CSE/PC/B/T/316
Computer Networks
Topic 1- Introduction:
Theory and Lab (Syllabus and COs), Socket Programming, Overview of Networking

Sarbani Roy
sarbani.roy@jadavpuruniversity.in
Office: CC-5-7
Cell: 9051639328

Syllabus

- Introduction: Uses of Computer Networks, Types of Computer Networks, OSI Reference Model, TCP/IP model [4L]
- Review of Physical Layer [4L]
- Data Link Control and Protocols: Link Layer Services, Error detection and Correction Techniques, Multi Access Protocols, Link Layer Addressing, Ethernet, Hubs, Switches and Switches, Point to Point Protocol, Asynchronous Transfer Mode, Multiprotocol Label Switching [6L]
- Network Layer: Introduction, Virtual Circuit and Datagram Networks, IP Addressing, Subnetting, Routing Algorithms (Link State, Distance Vector, Hierarchical), Routing in the Internet (RIP, OSPF, BGP), Broadcast and Multicast Routing Algorithms, Routers, ICMP, IPv6 [8L]
- Transport Layer: Introduction to Transport Layer Services, Multiplexing and Demultiplexing, Connectionless Transport: UDP, Principles of Reliable Data Transfer, Connection Oriented Transport: TCP, Principles of Congestion Control, TCP Congestion Control, Sockets and Socket Programming, Quality of services (QOS) [8L]
- Application Layer: Web and HTTP, Domain Name Space (DNS), Electronic Mail (SMTP, MIME, IMAP, POP3), File Transfer Protocol, Cryptography [6L]
- Introduction to Wireless and Mobile Networks [4L]

COs of Theory

- CO1: Understand the layered architecture and explain the contemporary issues and importance of MAC sublayer of the Data Link Layer, network layer, transport layer and application layer of TCP-IP model, and how they can be used to assist in network design and implementation.
- CO2: Understand the protocols of MAC sublayer of the Data Link Layer and describe the IEEE standards for Ethernet (802.3) and wireless LAN (802.11).
- CO3: Understand internetworking principles and explain algorithms for multiple access, routing and different networking techniques and able to analyze the performance of these algorithms.
- CO4: Explain the protocols in Transport Layer and able to design network applications using these protocols.
- CO5: Explain the protocols in application layer and how they can be used to design popular internet applications.

COs of Lab

- CO1: Design and implement error detection techniques within a simulated network environment.
- CO2: Design and implement flow control mechanisms of Logical Link Control of Data Link Layer within a simulated network environment.
- CO3: Design and implement medium access control mechanisms within a simulated network environment using IEEE 802 standards. CO4: Design and implement routing protocols within a simulated network environment.
- CO5: Analyze protocols and network traffic within a simulated network environment or using network tools.
- CO6: Design and implement various applications using Transport layer protocols and Application layer protocols for its implementation in client/server environments and analyze the performance.

Suggested Readings

1. Data Communications and Networking, Behrouz A Forouzan, McGraw Hill
2. Computer Networks, by Andrew S. Tanenbaum, Prentice Hall India
3. Computer Networking: A Top-Down Approach Featuring the Internet, by James F. Kurose and Keith W. Ross, Pearson Education

Slides are mainly prepared using online documents available of the above books and course materials of different institutes

REVIEW OF ERROR DETECTION IN DATA LINK LAYER

Computer Networks Lab

CO1: Design and implement error detection techniques within a simulated network environment.

Assignment 1: Design and implement an error detection module which has two schemes namely Checksum and Cyclic Redundancy Check(CRC).

Due on: 31 July - 04 August 2023 (in your respective lab classes)

Report submission due on: 6 August 2023

Language: You can write the program in any high level language like C, C++, Java, Python etc.

Please note that you may need to use these schemes separately for other applications (assignments).

Program

- **Sender Program:** The Sender program should accept the name of a test file (contains a sequence of 0,1) from the command line. Then it will prepare the dataword from the input. Based on the schemes, codewords will be prepared. Sender will send the codewords to the Receiver.
  - **Error injection module:** Inject error in random positions in the input data frame. Write a separate method for that. Sender program will randomly call this method before sending the codewords to the receiver.
- **Receiver Program:** Receiver will check if there is any error detected. Based on the detection it will accept or reject the dataword.

Approaches

- **Checksum (16-bit):** Checksum of a block of data is the complement of the one's complement of the 16-bit sum of the block. The message (from the input file) is divided into 16-bit words. The value of the checksum word is set to 0. All words are added using 1's complement addition. The sum is complemented and becomes the checksum. So, if we transmit the block of data including the checksum field, the receiver should see a checksum of 0 if there are no bit errors.
- **CRC:** CRC generator polynomials will be given as input (CRC-8, CRC-10, CRC-16 and CRC-32). Show how good is the selected polynomial to detect single-bit error, two isolated single-bit errors, odd number of errors, and burst errors.
  - CRC-8: $x^8 + x^7 + x^6 + x^4 + x^2 + 1$ (Use: General, Bluetooth wireless communication)
  - CRC-10: $x^{10} + x^9 + x^5 + x^4 + x + 1$ (Use: General, Telecommunication)
  - CRC-16: $x^{16} + x^{15} + x^2 + 1$ (Use: USB)
  - CRC-32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (Use: Ethernet IEEE802.3)

Evaluation and Result

- **Error types:** single-bit error, two isolated single-bit errors, odd number of errors, and burst errors.
- Test the above two schemes for the error types and CRC polynomials mentioned above for the following cases (not limited to).
  - Error is detected by both CRC and Checksum.
  - Error is detected by checksum but not by CRC.
  - Error is detected by CRC but not by Checksum.

