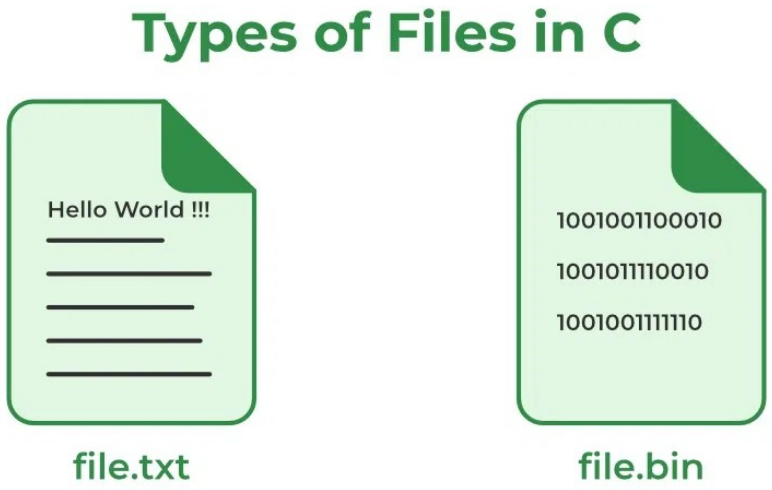
File Operation

Sujoy Saha
Department of CSE
NIT Durgapur

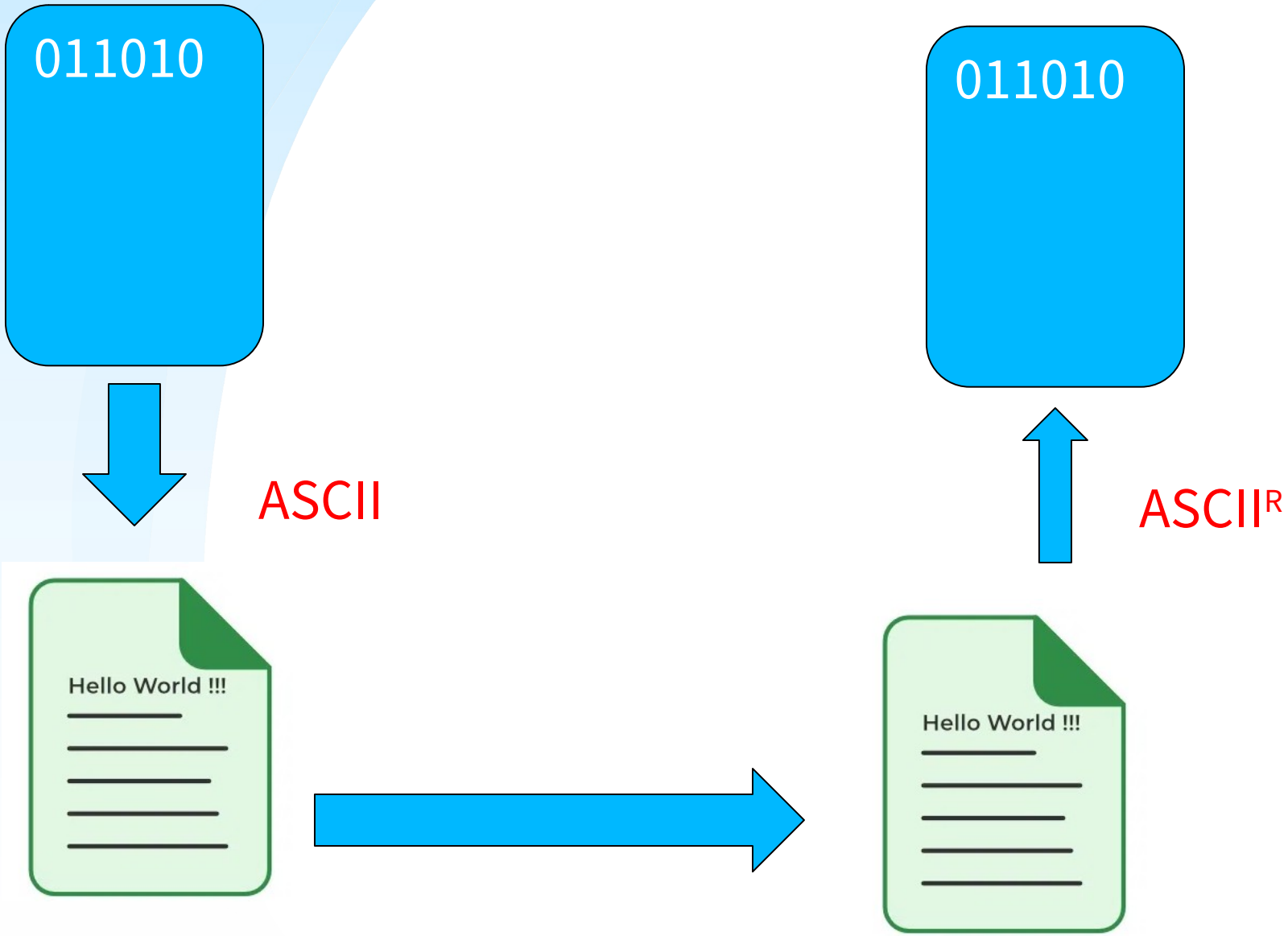
Types of Files in C



Text Files: A text file contains data in the **form of ASCII characters** and is generally used to store a stream of characters.

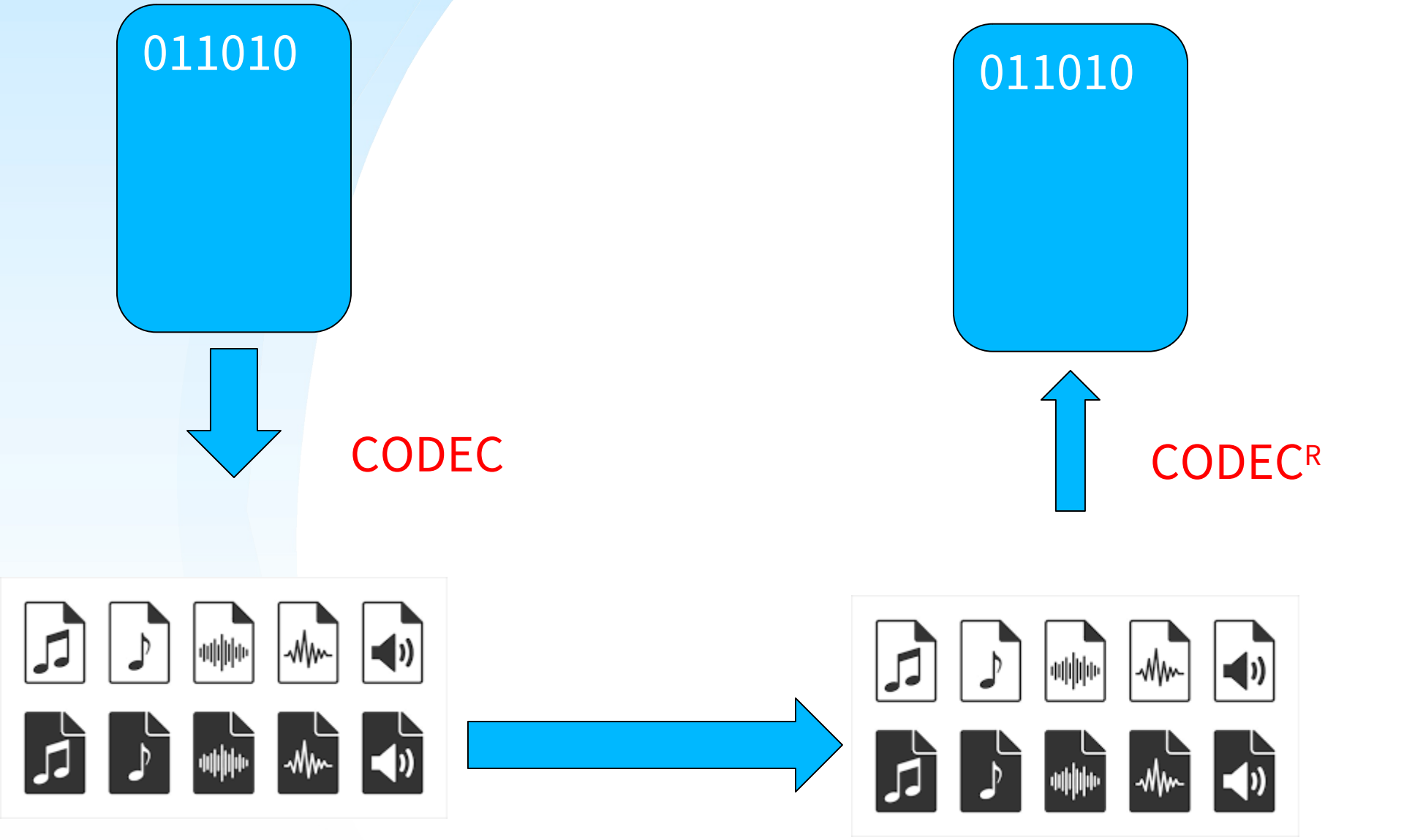
Binary Files: A binary file contains data in **binary form (i.e. 0's and 1's)** instead of ASCII characters. They contain data that is stored in a similar manner to how it is stored in the main memory.

Types of Files in C



When dealing with audio files ***(like .wav, .mp3, .flac, etc.)*** in C, the correct choice is almost always to use binary mode ***("rb", "wb", "ab", etc.)*** rather than ***text/ASCII mode ("r", "w", "a")***.

Types of Files in C



Why Binary Mode is Better (and Safer) for Audio/Video Files

- ***Audio files are binary by nature***
  - They don't store human-readable text, but raw byte sequences representing samples, headers, metadata, etc.
- the file is read and written **exactly as it is, byte by byte.**
  - This ensures that the ***data is preserved without any modification***, making it ideal for non-text files like images, videos, or executables.
- ***Text mode may corrupt the file***
  - In text mode, the runtime library may automatically translate certain characters (e.g., '\n' into "\r\n" on Windows).
  - That breaks the structure of audio files, since even one altered byte can make the file unreadable.
  - There is **no risk of accidental data corruption due to character encoding or newline conversions.**

Why Binary Mode is Better (and Safer) for Audio/Video Files

- ***Portability***
  - If you write in text mode, behavior may differ between platforms (Windows vs Linux vs macOS).
  - Binary mode guarantees consistent, byte-for-byte I/O across systems.

