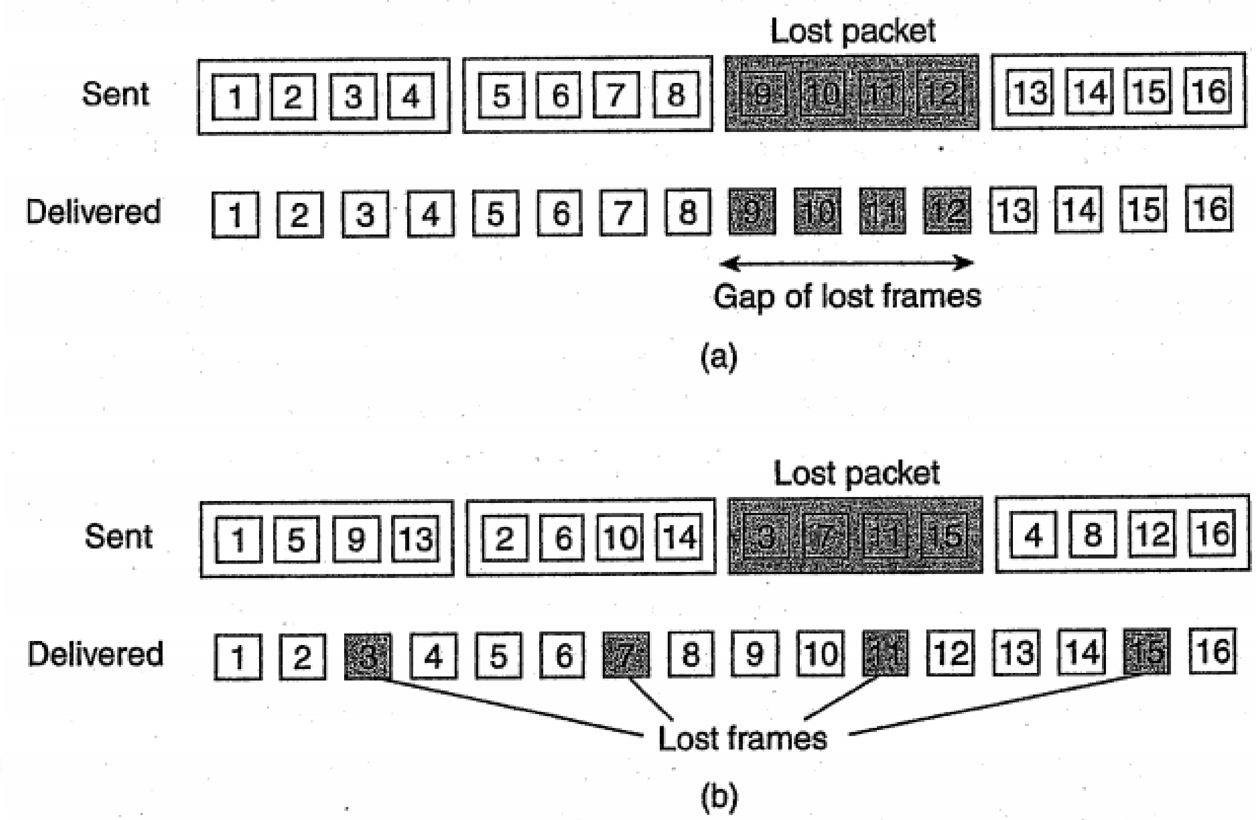
Enforcing QoS (cont.)