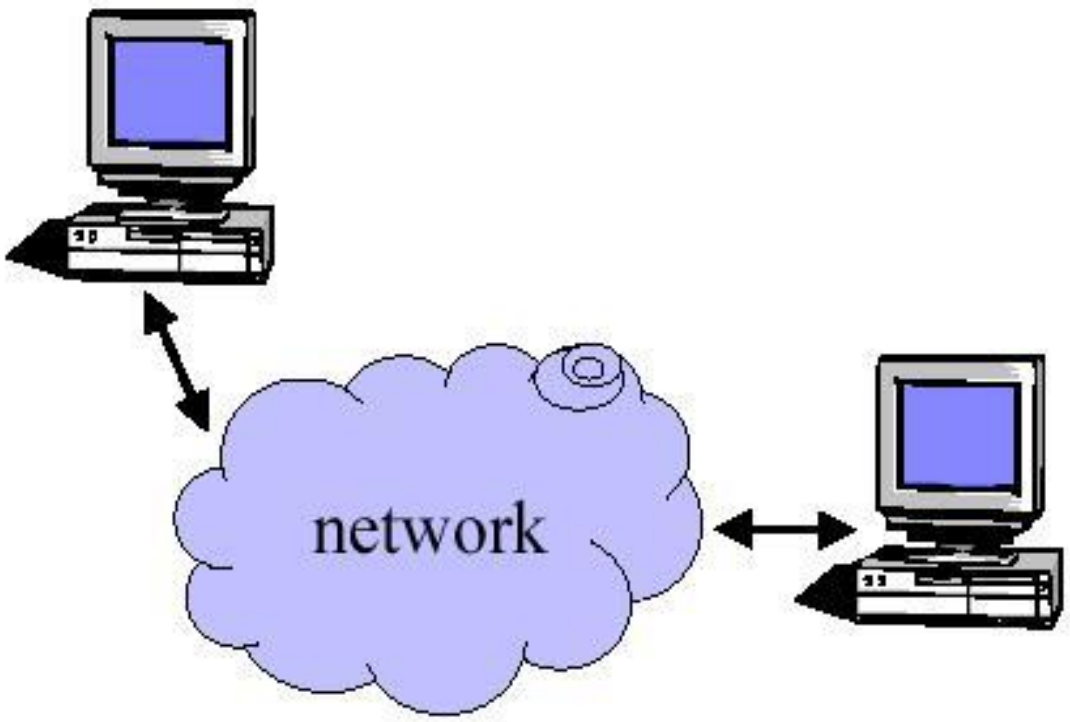
Report

- **Title:** Basic information like name, class, group, assignment number, problem statement and date (deadline and submission).
- **Design:** Short paragraph describing the purpose of the program, draw a structure diagram to reflect the procedural organization of the program as you have designed it in the previous step, input and output format.
- **Implementation:** Code snippet, method description, interfaces etc.
- **Test cases:** Describe the tests you will perform to verify the correctness of your program. This should be a *thorough* and *exhaustive* list of test cases designed to show that your program does everything it is supposed to do. The combined tests should exercise every line of code in your program. For each test case, give sample test data and state what you are checking.
- **Results:** Figures or graphs, tables
- **Analysis:** Discussion on results considering different test cases, error, correctness, known bugs, possible improvements
- **Comments:** Evaluate the lab assignment. What did you learn from it? Was it too hard? (explain why?) Too easy? (explain why?) Suggest improvements if you can.

SOCKET PROGRAMMING

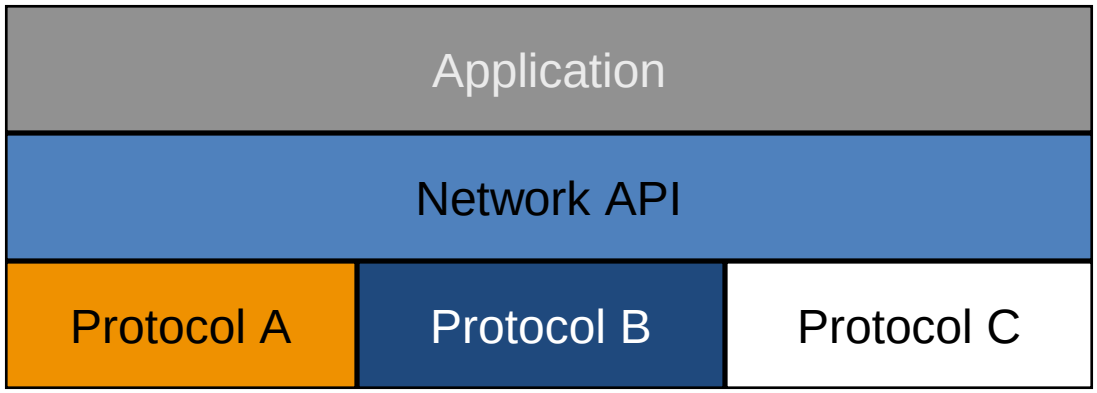
Why do we need sockets?

Provides an abstraction for inter-process communication



OS point of view

- The services provided (often by the operating system) that provide the interface between application and protocol software.



Functions

- Define an “end- point” for communication
- Initiate and accept a connection
- Send and receive data
- Terminate a connection gracefully

Examples

- File transfer apps (FTP), Web browsers
- (HTTP), Email (SMTP/ POP3), etc…

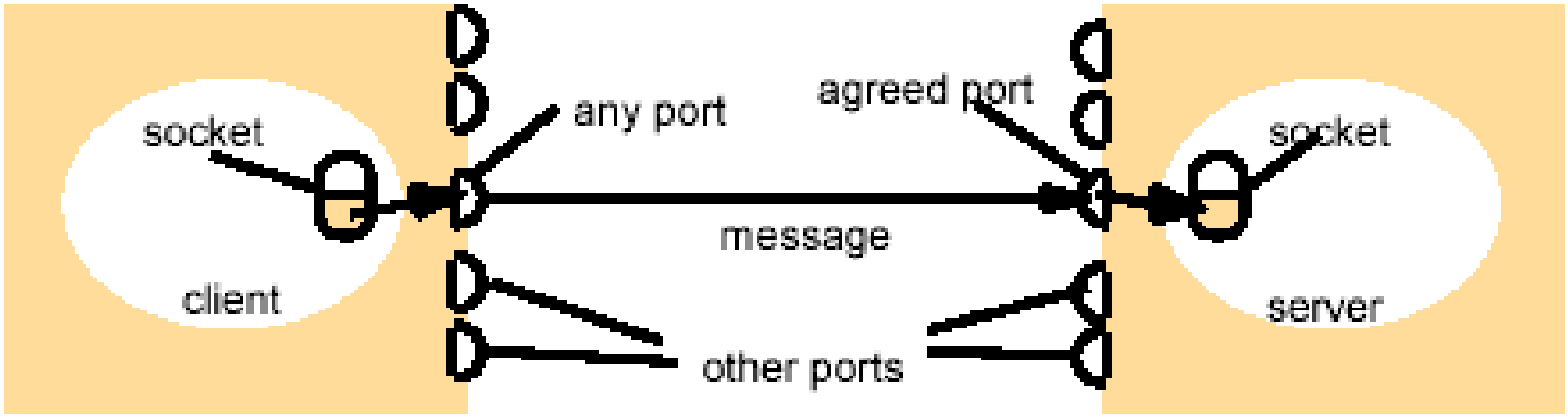
Types of Sockets

- Two different types of sockets :
  - stream vs. datagram

- **Stream socket**:(*a. k.* a. connection- oriented socket)
  - It provides reliable, connected networking service
  - Error free; no out- of- order packets (uses TCP)
  - applications: telnet/ ssh, http, …

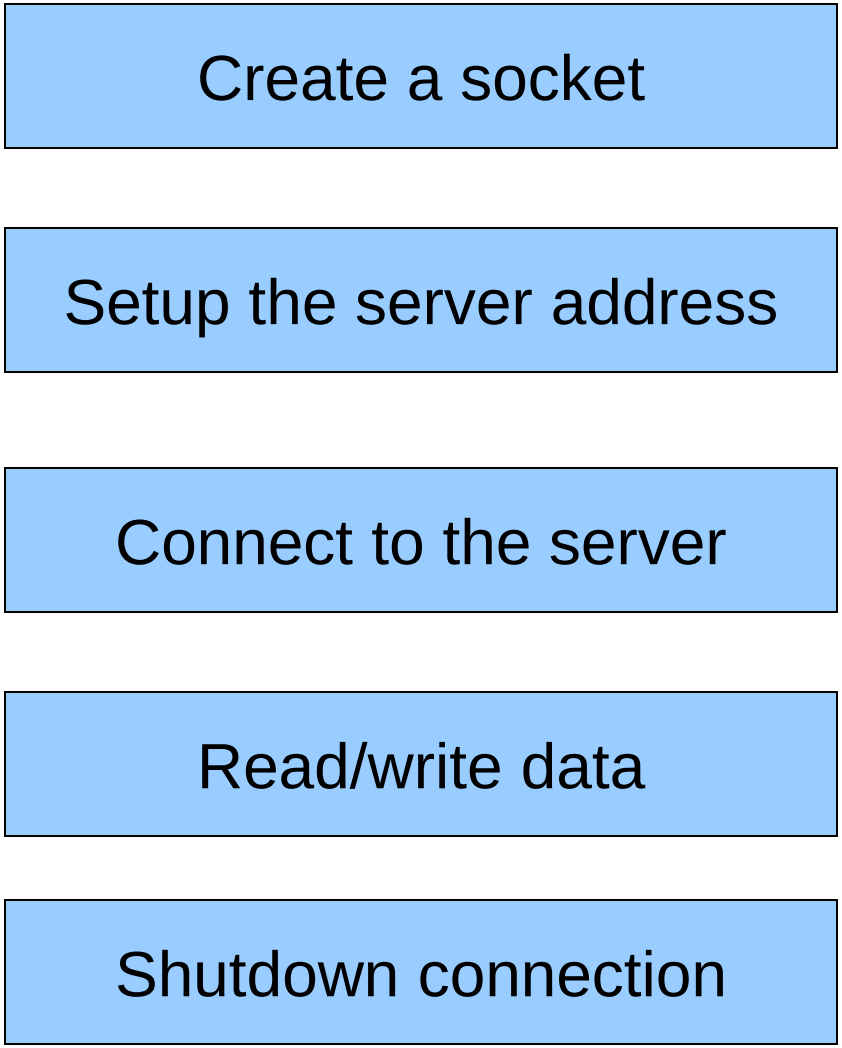
- **Datagram socket**:(*a. k.* a. connectionless socket)
  - It provides unreliable, best- effort networking service
  - Packets may be lost; may arrive out of order (uses UDP)
  - applications: streaming audio/ video (realplayer), …