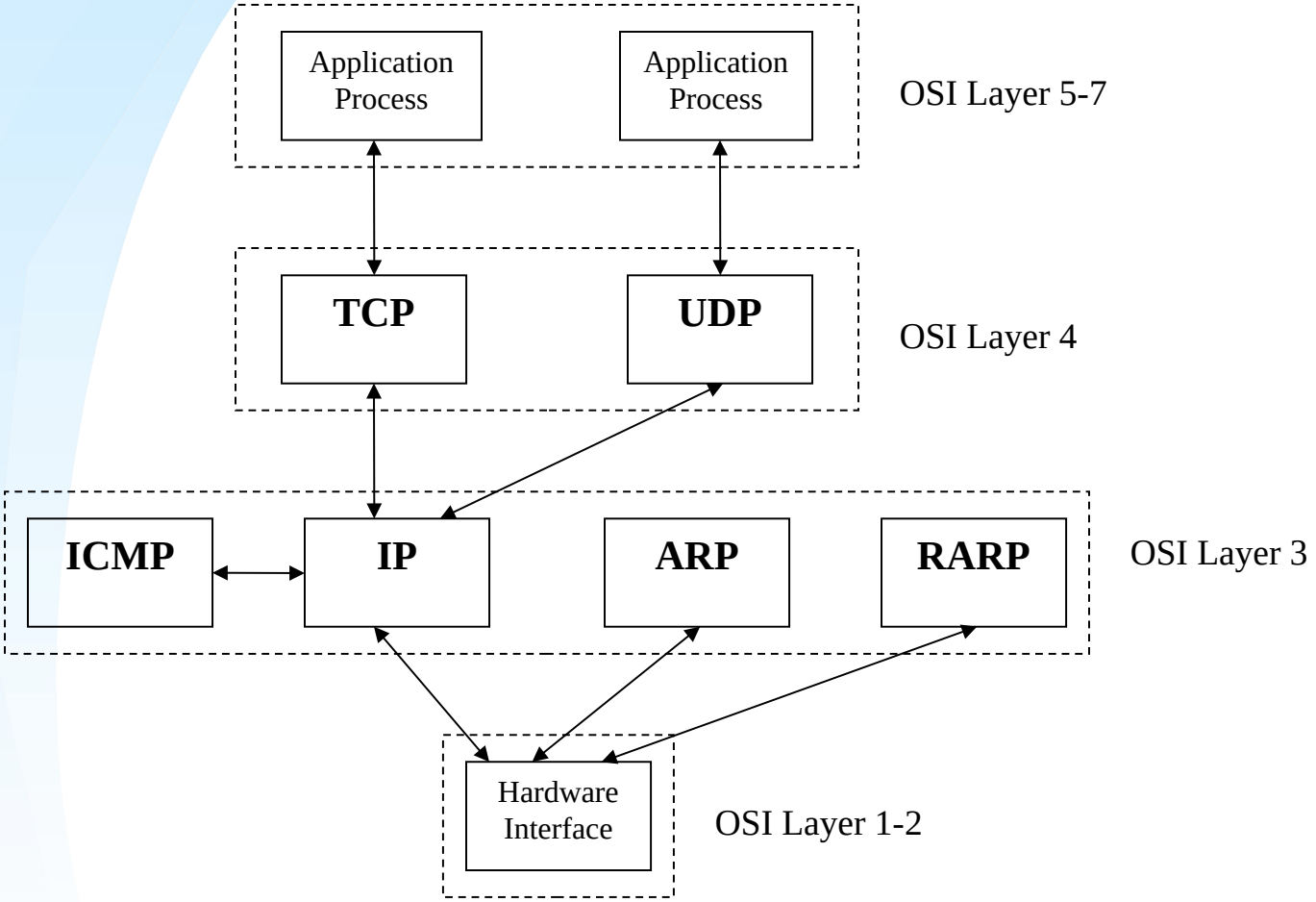
TCP/IP Socket Programming
in C

Sujoy Saha

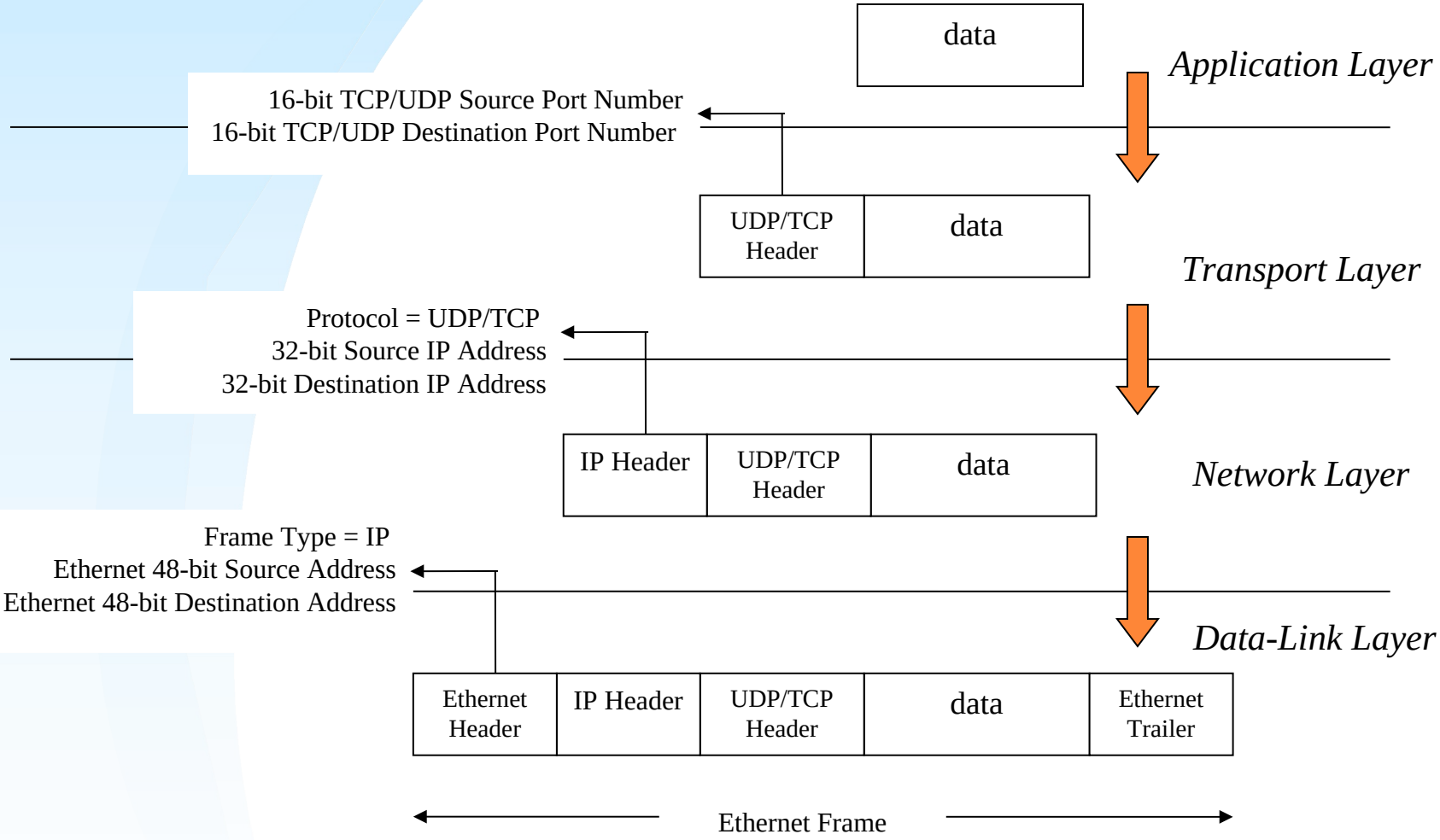
Department of CSE

NIT Durgapur

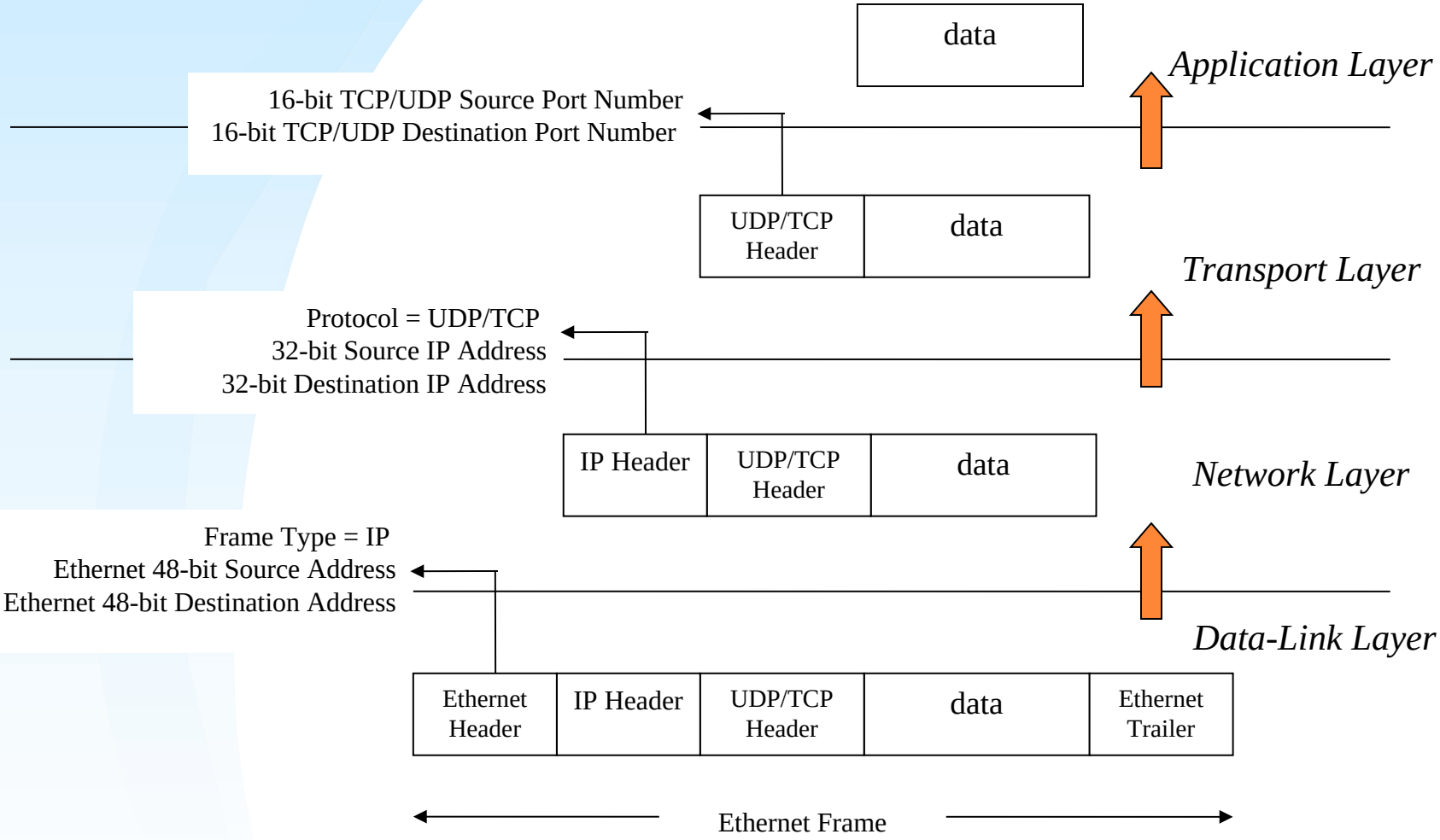
TCP/IP Protocol Suite



Sending Data

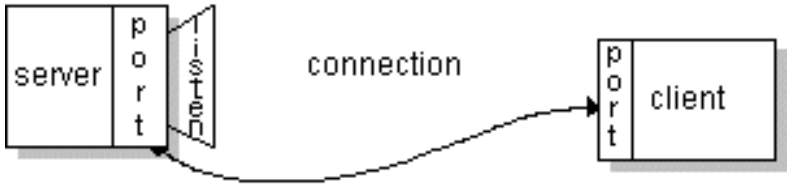


Receiving Data



Socket

"A **network socket** is an endpoint of an inter-process communication across a computer network."



Connection

- In TCP/IP protocol suite
  - a connection defines the communication link between two processes
  - An *association* defined by the 5-tuple completely specifies two processes that make up a connection

{protocol, local-addr,local-port, foreign-addr, foreign-port}

  - The protocol (TCP or UDP)
  - The local host's IP address (32-bit)
  - The local port number (16-bit)
  - The foreign host's IP address (32-bit)
  - The foreign port number (16-bit)
- Example

```
{tcp, 192.168.2.2, 1500, 192.168.2.10, 21}
```

Sockets

- Sockets are *application program interface (API)* to the communication protocol.
- Availability of an API depends both on the operating system and the programming language.

Socket Addresses

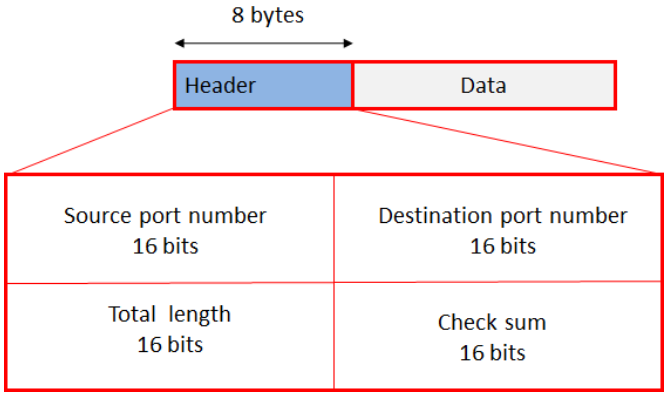
{tcp/UDP, 192.168.2.2, 1500, 192.168.2.10, 21}

- For Internet family, following structures are defined in

<netinet/in.h>

```
struct in_addr {
  u_long      s_addr;     /*32-bit netid/hostid*/
};

struct sockaddr_in {
  short       sin_family;/*AF_INET*/
  u_short     sin_port;  /*16-bit port number*/
  struct in_addr  sin_addr;  /*32-bit netid/hostid*/
  char    sin_zero[8]; /*unused*/
};
```



- The header file <sys/types.h> provides data type definitions

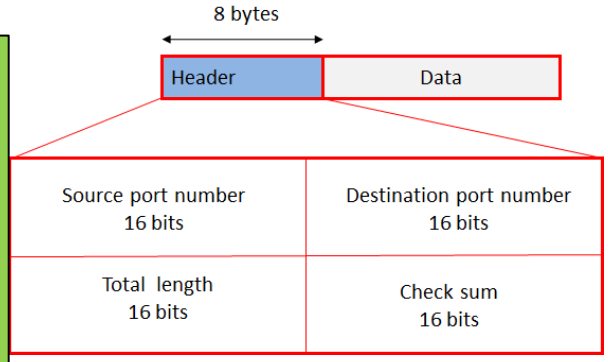
Socket Addresses

```
{tcp/UDP, 192.168.2.2, 1500,
  192.168.2.10, 21}
```

- For Internet family, following structures are defined in

In the socket API, the family (sometimes called domain) tells the operating system *what kind of addresses the socket will use*.