- Using a buffer to reduce jitter

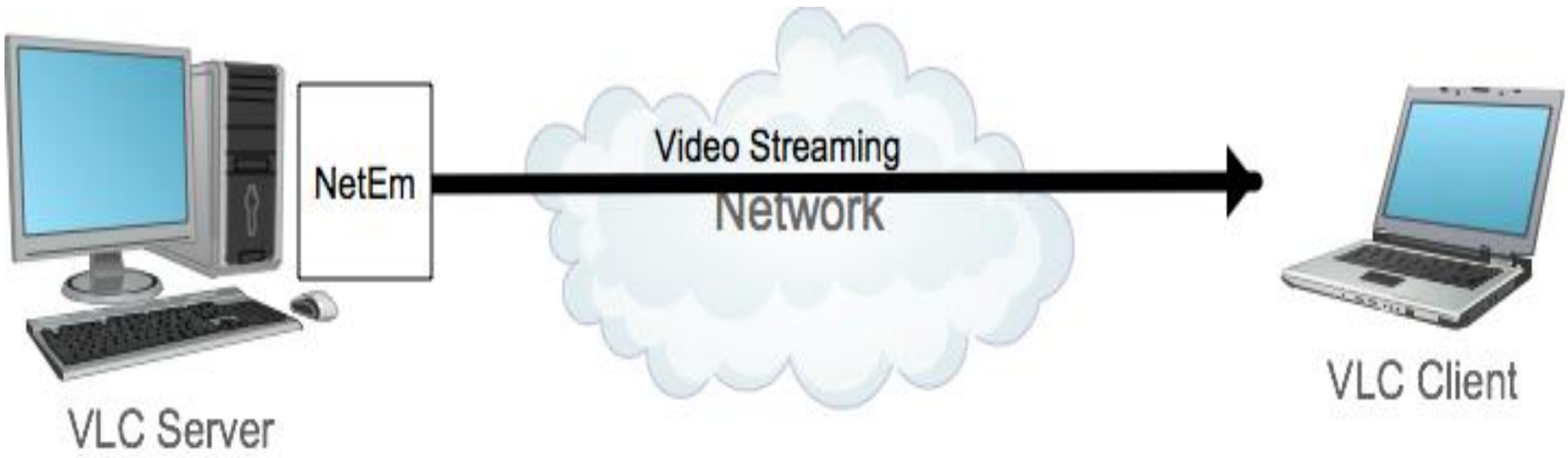


Enforcing QoS (cont.)