Addressing



Client ⟷ Server

Client – high level view



```
int connect_ socket( char *hostnar
    int sock;
    struct sockaddr_in sin;
    struct hostent *host;
    sock = socket( AF_ INET, SOCK_
    if (sock == -1)
          return sock;
    host = gethostbyname( hostname
    if (host == NULL) {
          close( sock);
          return -1;
    }
    memset (& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ port = htons( port);
    sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[
    0];
    if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
          close (sock);
          return -1;
    }
    return sock;
}
```

Ipv4 socket address structure

```
struct socketaddr_in{
  uint8_t          sin_len;   /*length of the structure (16)*/
  sa_falimily_t    sin_family /* AF_INT*/
  in_port_t        sin_port   /* 16 bit TCP or UDP port
number*/
  struct in_addr   sin_addr   /* 32 bit Ipv4 address */
  char             sin_zero(8)/* unused*/
}
```

```
int connect_ socket( char *hostname, int port) {
    int sock;
    struct sockaddr_in sin;
    struct hostent *host;
    sock = socket( AF_ INET, SOCI
    if (sock == -1)
          return sock;
    host = gethostbyname( hostnar
    if (host == NULL) {
          close( sock);
          return -1;
    }
    memset (& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ port = htons( port);
    sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[
    0];
    if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
          close (sock);
          return -1;
    }
    return sock;
}
```

Hostent structure

```
struct hostent{
  char *    h_name        /*official name of host*/
  char **   h_aliases;    /* pointer ot array of\
                             pointers to alias name*/
  int       h_addrtype    /* host address type*/
  int       h_length      /* length of address */
  char **   h_addr_list   /*prt to array of ptrs with \
                            IPv4 or IPv6 address*/
}
```

```
int connect_ socket( char *hostname, int port) {
    int sock;
    struct sockaddr_in sin;
    struct hostent *host;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock == -1)
         return sock;
    host = gethostbyname( ho
    if (host == NULL) {
         close( sock);
         return -1;
    }
    memset (& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ port = htons( port);
    sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[
    0];
    if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
         close (sock);
         return -1;
    }
    return sock;
}
```

Make the socket

Socket(int family , int type, in t protocol);
return nonnegative value for OK, -1 for error

```
int connect_ socket( char *hostname, int port) {
    int sock;
    struct sockaddr_in sin;
    struct hostent *host;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock == -1)
         return sock;
    host = gethostbyname( hostname);
    if (host == NULL) {
         close( sock);
         return -1;
    }
    memset (& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ port = htons( port);
    sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[
    0];
    if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
         close (sock);
         return -1;
    }
    return sock;
}
```

Resolve the host

```
struct hostent *gethostbyname( const char *hostname);
/*Return nonnull pointer if OK, NULL on error */
```

```
int connect_ socket( char *hostname, int port) {
    int sock;
    struct sockaddr_in sin;
    struct hostent *host;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock == -1)
         return sock;
    host = gethostbyname( hostname);
    if (host == NULL) {
         close( sock);
         return -1;
    }
    memset (& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ port = htons( port);
    sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[
    0];
    if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
         close (sock);
         return -1;
    }
    return sock;
}
```

Setup up the struct

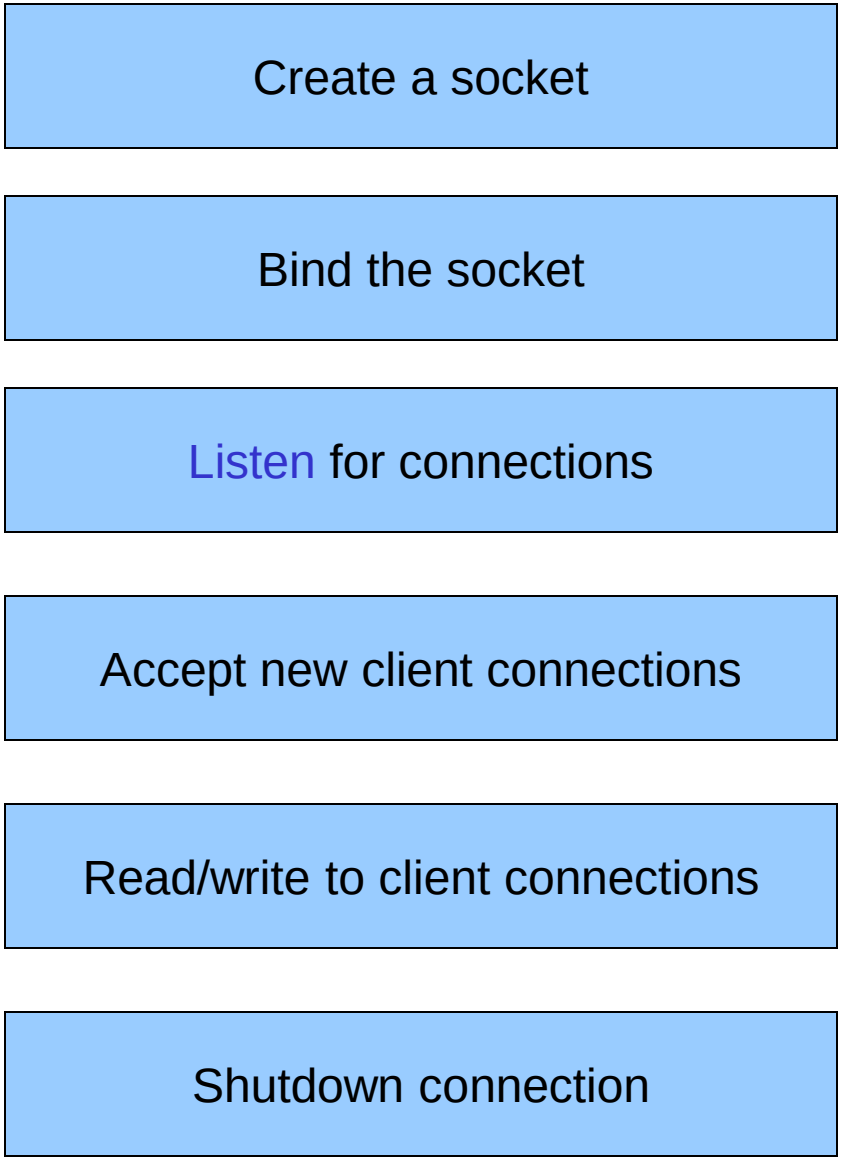
unit16_t htons(unit16_t host16bitvaule)
/*Change the port number from host byte order to network byte order */

```
int connect_ socket( char *hostname, int port) {
    int sock;
    struct sockaddr_in sin;
    struct hostent *host;
    sock = socket( AF_ INET, SOCK_ STREAM, 0);
    if (sock == -1)
          return sock;
    host = gethostbyname( hostname);
    if (host == NULL) {
          close( sock);
          return -1;
    }
    memset (& sin, 0, sizeof( sin));
    sin. sin_ family = AF_ INET;
    sin. sin_ port = htons( port);
    sin. sin_ addr. s_ addr = *( unsigned long *) host-> h_ addr_ list[
    0];
    if (connect( sock, (struct sockaddr *) &sin, sizeof( sin)) != 0) {
          close (sock);
          return -1;
    }
    return sock;
}
```