- AF_INET → IPv4 addresses (e.g., 192.168.1.1)
- AF_INET6 → IPv6 addresses (e.g., 2001:db8::1)
- AF_UNIX / AF_LOCAL → Local (Unix domain) sockets for communication on the same machine – Not use IP Address
  - it uses a file ***path*** in the filesystem (e.g., /tmp/mysocket) to identify the communication endpoint.

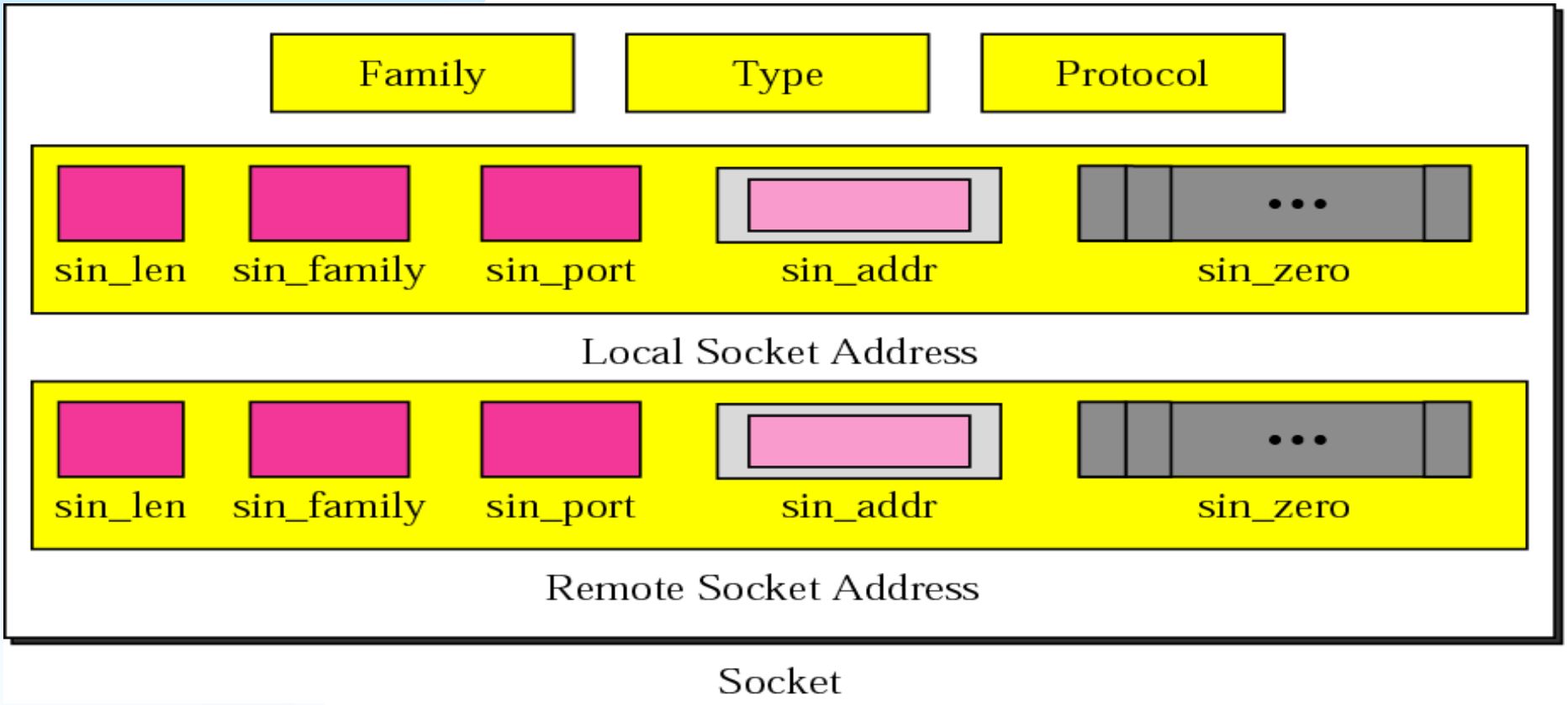


tid*/

data type

Figure 16-4

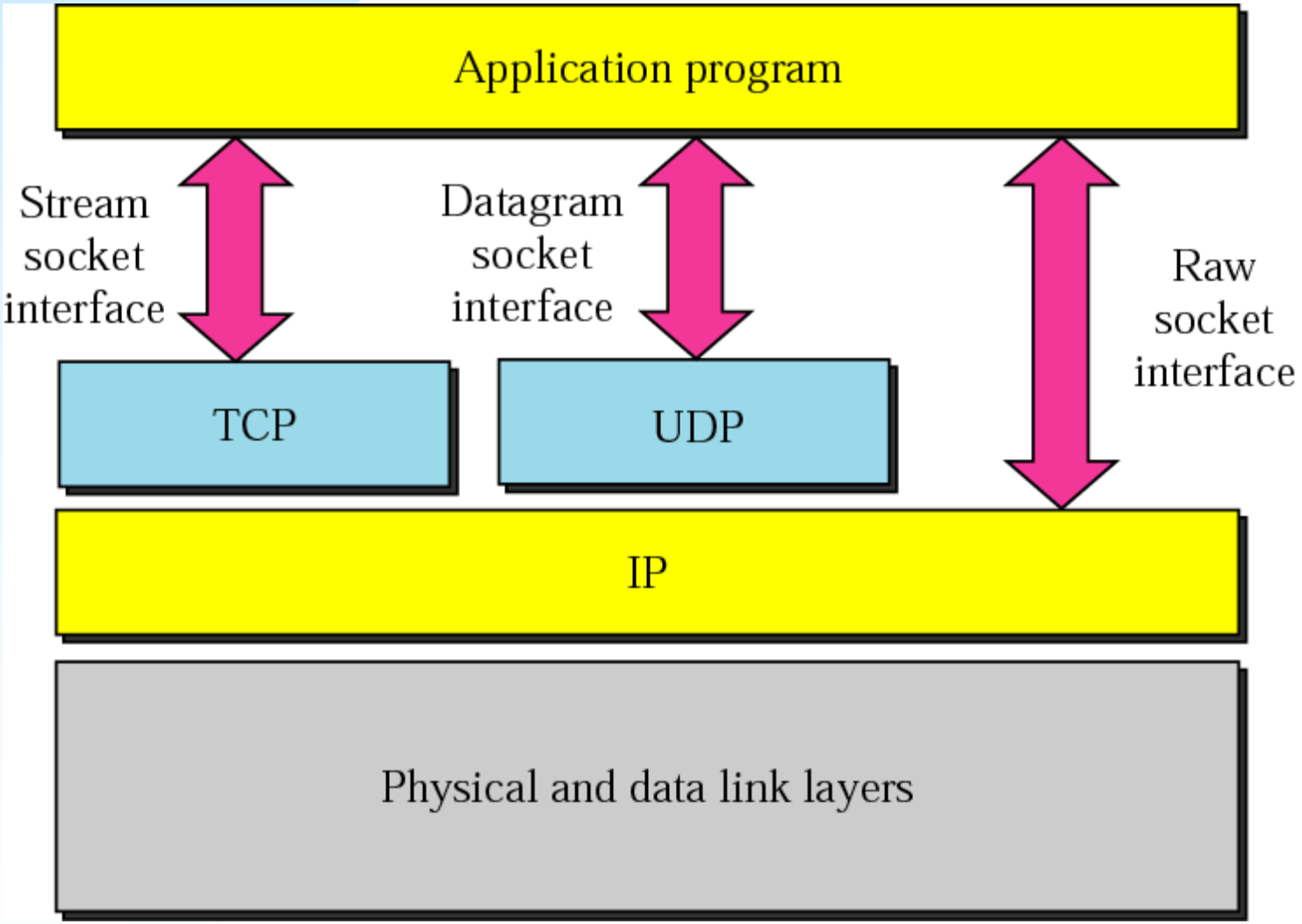
Socket structure



Communication domain in which the socket should be created. Some of address families are AF_INET (IP), AF_INET6 (IPv6), AF_UNIX (local channel, similar to pipes), AF_ISO (ISO protocols), and AF_NS (Xerox Network Systems protocols).

Figure 16-5

Socket types



BYTE ORDERING

Figure 16-6

Big-endian byte order

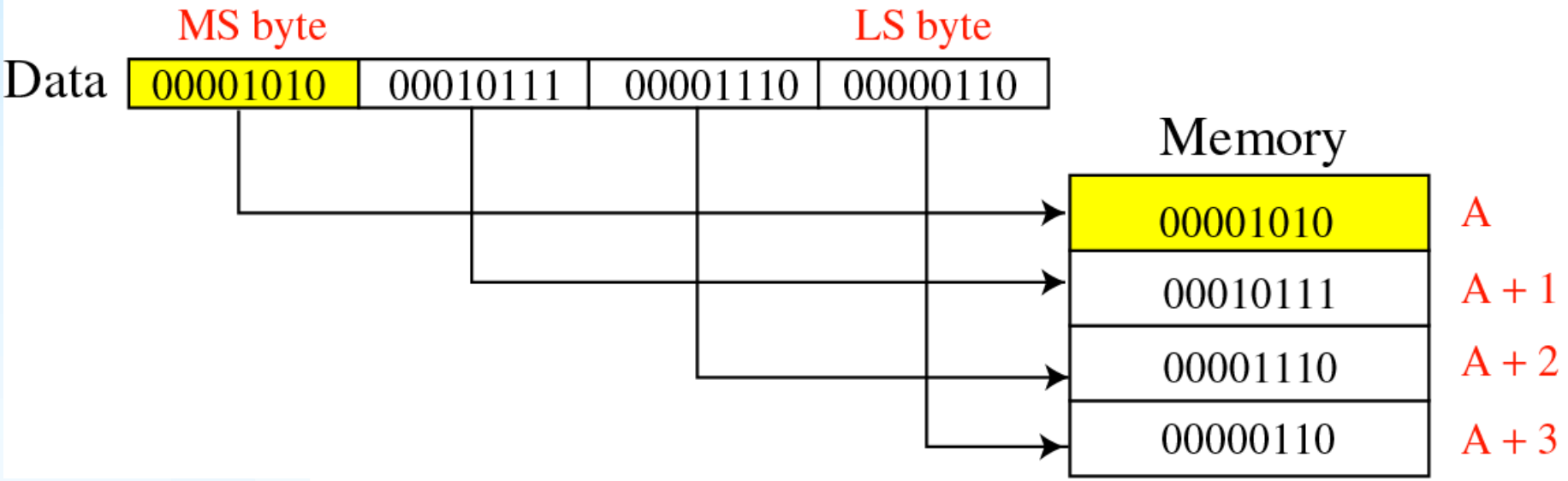
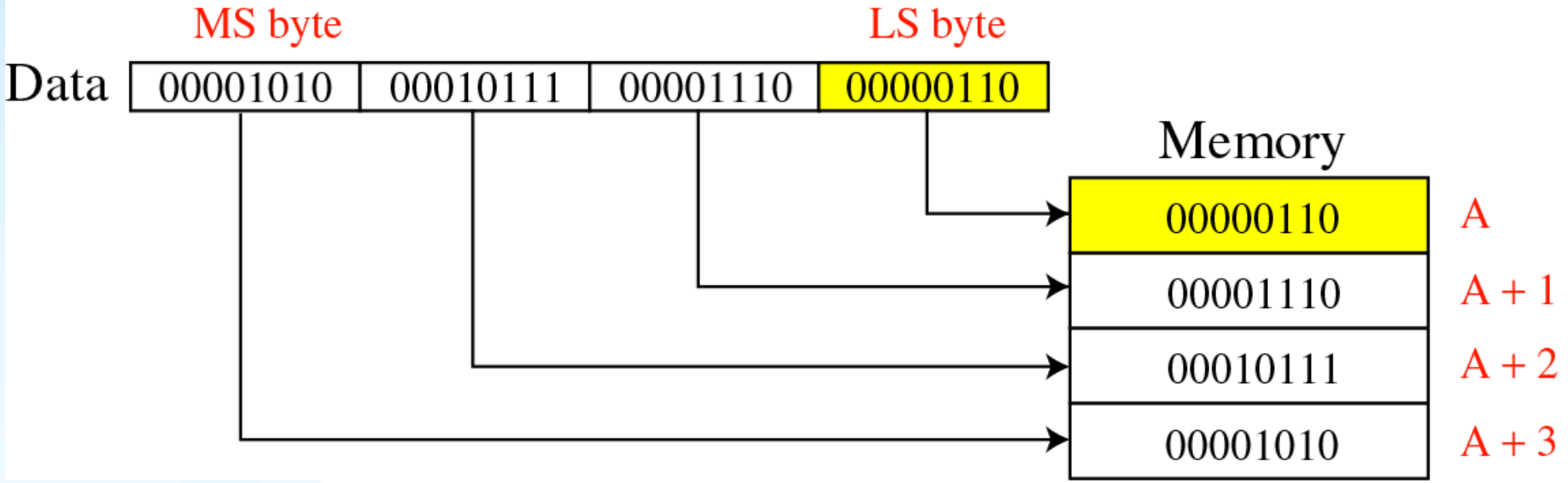


Figure 16-7

Little-endian byte order



Note

The byte order for the TCP/IP protocol suite is big endian.