- Forward error correction (FEC)
  - Interleaved transmission



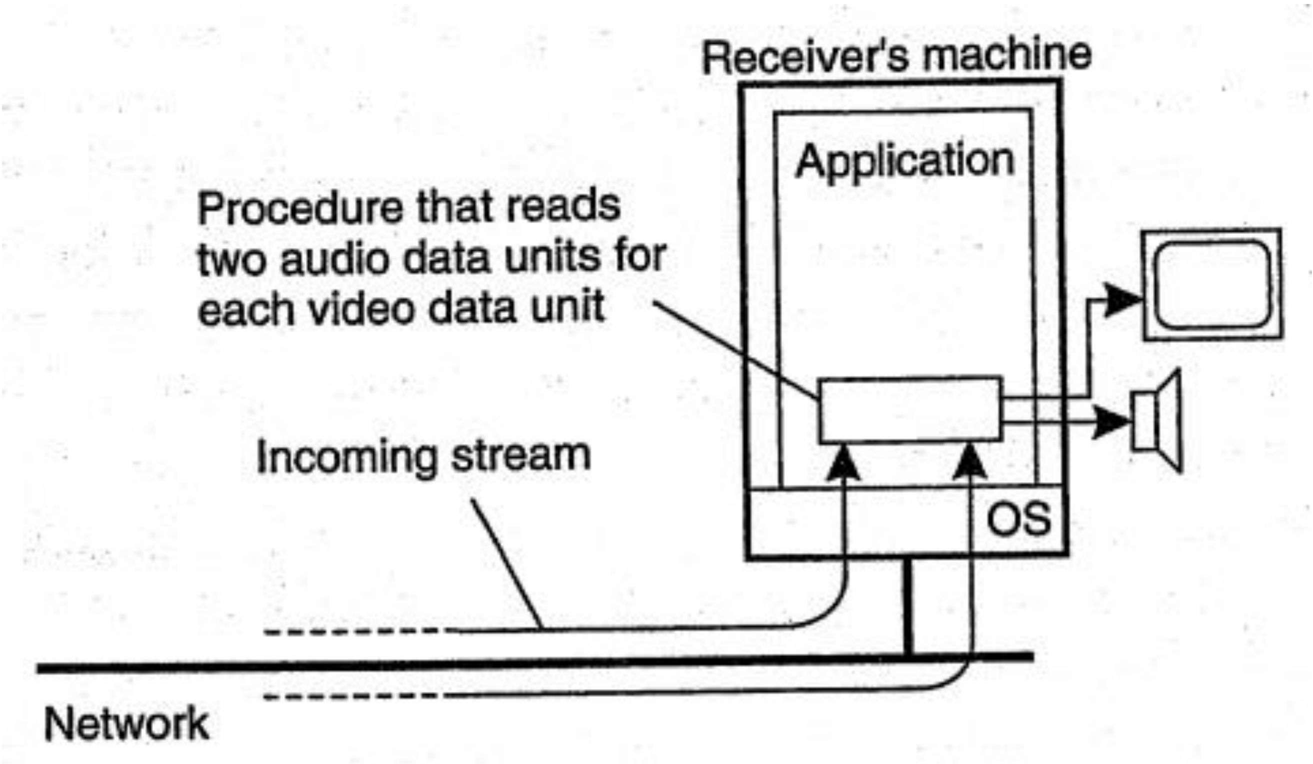
Labwork



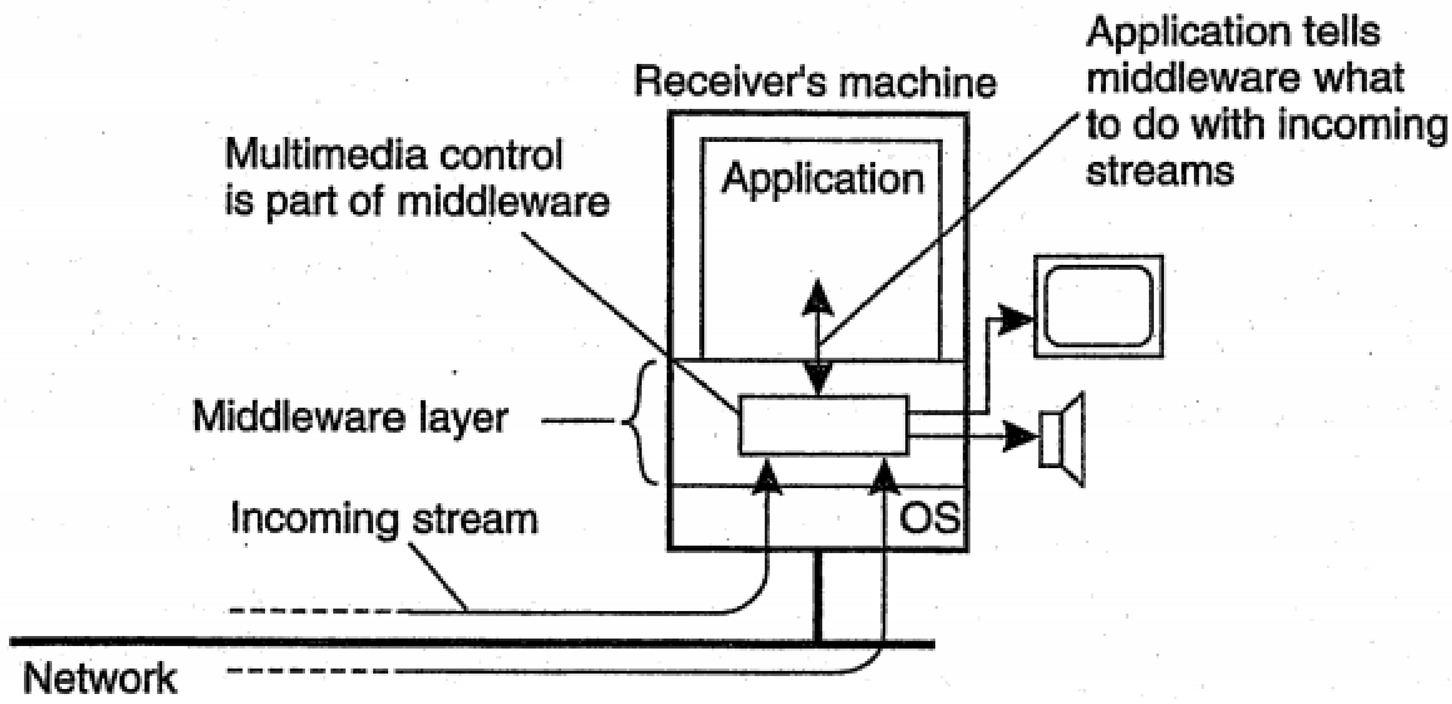
4.3. Stream Synchronization

- 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



VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Questions?