Connect

connect(int socketfd, const struct sockaddr * servaddr,
socket_t addrlen)
/*Perform the TCP three way handshaking*/

Server – high level view



Listening on a port (TCP)

```
int make_ listen_ socket( int port) {
   struct sockaddr_ in sin;
   int sock;
   sock = socket( AF_ INET, SOCK_ STREAM, 0);
   if (sock < 0)
       return -1;
   memset(& sin, 0, sizeof( sin));
   sin. sin_ family = AF_ INET;
   sin. sin_ addr. s_ addr = htonl( INADDR_ ANY);
   sin. sin_ port = htons( port);
   if (bind( sock, (struct sockaddr *) &sin, sizeof(
   sin)) < 0)
       return -1;
   return sock;
}
```

Make the socket

Setup up the struct

Bind

bind(int sockfd, const struct sockaddr * myaddr, socklen_t addrlen);
/* return 0 if OK, -1 on error
assigns a local protocol adress to a socket*/

Accepting a client connection (TCP)

```
int get_ client_ socket( int listen_ socket) {
   struct sockaddr_ in sin;
   int sock;
   int sin_ len;
   memset(& sin, 0, sizeof( sin));
   sin_ len = sizeof( sin);
   sock = accept( listen_ socket, (struct sockaddr *)
   &sin, &sin_ len);
   return sock;
}
```