Byte Ordering Functions

- Hosts can store multi-byte values differently (*host byte order)*
  - Little-endian byte order
  - Big-endian byte order
- Network protocols must specify a *network byte order*
  - Internet protocols use big-endian byte ordering
- Byte ordering routines

```
u_long htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

u_long: A data type that can store **non-negative integers**. The size is usually **4 (32 bit machine) or 8 bytes (64 bit machine)**, depending on the system.

Often defined as a shorthand for **unsigned long**, usually in **system headers or legacy code**.

It's not part of the standard C library but may appear in codebases or system-specific headers.

Figure 16-8

Bite-order transformation

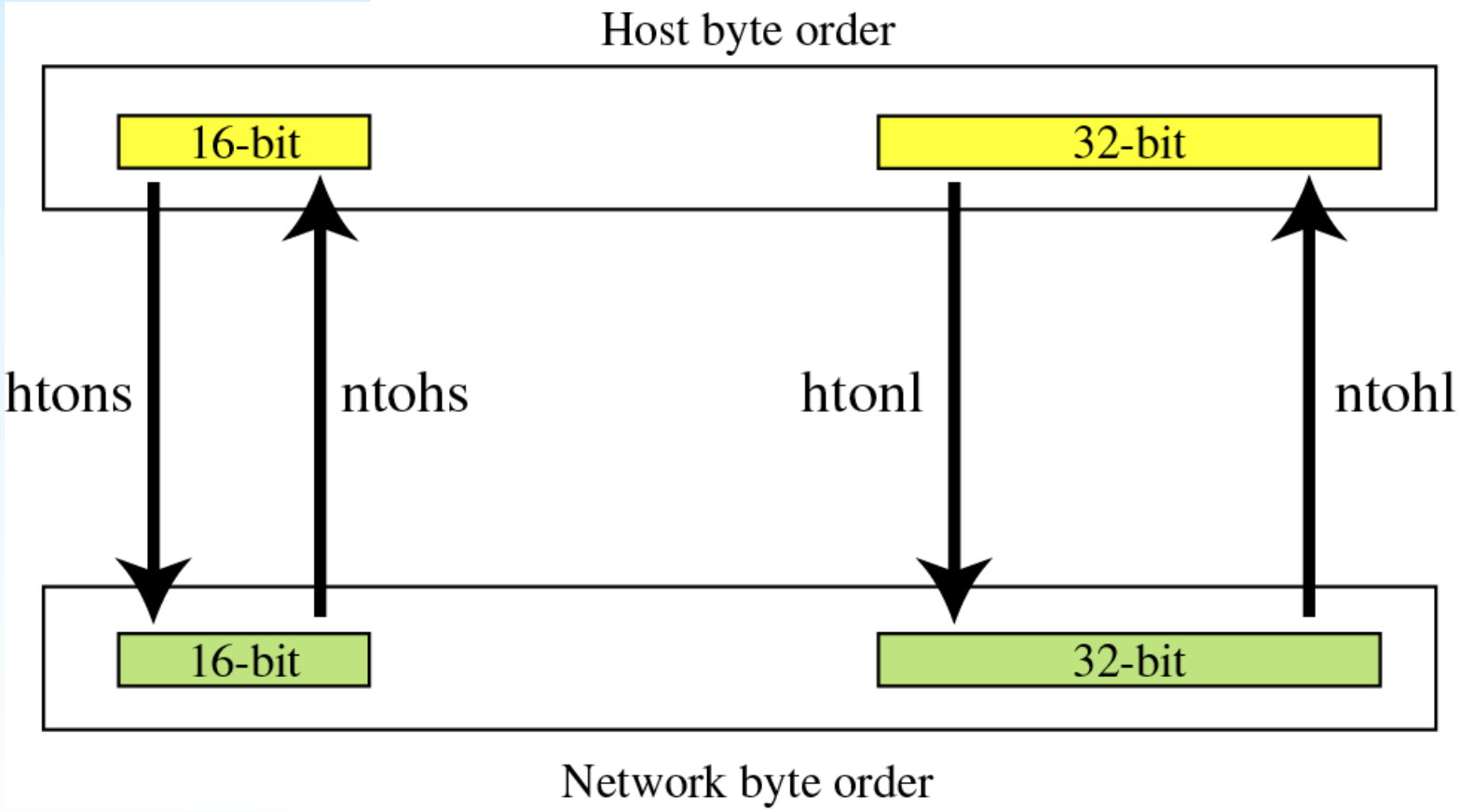


Figure 16-1

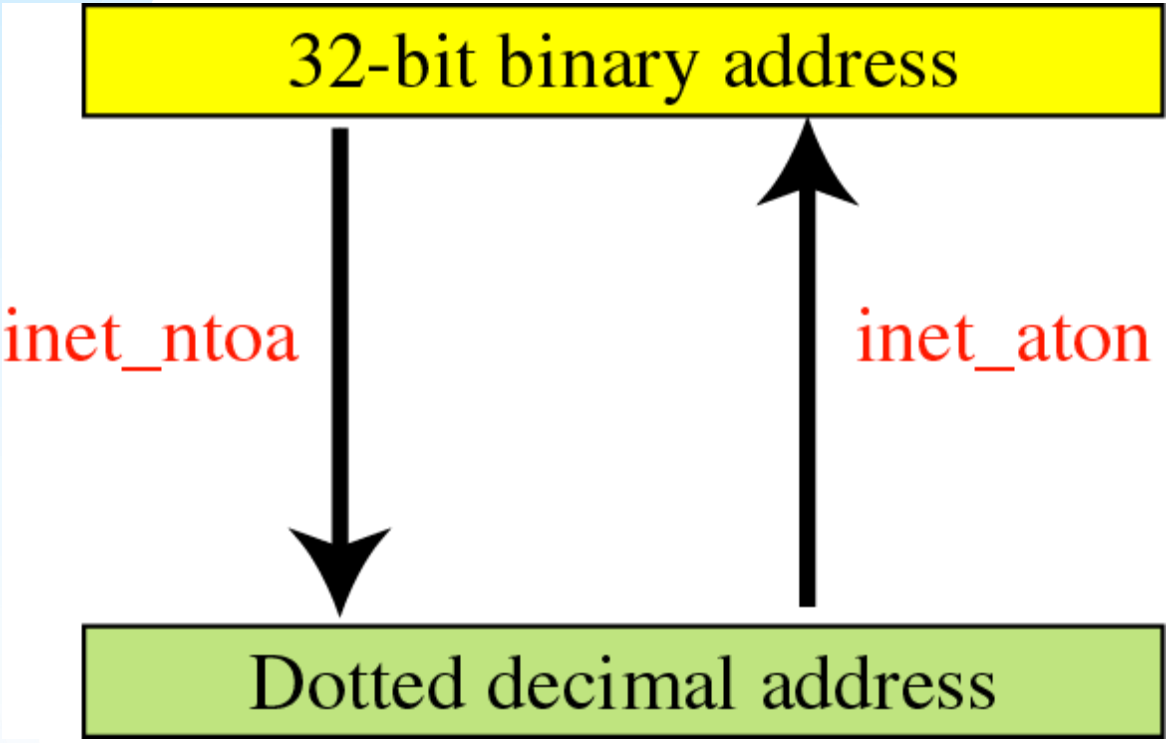
Data types

#include<stdint.h>

int8_t	Signed 8-bit integer	-128 to 127 [7 bit → number and 1 bit for sign]
int16_t	Signed 16-bit integer	
int32_t	Signed 32-bit integer	
uint8_t	Unsigned 8-bit integer	Only positive numbers→0-255[utilizing all 8 bits]
uint16_t	Unsigned 16-bit integer	
uint32_t	Unsigned 32-bit integer	

Figure 16-10

Address transformation



The inet_ntoa function in C is used to convert an **IP address in a numerical format** (stored as a struct in_addr) into a **human-readable dotted-decimal string** (e.g., 192.168.1.1).

Address Conversion Routines

```
unsigned long inet_addr(char *ptr);
int inet_aton(char *ptr, struct in_addr *addrptr);
```

Converts a character string in dotted-decimal notation to a 32-bit Internet address in network byte order

```
char *inet_ntoa(struct in_addr inaddr);
```

Converts a 32-bit Internet address in network byte order to a character string in dotted decimal notation

BYTE MANIPULATION FUNCTIONS

Declarations for byte-manipulation functions

```
void      *memset  ( void *dest , int  chr , size_t len ) ;

void      *memcpy  ( void *dest , const void  *src , size_t len ) ;

int        memcmp  ( const void *first , const void  *second , size_t len ) ;
```

```
char buffer[1400];

memset(buffer,0,sizeof(buffer));

bzero(buffer, sizeof(buf));
```

```
struct MSG
{
int m_txt[MAX_TEXT];
int m_seq;
int
total_no_of_packets;
};
```

```
memcpy(buffer, msg.m_txt, sizeof(msg.m_txt));
```

SOCKET SYSTEM CALLS

`socket` System Call

```
int socket(int family, int type, int protocol);
```

- The family is one of

`AF_UNIX`	Unix internal protocols
`AF_INET`	Internet Protocols
`AF_NS`	Xerox NS Protocols
`AF_IMPLINK`	IMP link Layer

- AF_INET uses the TCP/IP protocol.
- AF_UNIX creates filesystem objects and it only works between processes on the same host.
- AF_NS is a set of protocols that were used by Xerox Systems for data communication. Its basic working mechanism is almost the same as in the TCP/IP protocol suit, but **XNS contains only two network layers**

AF_INET and AF_UNIX

- AF_INET uses the TCP/IP protocol.

- AF_UNIX creates filesystem objects and it only works between processes on the same host.
- AF_UNIX is much faster than AF_INET.

- Xerox Network Systems (XNS) is a set of protocols that were used by Xerox Systems for data communication. Xerox used **XNS for file transfers, sharing network resources, packet transfers, sharing routing information and remote procedure calls**. Its basic working mechanism is almost the same as in the TCP/IP protocol suit, but **XNS contains only two network layers**. This differs from the seven-layer Open Systems Interconnection (OSI) model, although the functionality is basically the same.

`socket` System Call

```
int socket(int family, int type, int protocol);
```

- The family is one of

`AF_UNIX`	Unix internal protocols
`AF_INET`	Internet Protocols
`AF_NS`	Xerox NS Protocols
`AF_IMPLINK`	IMP link Layer

- The socket type is one of the following

`SOCK_STREAM`	Stream Socket
`SOCK_DGRAM`	Datagram Socket
`SOCK_RAW`	Raw Sockets
`SOCK_SEQPACKET`	Sequenced Packet Socket
`SOCK_RDM`	Reliably Delivered Message Socket

- The protocol argument to the socket system call is typically set to 0
- The `socket` system call returns an integer value called socket descriptor
- This number is passed as a parameter to almost all of the other library calls.

Figure 16-15

Declaration for *socket* function