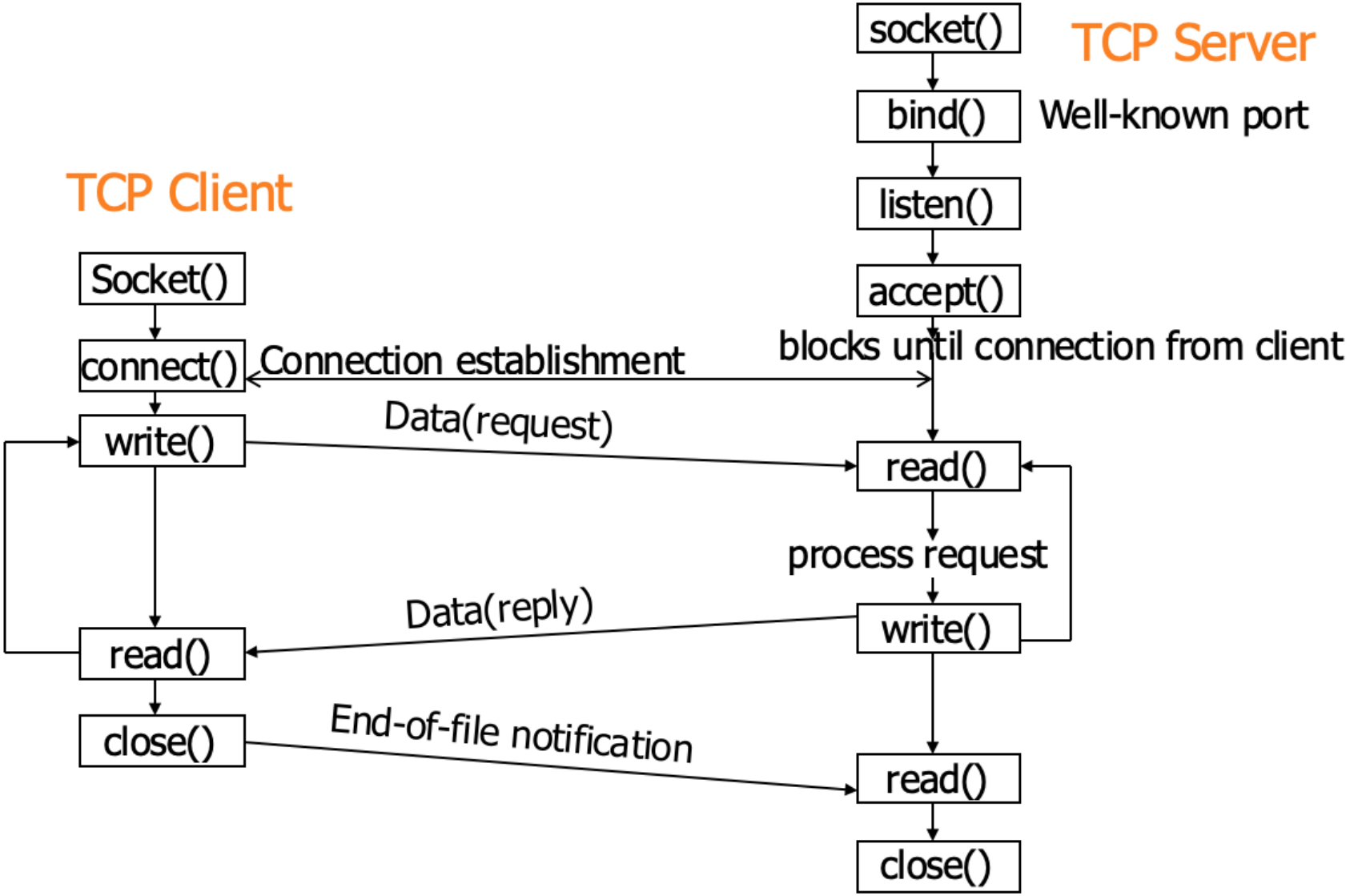
Setup up the struct

Accept the client connection

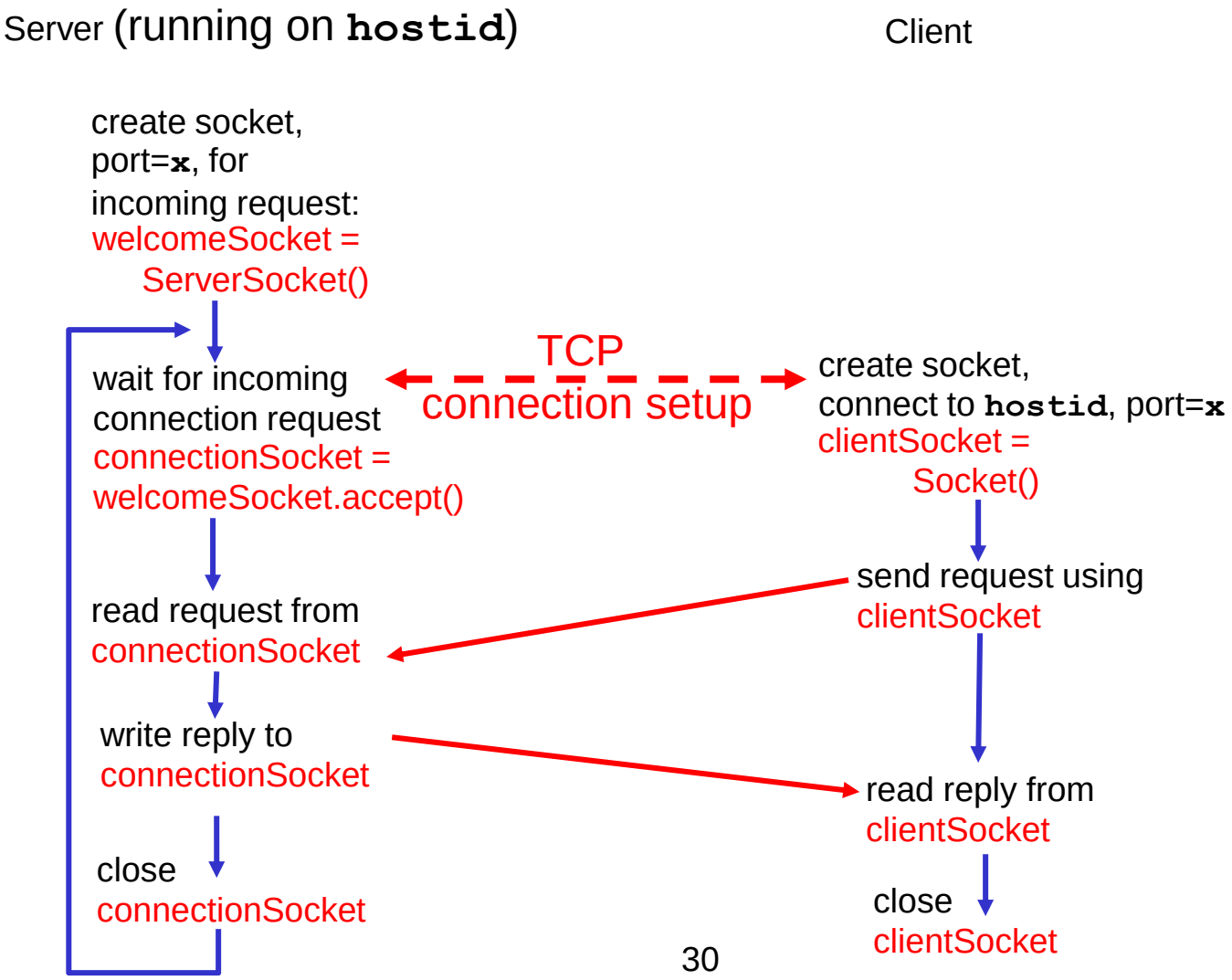
accept(int sockefd, struct sockaddr * claddr, socklen_t * addrlen)
/* return nonnegative descriptor if OK, -1 on error
return the next completed connection from the front of the completed connection queue.
if the queue is empty,
the process is put to sleep(assuming blocking socket)*/

Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]


TCP Server
socket()
bind()
Well-known port
listen()
accept()
blocks until connection from client
TCP Client
Socket()
connect()
Connection establishment
write()
Data(request)
read()
process request
write()
Data(reply)
read()
close()
End-of-file notification
read()
close()

Client/server socket interaction: TCP



Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

Create input stream attached to socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

    }
}
```

Send line to server

Read line from server

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

       Socket connectionSocket = welcomeSocket.accept();

      BufferedReader inFromClient =
        new BufferedReader(new
        InputStreamReader(connectionSocket.getInputStream()));
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

Create output stream, attached to socket

Read in line from socket

Write out line to socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);
            }
        }
    }
```

End of while loop, loop back and wait for another client connection

server.py

```
import socket #import the socket library

s_obj = socket.socket() # creating a socket object

port = 4321  # reserve a port on your computer. It can be anything (0-65535)

s_obj.bind((' ', port))  # Next, bind to the port.
# The empty string makes the server listen to requests coming from other computers on the
network

print("socket binded to %s" %(port))

# put the socket into listening mode
# Here means that 4 connections are kept waiting if the server is busy
# and if a 5th socket tries to connect then the connection is refused.
s_obj.listen(4)
print("socket is listening")

# a forever loop until we interrupt it or an error occurs
while True:

   # Establish connection with client.
   c, addr = s_obj.accept()
   print ('Got connection from', addr )

  # send a message to the client. Encoding to send byte type.
   c.send('Thank you for connecting'.encode())

   c.close()  # Close the connection with the client

   break   # Breaking once connection closed
```

client.py

```
import socket  # Import socket

s_obj = socket.socket()  # Create a socket object

port = 4321  # Define the port on which you want to connect

s_obj.connect(('127.0.0.1', port))  # connect to the server on local computer

print(s_obj.recv(1024).decode())  # receive data from the server and decoding to get the string.

s_obj.close()  # close the connection
```

Dealing with blocking calls

- Many functions block
  - accept(), connect(),
  - All recv()
- For simple programs this is fine
- What about complex connection routines
  - Multiple connections
  - Simultaneous sends and receives
  - Simultaneously doing non-networking processing

Dealing with blocking (cont..)

- Options
  - Create multi-process or multi-threaded code
  - Turn off blocking feature (*fcntl()* system call)
  - Use the *select()* function
- What does select() do?
  - Can be permanent blocking, time-limited blocking or non-blocking
  - Input: a set of file descriptors
  - Output: info on the file-descriptors' status
  - Therefore, can identify sockets that are "ready for use": calls involving that socket will return immediately

select function call

- int status = select()
  - Status: # of ready objects, -1 if error
  - nfds: 1 +largest file descriptor to check
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - Timeout: time after which select returns

UDP socket()

- The UDP server must create a **datagram** socket…

```
int fd;                 /* socket descriptor */

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(1);
}
```

- ***socket*** returns an integer (**socket descriptor**)
  - ***fd*** < 0 indicates that an error occurred

- AF_INET: associates a socket with the Internet protocol family
- **SOCK_DGRAM: selects the UDP protocol**

Socket I/O: bind()

- A ***socket*** can be bound to a ***port***

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */

/* create the socket */

/* bind: use the Internet address family */
srv.sin_family = AF_INET;

/* bind: socket 'fd' to port 80*/
srv.sin_port = htons(80);

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
        perror("bind"); exit(1);
}
```

- **Now the UDP server is ready to accept packets…**

Socket I/O: recvfrom()

- ***read*** does not provide the client's address to the UDP server

```
int fd;                                 /* socket descriptor */
struct sockaddr_in srv;                 /* used by bind() */
struct sockaddr_in cli;                 /* used by recvfrom()
*/
char buf[512];                          /* used by recvfrom()
*/
int cli_len = sizeof(cli);              /* used by recvfrom()
*/
int nbytes;                             /* used by recvfrom()
*/

/* 1) create the socket */
/* 2) bind to the socket */

nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
```

Socket I/O: recvfrom() continued...

```
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) cli, &cli_len);
```

- The actions performed by ***recvfrom***
  - returns the number of bytes read (***nbytes***)
  - copies ***nbytes*** of data into ***buf***
  - returns the address of the client (***cli***)
  - returns the length of ***cli*** (***cli_len***)
  - don't worry about flags

Socket I/O: sendto()

- ***write*** is not allowed
- Notice that the UDP client does not ***bind*** a port number
  - a port number is **dynamically assigned** when the first ***sendto*** is called

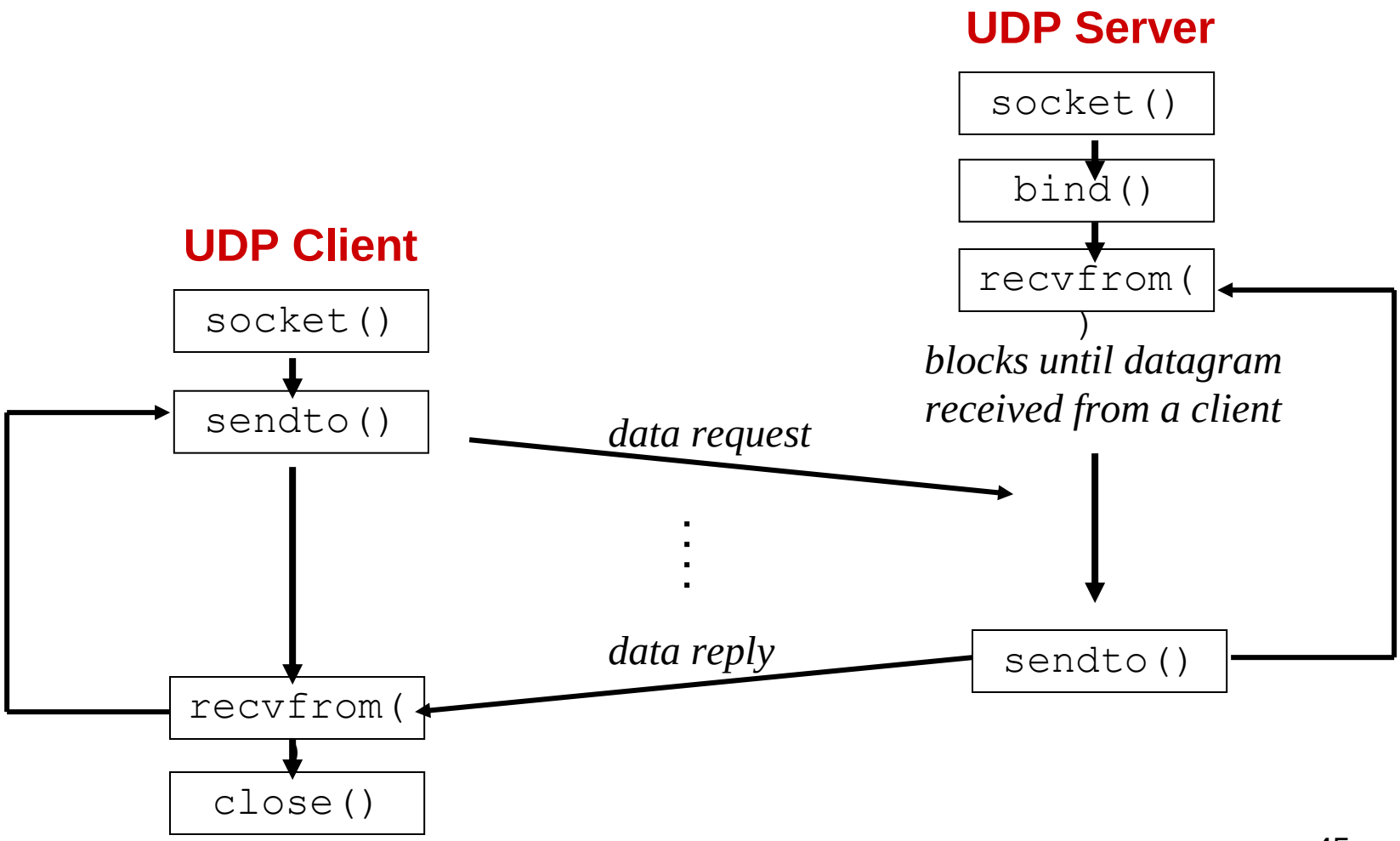
```
int fd;                              /* socket descriptor */
struct sockaddr_in srv;                    /* used by sendto() */