```
int socket (int family , int type , int protocol) ;
```

AF_INET SOCK_DGRAM SOCK_STREAM 0

Returns a socket descriptor if successful; -1 if error.

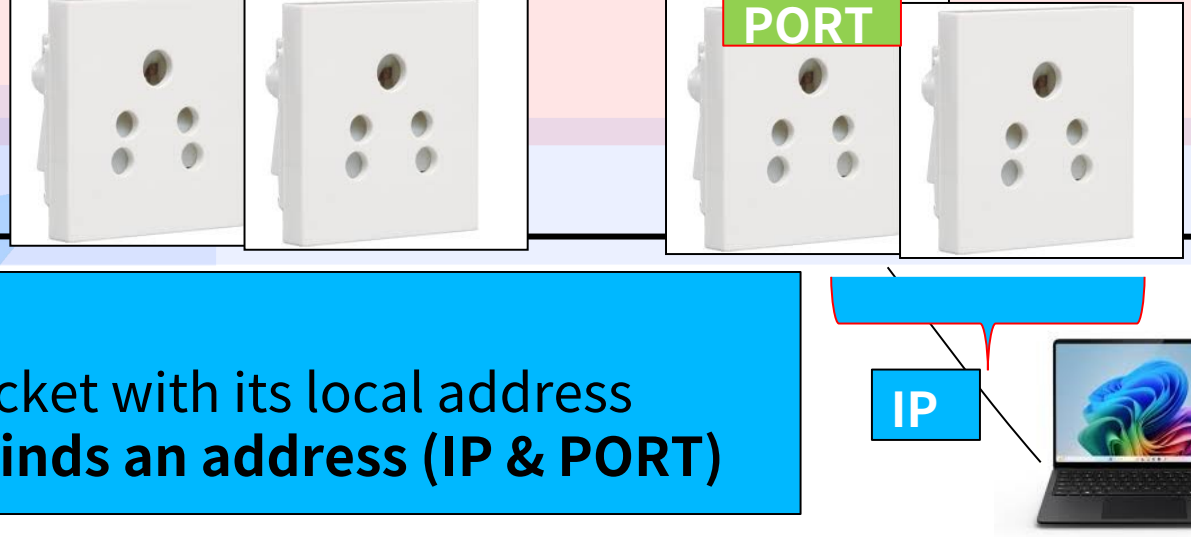
- The socket( ) system call returns a file descriptor.
- In fact, a socket is similar to an opened file because it is possible to read and write data on it by means of the usual read( ) and write( ) system calls.
- in the context of the client-server model, the socket system calls needed to create and connect a pair of sockets and transmit data.

type \ *family*	AF_INET
SOCK_STREAM	**TCP** or SCTP
SOCK_DGRAM	**UDP**
SOCK_SEQPACKET	SCTP
SOCK_RAW	IPv4

Protocol
IPPROTO_TCP
IPPROTO_UDP
IPPROTO_SCTP

Family	Description
AF_UNIX	UNIX internal (file system sockets)
AF_INET	ARPA Internet protocols (UNIX network sockets)
AF_ISO	ISO standard protocols
AF_NS	Xerox Network Systems protocols
AF_IPX	Novell IPX protocol
AF_APPLETALK	Appletalk DDS

bind



bind system call:

- bind() associates the socket with its local address
- The bind() system call **binds an address (IP & PORT)**

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

Returns: 0 if OK, -1 on error

sockaddr is a generic structure, bind system call uses typepunning

- e.g. bind()

```
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl( INADDR_ANY );
myaddr.sin_port = htons( 7 );// 7 for echo service

if( bind( sockfd, (struct sockaddr *)&myaddr,
sizeof(myaddr) ) < 0 ) { ... }
```

`bind` System Call

- Uses of `bind` system call
  - Servers register their well-known address with the system; both connection-oriented and connection-less servers need to do this
  - A client can register a specific address for itself
- The `bind` system call specifies the *local-addr* and *local-port* elements of the 5-tuple of an association

```
struct sockaddr_in {
  short     sin_family;  /*AF_INET*/
  u_short       sin_port;    /*16-bit port number*/
  struct in_addr    sin_addr;    /*32-bit netid/hostid*/
  char sin_zero[8]; /*unused*/
};
```

```
Struct sockaddr
{
 unsigned short sa_family;
 Char sa_data[14]
}
```

sin_port (2 Byte) + Sin_addr(4 Byte) + sin_Zero (8Byte) = 14 byte

Why is a bind system call required in socket programming?

- Otherwise if a port is not binded to any socket descriptor, we can use the same port and IP combination again to connect to a different destination.

- bind() assigns the address specified to by addr to the socket referred to by the file descriptor sockfd.

- bind() associates the socket with its local address [that's why server side binds, so that clients can use that address to connect to server.]

sendto and recvfrom Functions

```
int sendto(int sockfd, char *buff, int length, int flags,
  struct sockaddr *to, socklen_t addrlen);
```

to argument is a socket address structure containing protocol address of where the data is to be sent

sockfd -- The socket descriptor.

buff --The pointer to the buffer containing the message to transmit.

length -- The length of the message in the buffer pointed to by the msg parameter.

Flags-- **MSG_DONTROUTE --** Don't use a gateway to send out the packet, only send to hosts on directly connected networks. This is usually used only by diagnostic or routing programs ;

MSG_DONTWAIT - Enables nonblocking operation;

MSG_WAITALL requests blocking until the entire number of bytes requested can be read.

to --The address of the target.

addr_len --The size of the address pointed to by address.

`sendto` and `recvfrom` Functions

- A blocking system call is one that must wait until the action can be completed. read() would be a good example - if no input is ready, it'll sit there and wait until some is (provided you haven't set it to non-blocking, of course, in which case it wouldn't be a blocking system call)

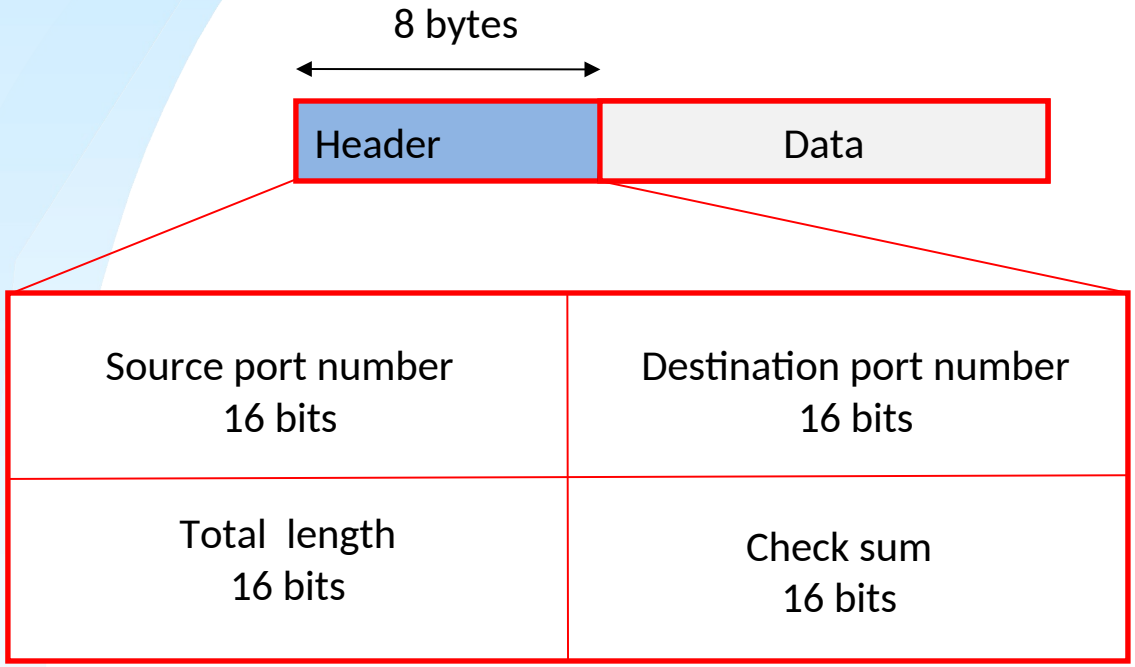
- If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, sendto() blocks the caller until additional buffer space becomes available.

`sendto` and `recvfrom` Functions

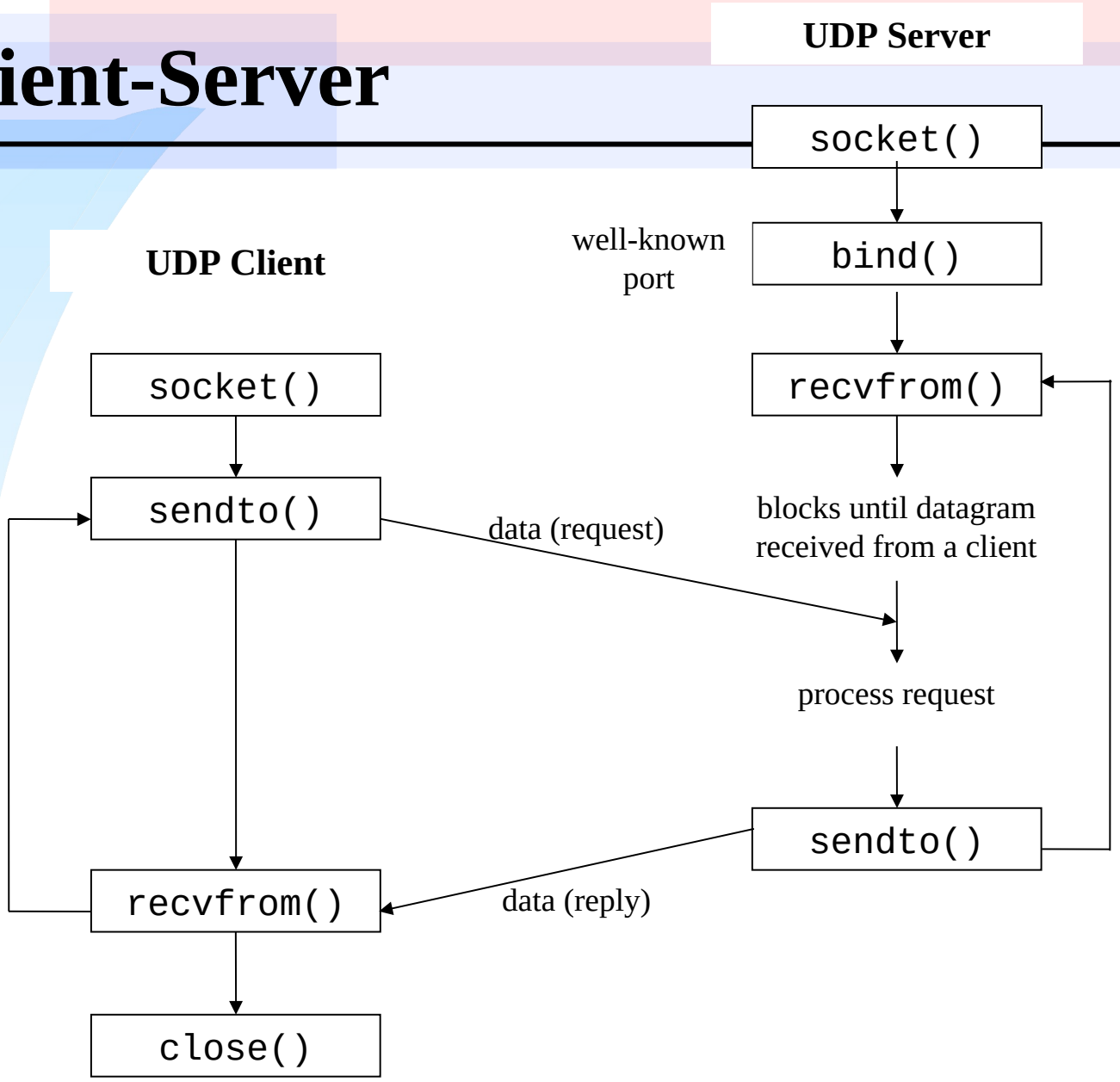
```
int recvfrom(int sockfd, char *buff, int nbytes, int
flags, struct sockaddr *from, socklen_t *addrlen);
```

- Fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram
- If no incoming data is available at the socket, the recvfrom function blocks and waits for data to arrive according to the blocking rules

- *User datagram format :*



UDP Client-Server



Header Files

- #include <sys/socket.h> // Core BSD socket functions and data structures.
- #include <netinet/in.h> // AF_INET and AF_INET6 address families and their corresponding protocol families PF_INET and PF_INET6.
- #include <arpa/inet.h> // Functions for manipulating numeric IP addresses.
- #include <netdb.h> // Name resolution

Example (UDP_CLIENT)

- struct MSG

{

int m_txt[MAX_TEXT];

int m_seq;

int total_no_of_packets;

};

- struct MSG msg;

- struct sockaddr_in sa;

- sock1 = socket(AF_INET,SOCK_DGRAM,0);

- bytes_sent = sendto(sock1, (void*)&msg, sizeof(struct MSG),0,(struct sockaddr*) &sa, sizeof(struct sockaddr_in));

Example (UDP_SERVER)

- struct MSG

{

int m_txt[MAX_TEXT];

int m_seq;

int total_no_of_packets;

};

- struct MSG msg;
- int s_length;
- struct sockaddr_in sa,cli;

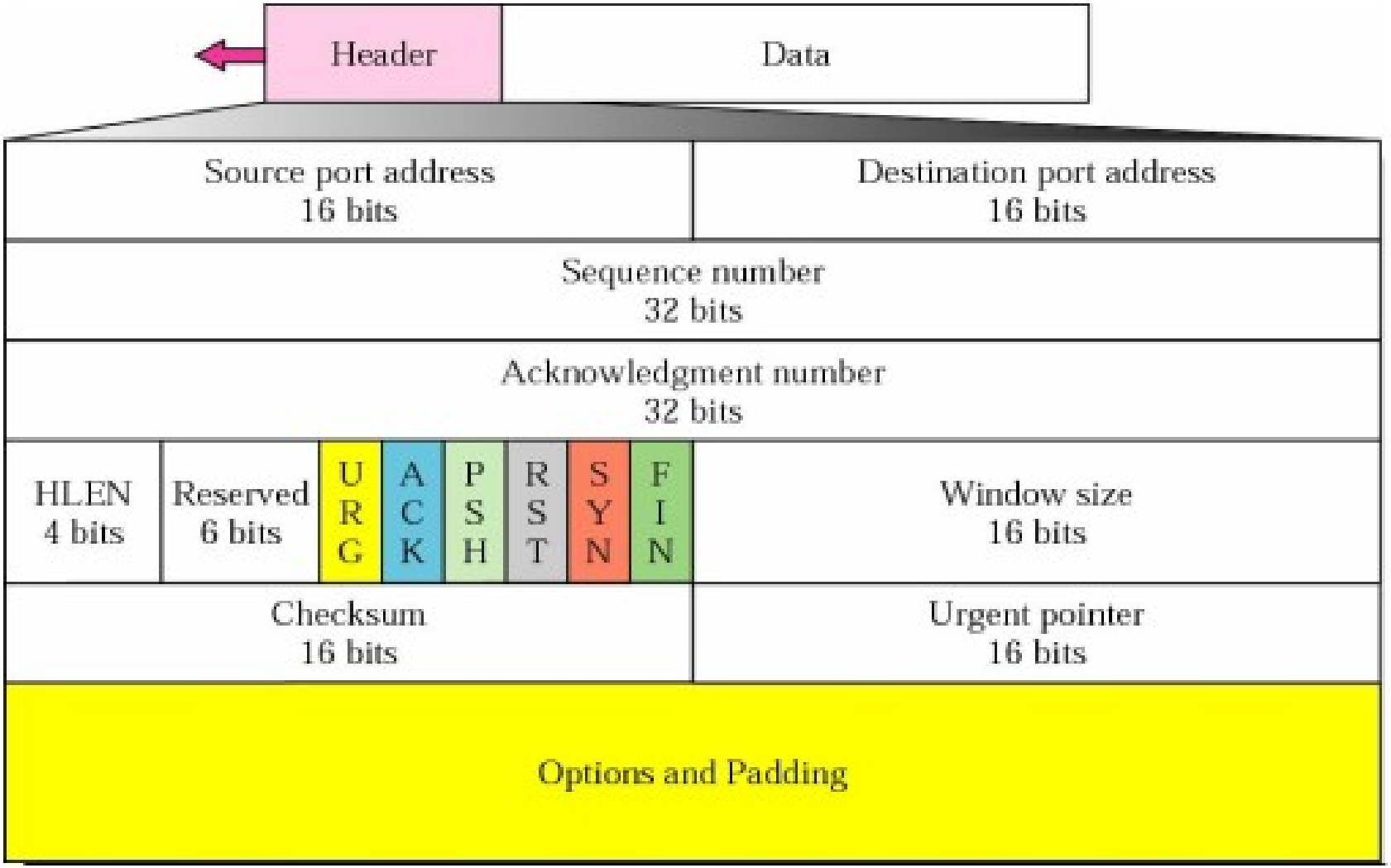
- sock1 = socket(AF_INET,SOCK_DGRAM,0);

Example UDP_SERVER

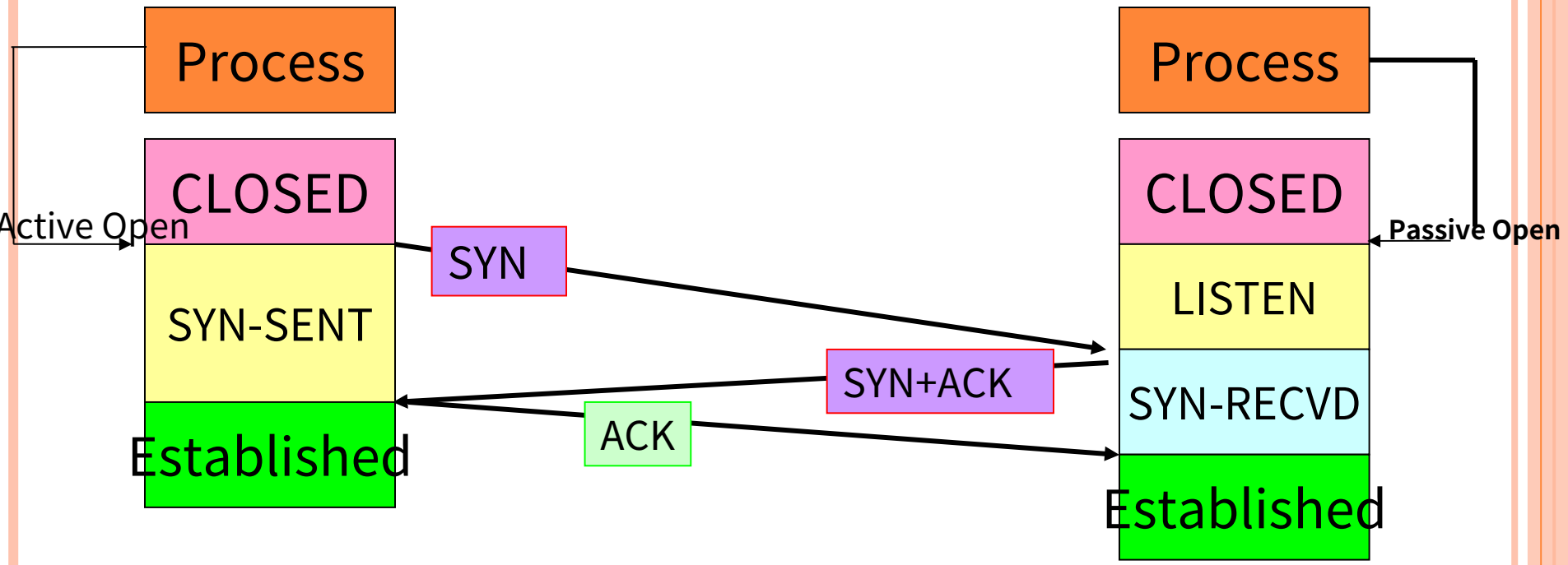
- sa.sin_family = AF_INET;
- sa.sin_port = htons(PORT);
- sa.sin_addr.s_addr = htonl(INADDR_ANY);
- s_length=sizeof(sa);
- bind(sock1,(struct sockaddr *)&sa, s_length);
- rec=recvfrom(sock1,(void*)&msg,sizeof(struct MSG),0,(struct sockaddr *) &cli,sizeof(cli));

TCP Protocol

TCP Segment Format



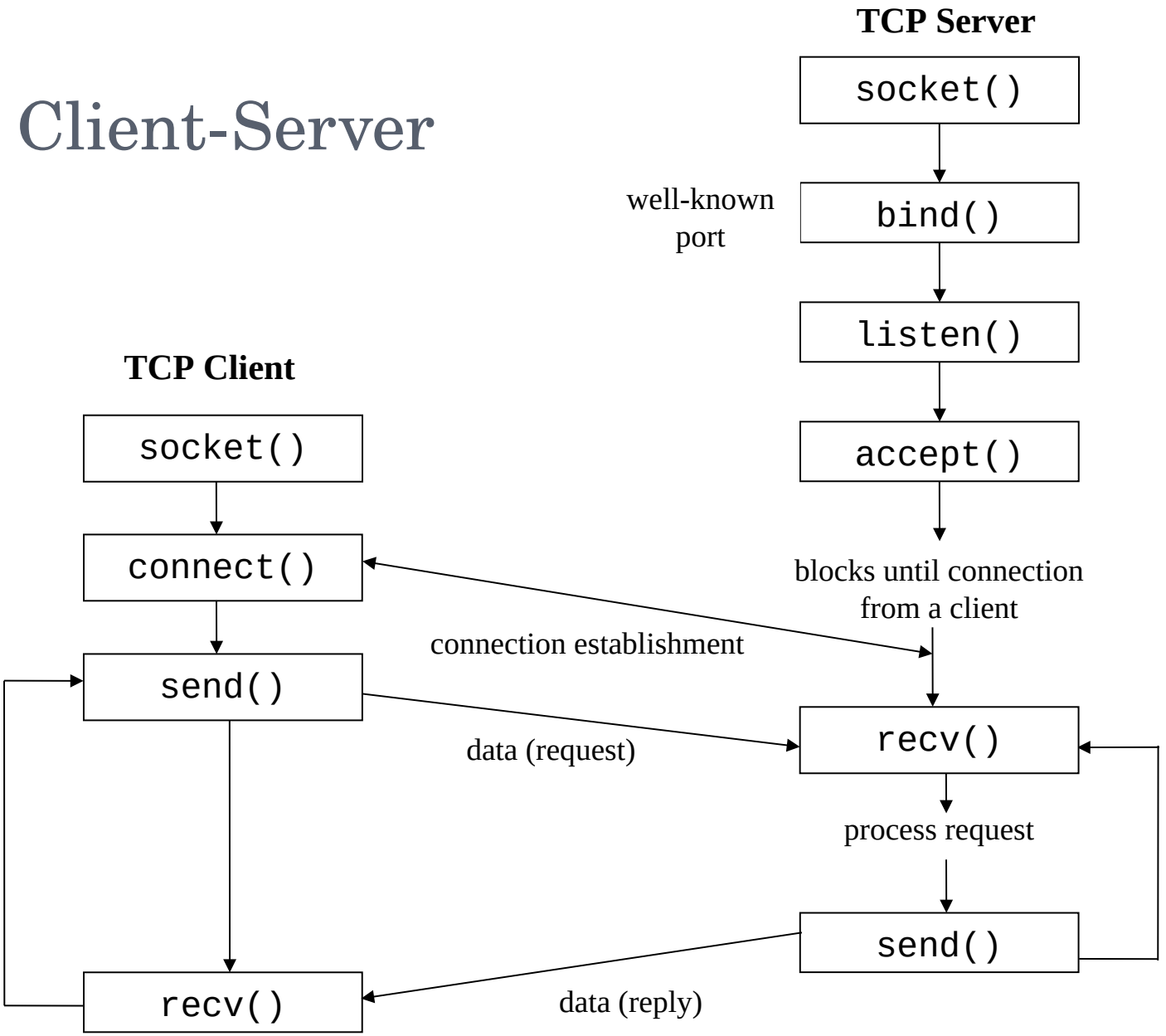
THREE-WAY HANDSHAKING



Passive Open: The application on the server is passive.
The application is listening, awaiting a connection

The application on the client makes a connection request to the server where the application is passive open.
The application on the client is said to be **"active open".**

TCP Client-Server



`connect` System Call