/* 1) create the socket */

/* sendto: send data to IP Address "128.2.35.50" port 80 */
srv.sin_family = AF_INET;
srv.sin_port = htons(80);
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
                    (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
        perror("sendto");      exit(1);
}
```

Review: UDP Client-Server Interaction



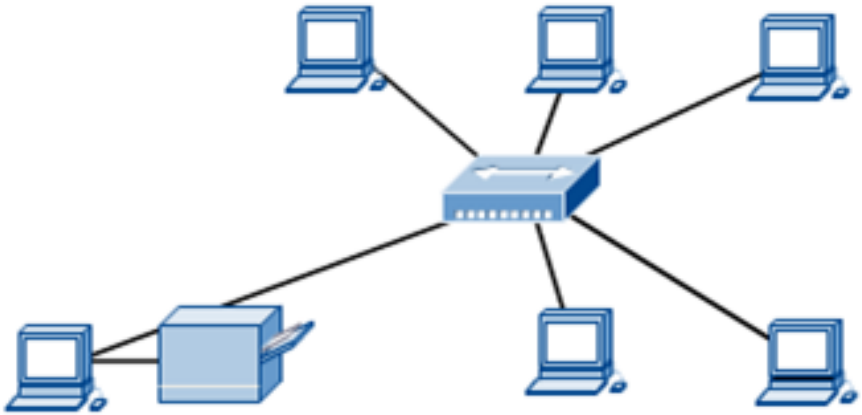
from UNIX Network Programming Volume 1, figure 8.1

Networks

- Network is a set of devices (often referred to as nodes) connected by communication links.
- A node can be a computer, printer, or any other device capable of sending and/or receiving data generated by other nodes on the network. A link can be a cable, air, optical fiber, or any *medium* which can transport a signal carrying information.

Computer Networks

- Computer network connects two or more autonomous computers.

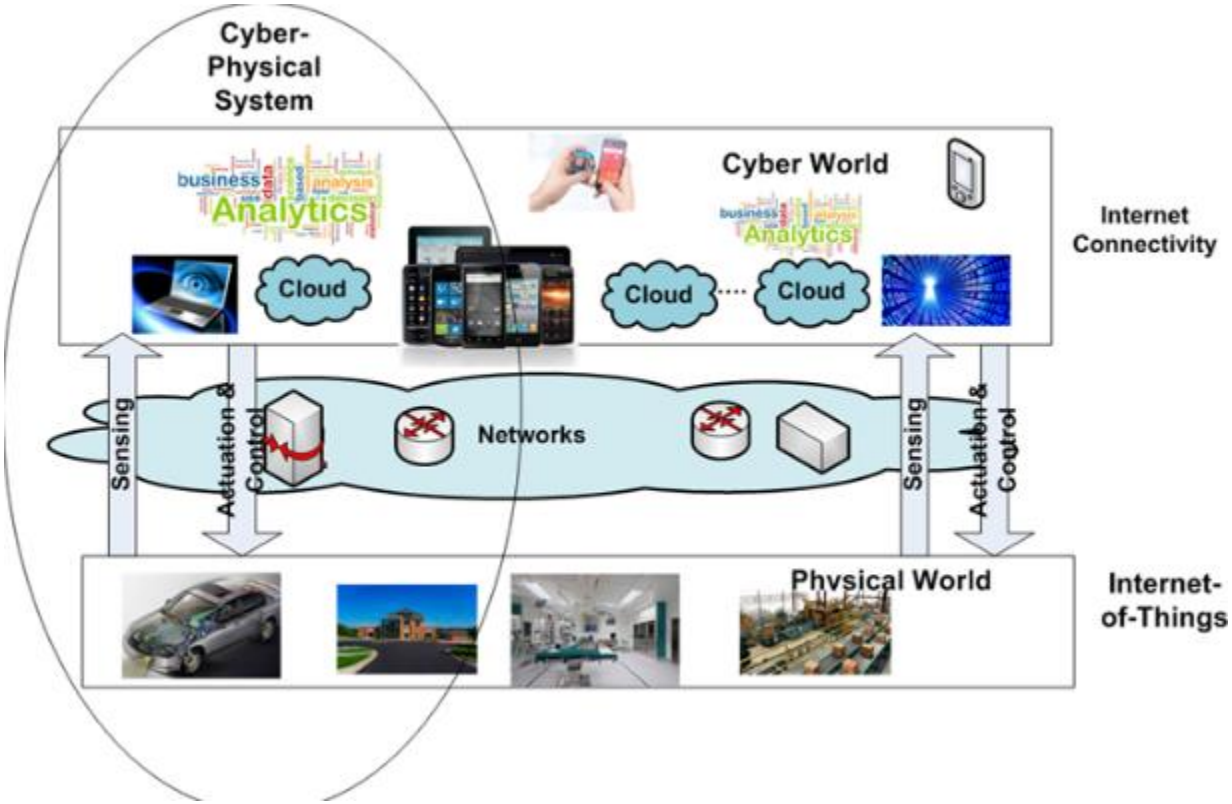


- The computers can be geographically located anywhere.

Applications of Networks

- **Resource Sharing**
  - Hardware (computing resources, disks, printers)
  - Software (application software)
- **Information Sharing**
  - Easy accessibility from anywhere (files, databases)
  - Search Capability (WWW)
- **Communication**
  - Email
  - Message broadcast
- **Remote computing**