```
int connect(int sockfd, struct sockaddr *servaddr,
            int addrlen);
```

- For connection oriented protocols the `connect` system call results in the actual establishment of a connection between the local and foreign system
- *`sockfd`* is a socket descriptor that was returned by the `socket` system call
- *`servaddr`* is a pointer to an address structure containing the address of the server
- *`addrlen`* is the length of the address structure

connect()

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

Returns: 0 if OK, -1 on error

- e.g. connect()
  - struct sockaddr_in servaddr;

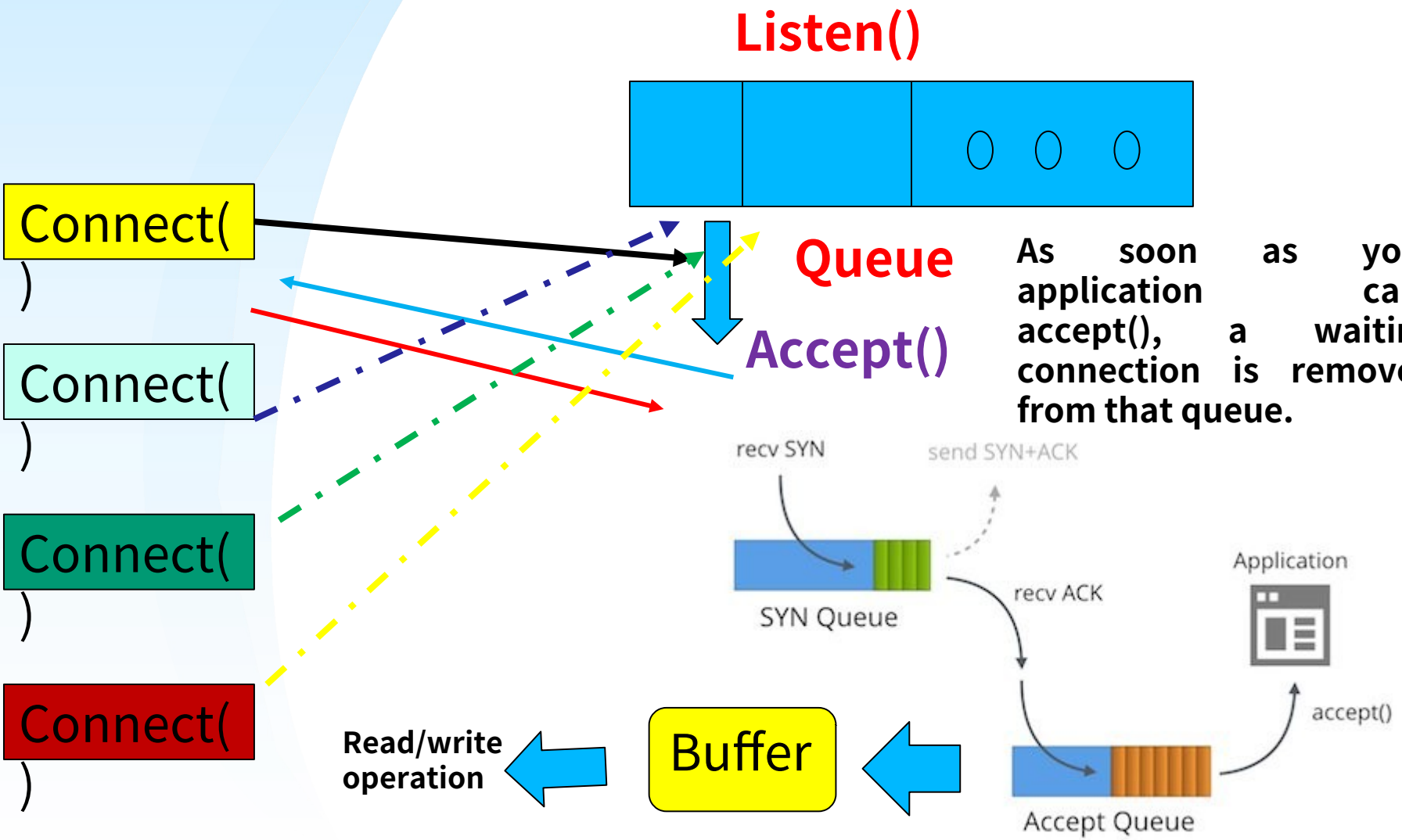
```
...
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl( “220.95.133.100” );
servaddr.sin_port = htons( 7 );   // 7 for echo service

if( connect( sockfd, (struct sockaddr *)&servaddr,
sizeof(servaddr) ) < 0 )
{  ... }
...
```

`listen` System Call

```
int listen(int sockfd, int backlog);
```

- This system call is used by a connection-oriented server to indicate that it is willing to receive connection
- Called after both the `socket` and `bind` system calls, and immediately before the `accept` system call
- The *backlog* argument specifies **how many connection requests can be queued by the system** while it waits for the server to execute the `accept` system call
- That number is only the **size of the connection queue**, where new connections wait for somebody to accept them. As soon as your application calls accept(), a waiting connection is removed from that queue. So, you can definitely handle more than 128 simultaneous connections because they usually only spend a short time in the queue


Listen()
Connect(
)
Connect(
)
Connect(
)
Connect(
)
Queue
Accept()
As soon as your application calls accept(), a waiting connection is removed from that queue.
recv SYN
send SYN+ACK
SYN Queue
recv ACK
Application
accept()
Accept Queue
Buffer
Read/write operation

`accept` System Call

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

- Takes first connection request on the queue and creates another socket with the same property as *`sockfd`*
- Blocks the caller if no connection request is pending
- *`peer`* argument return the address of the connected peer client
- *`addrlen`* returns the length of the client address structure

listen() and accept()

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

Returns: 0 if OK, -1 on error

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

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

- e.g. listen()
  - `#define LISTENQ 5`

```
...
if( listen( mysock, LISTENQ ) < 0 ) { ... }

while( 1 ){
  clilen = sizeof( cliaddr );
  if( ( clisockfd = accept( mysock, (struct sockaddr *)&cliaddr,
&clilen ) ) < 0 ) { ... }
  ...
}
```

recv() and send()

```
#include <sys/types.h>
#include <sys/socket.h>
int recv( int sockfd, void * msg, size_t len, int flags );
int send( int sockfd, const void * msg, size_t len, int flags );
```

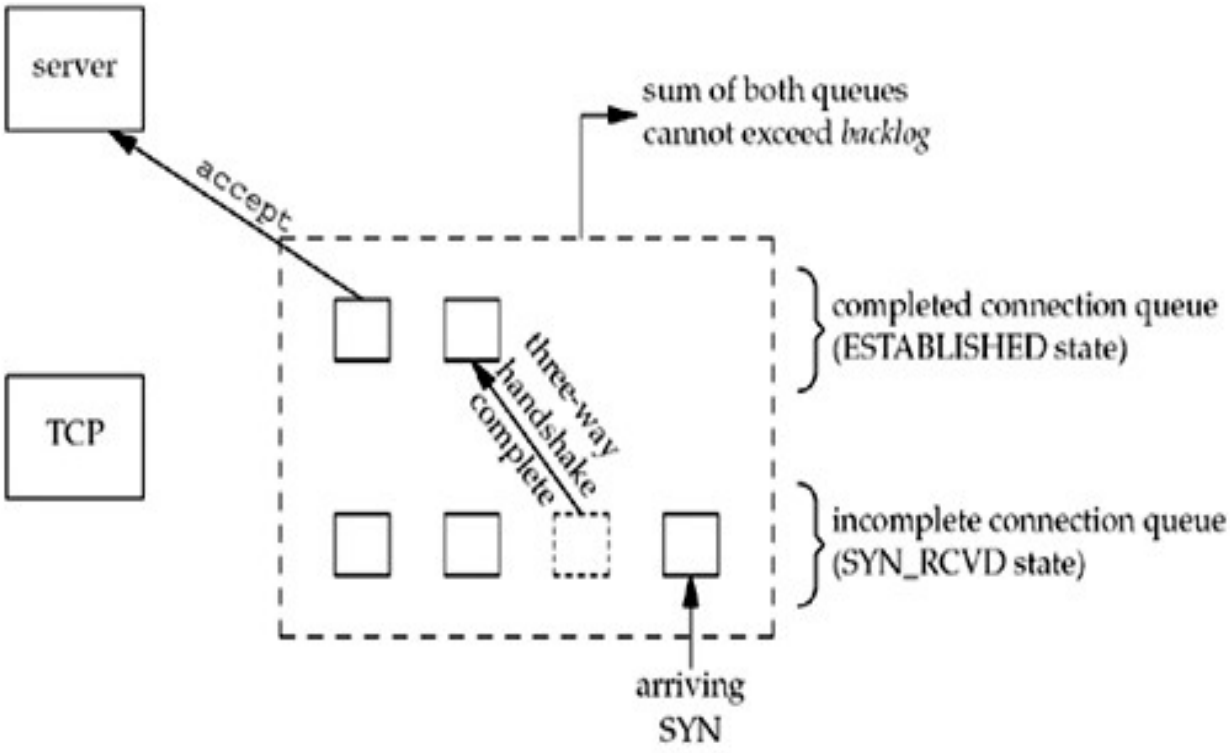
Both return: number of bytes read or written if OK, -1 on error

msg: message you want to send
len: length of the message
flags := 0
returned: the number of bytes actually sent or received

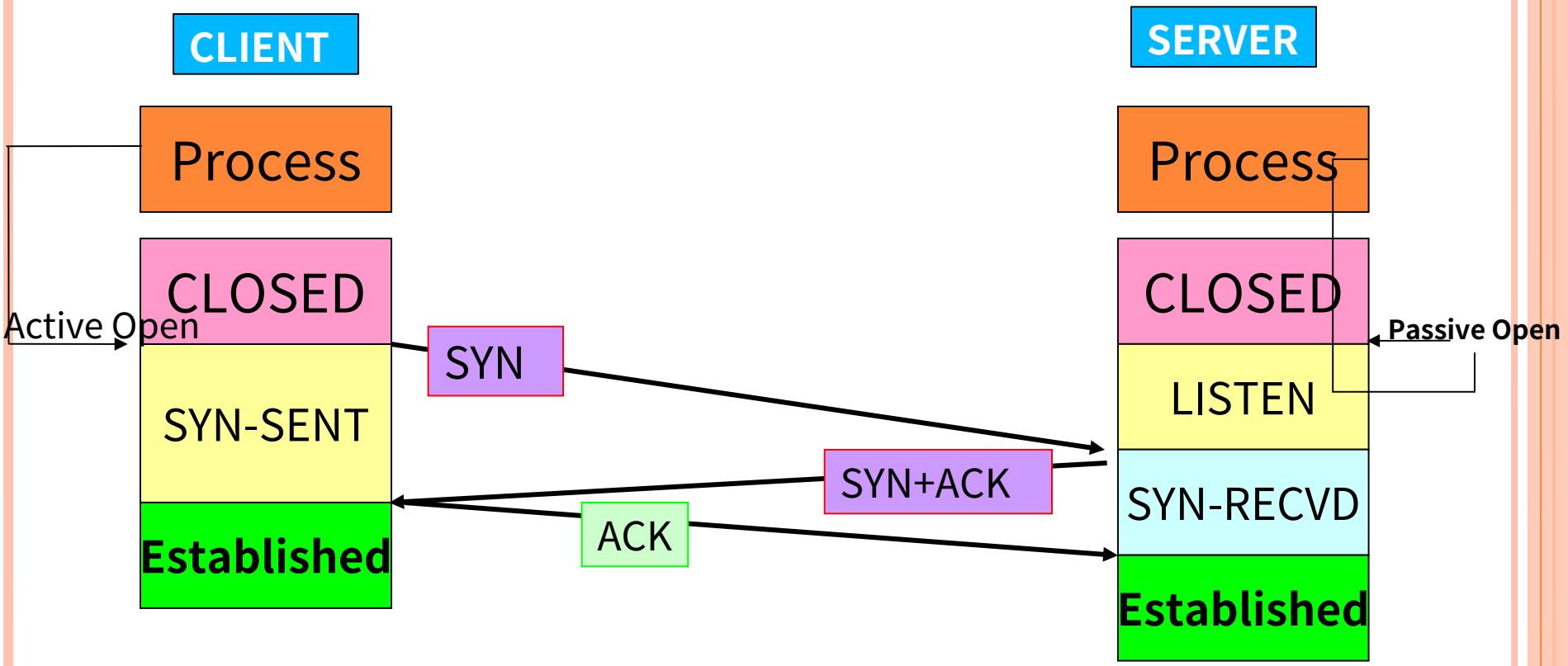
- e.g. send() and recv()

```
if( ( n = send( sockfd, buff, len, 0 ) ) != len ){ ... }
...
if( ( n = recv( sockfd, buff, BUFMAX, 0 ) ) < 0 ) { ... }
```

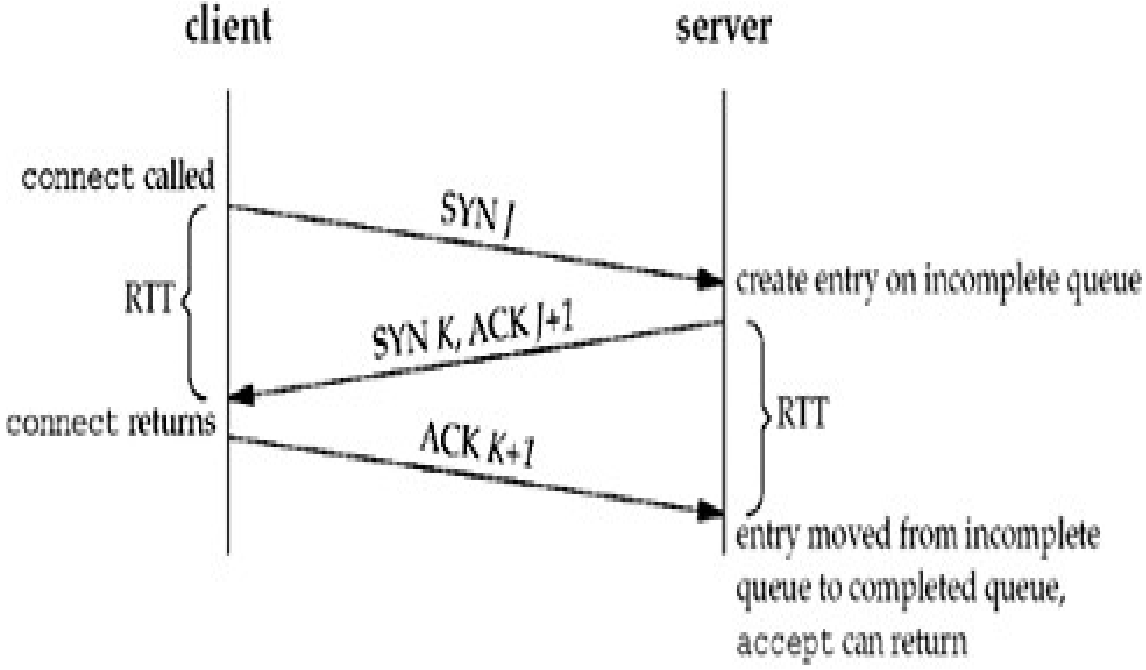
The two queues maintained by TCP for a listening socket.



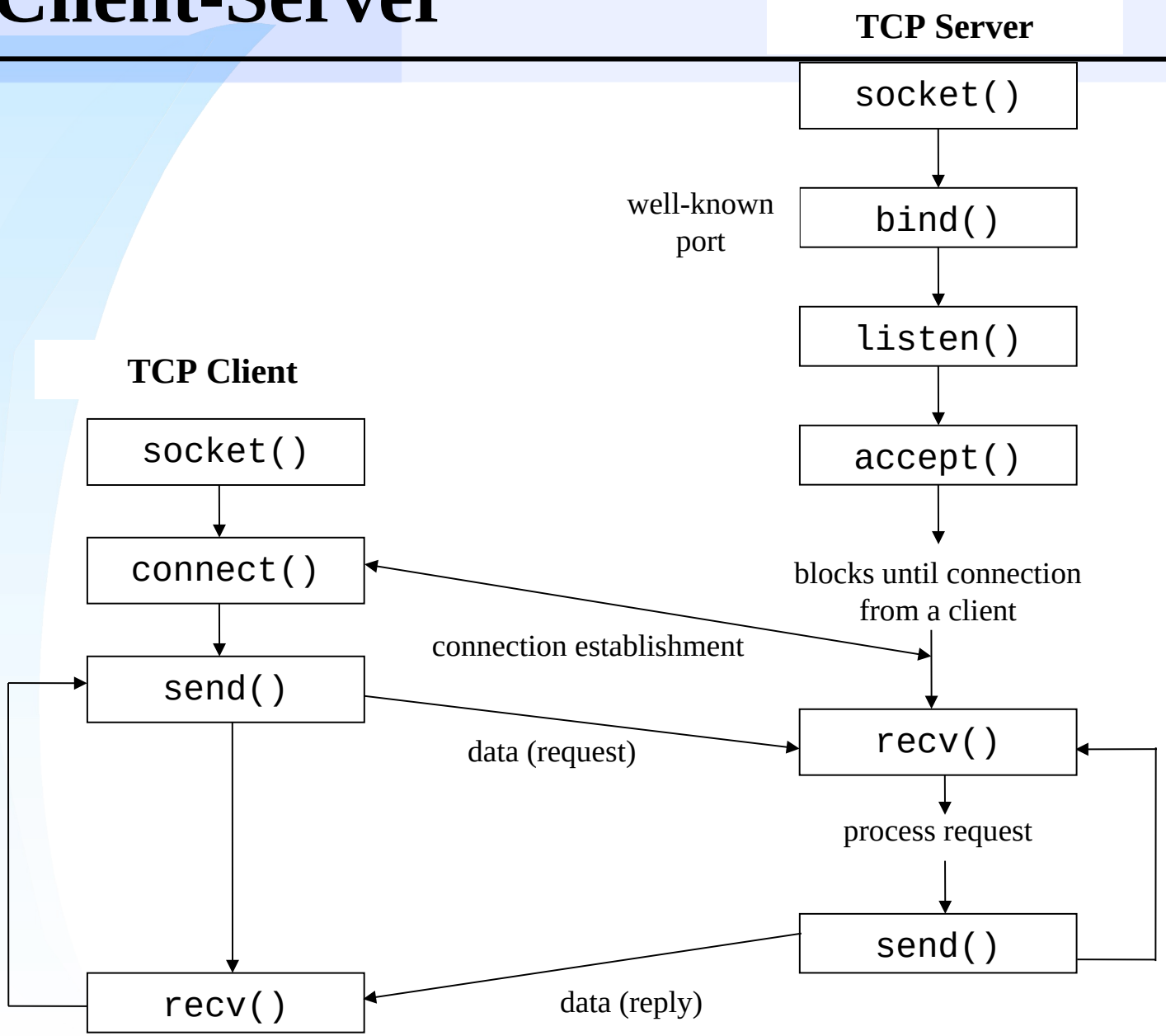
THREE-WAY HANDSHAKING



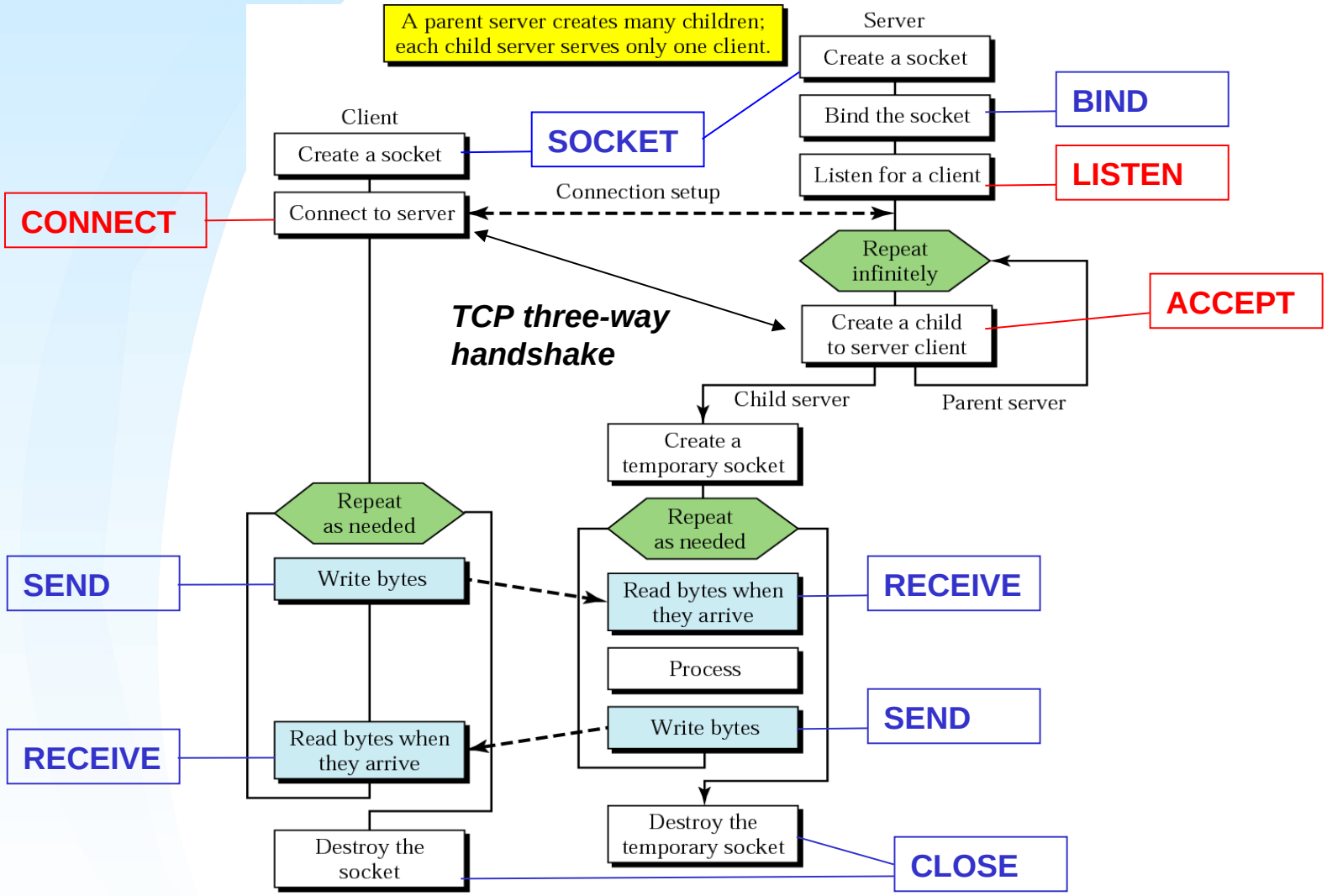
TCP three-way handshake and the two queues for a listening socket.



TCP Client-Server



Client+server: connection-oriented



Concurrent server

TCP SERVER

- struct sockaddr_in server_addr,client_addr;
- int sin_size;
- if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  perror("Socket");
  exit(1);
  }

- server_addr.sin_family = AF_INET;
- server_addr.sin_port = htons(5000);
- server_addr.sin_addr.s_addr = INADDR_ANY;
- bzero(&(server_addr.sin_zero),8);

- if (bind(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) ==-1)

{
  perror("Unable to bind");
  exit(1);
}

TCP SERVER

- if (listen(sock, 5) == -1)
  {
  perror("Listen");
  exit(1);
  }
- sin_size = sizeof(struct sockaddr_in);
- connected = accept(sock, (struct sockaddr *) &client_addr, &sin_size);
- char send_data[1024],recv_data[1024];
- gets(send_data);
- send(connected, send_data,strlen(send_data), 0);

TCP CLIENT

- struct sockaddr_in server_addr;

- if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)

```
{
        perror("Socket");
        exit(1);
}
```

- server_addr.sin_family = AF_INET;
- server_addr.sin_port = htons(5000);
- server_addr.sin_addr = inet_addr(“server IP Address”); //inet_addr(“192.168.148.12”)
- bzero(&(server_addr.sin_zero),8);

- if (connect(sock, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1)

```
    {
        perror("Connect");
      exit(1);
  }
```

TCP CLIENT

- char recv_data[1024];
- bytes_recieved=recv(sock,recv_data,1024,0);

References

- Unix Network Programming, Volume 1, Second Edition
  - W Richard Stevens
  - Pearson Education