- **Distributed processing (Cloud Computing)**

Some terms..CPS, IoT, Cloud



Network Components

- **Physical Media**
- **Interconnecting Devices**
- **Computers**
- **Networking Software**
- **Applications**

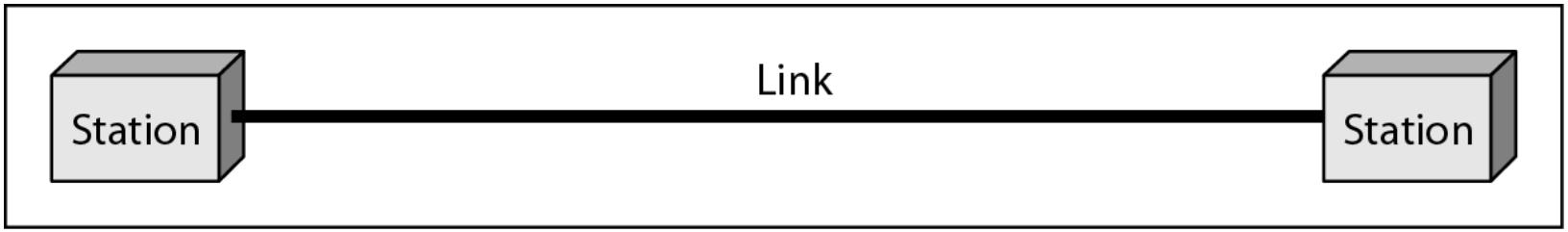
Network Criteria

- Performance
  - Depends on Network Elements
  - Measured in terms of Delay and Throughput
- Reliability
  - Failure rate of network components
  - Measured in terms of availability/robustness
- Security
  - Data protection against corruption/loss of data due to:
    - Errors
    - Malicious users

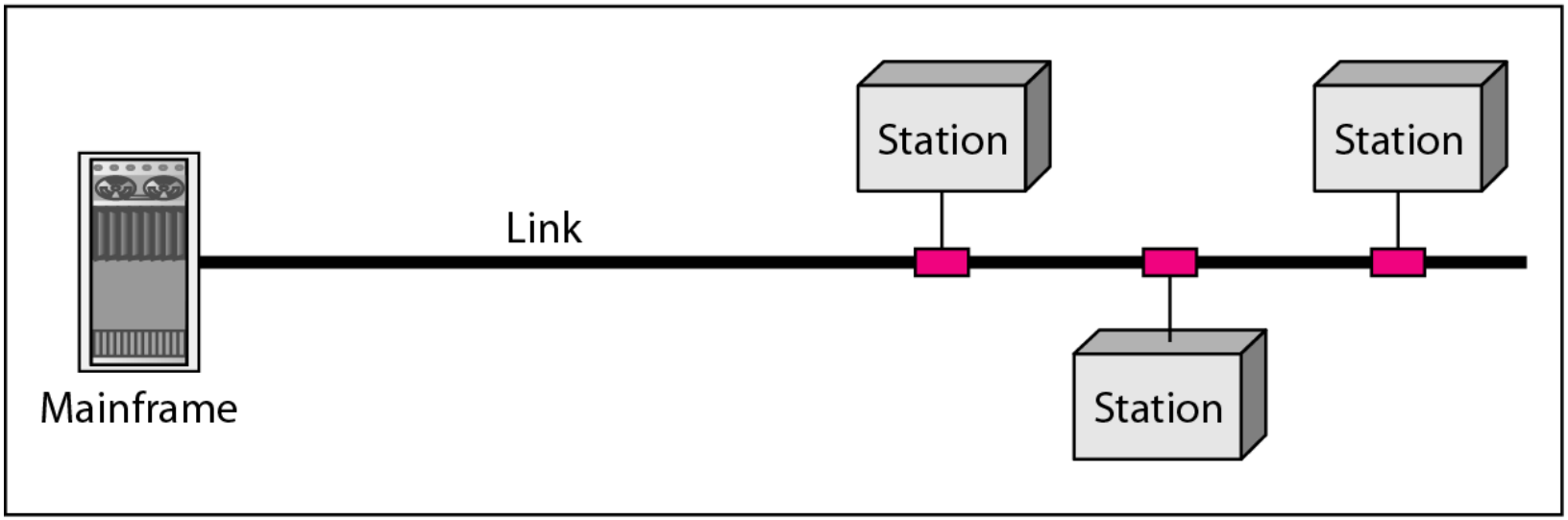
Physical Structures

- Type of Connection
  - Point to Point - single transmitter and receiver
  - Multipoint - multiple recipients of single transmission
- Physical Topology
  - Connection of devices
  - Type of transmission - unicast, mulitcast, broadcast

Types of Connections

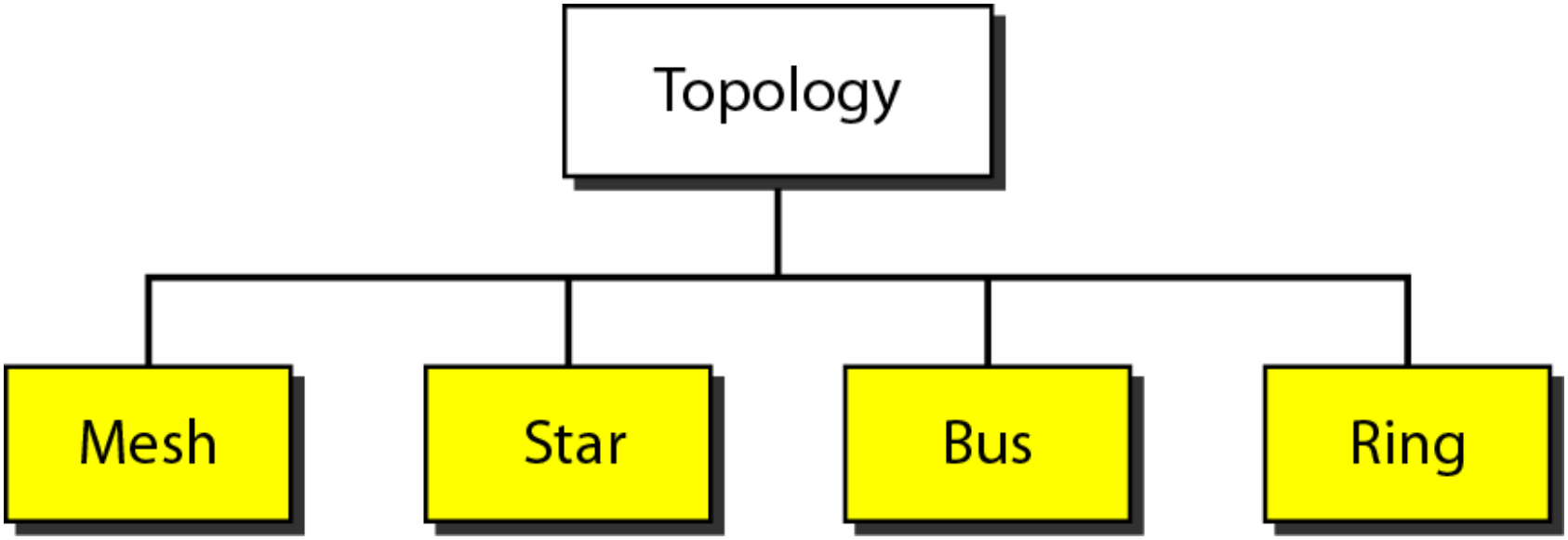


a. Point-to-point



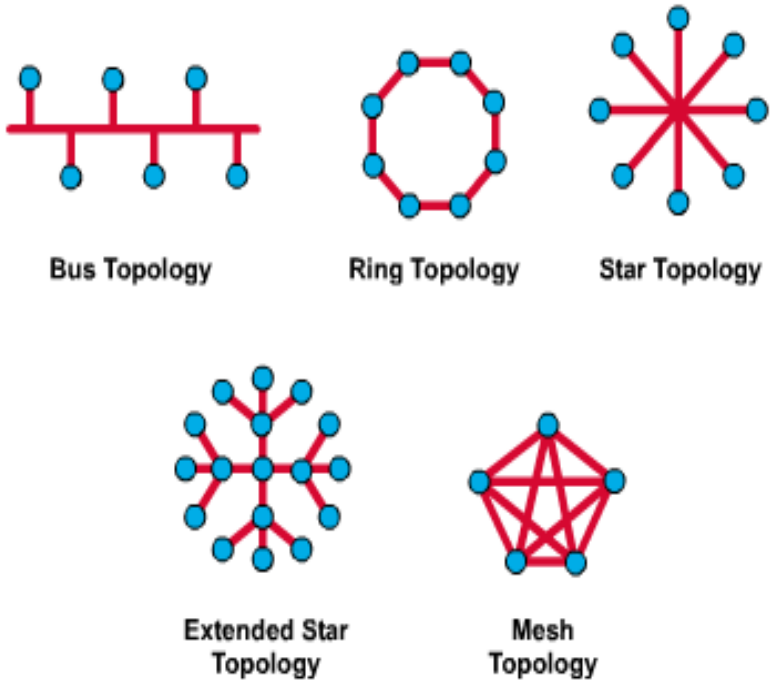
b. Multipoint

Categories of Topology

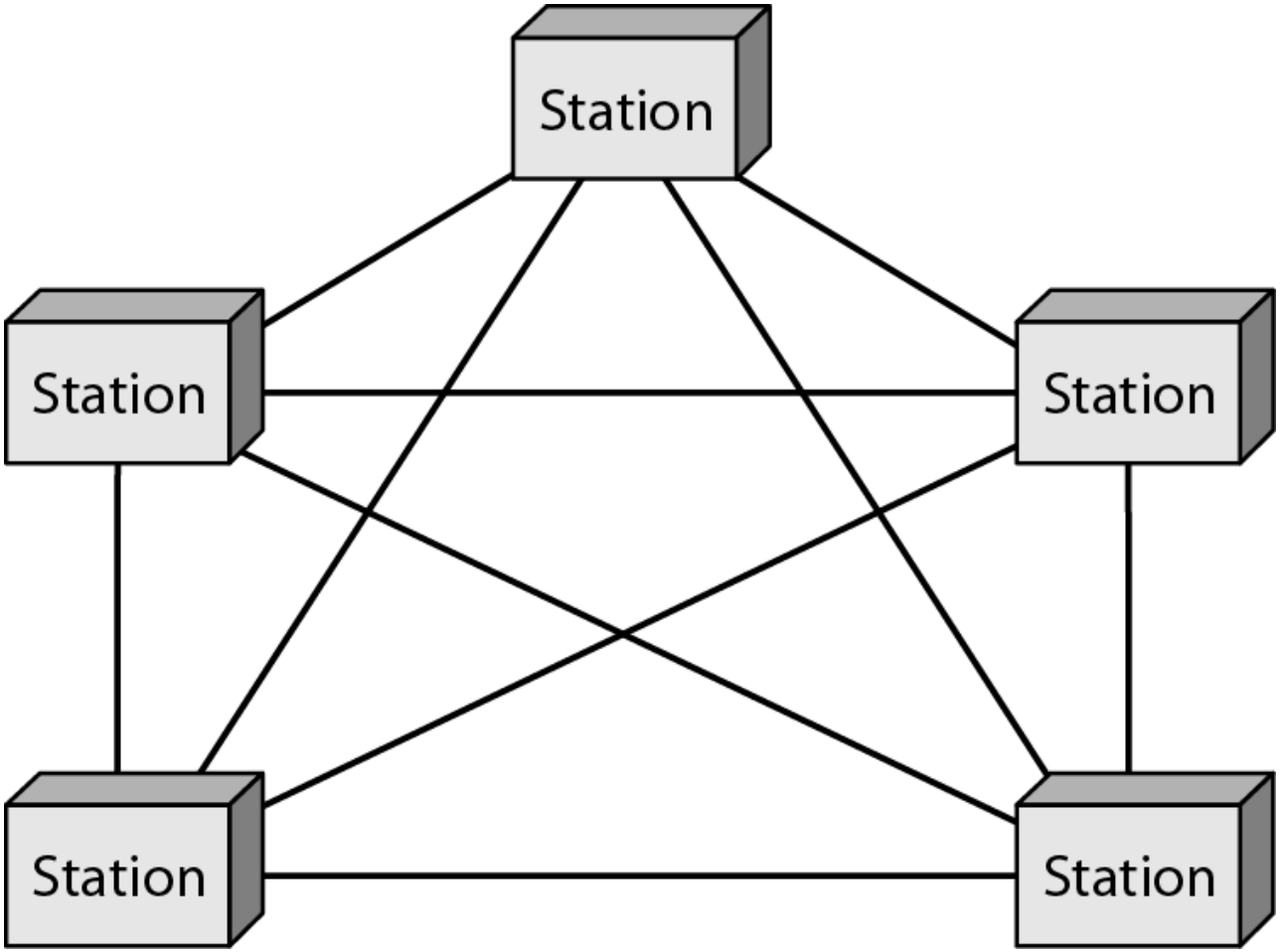


Network Topology

- The network topology defines the way in which computers, printers, and other devices are connected. A network topology describes the layout of the wire and devices as well as the paths used by data transmissions.

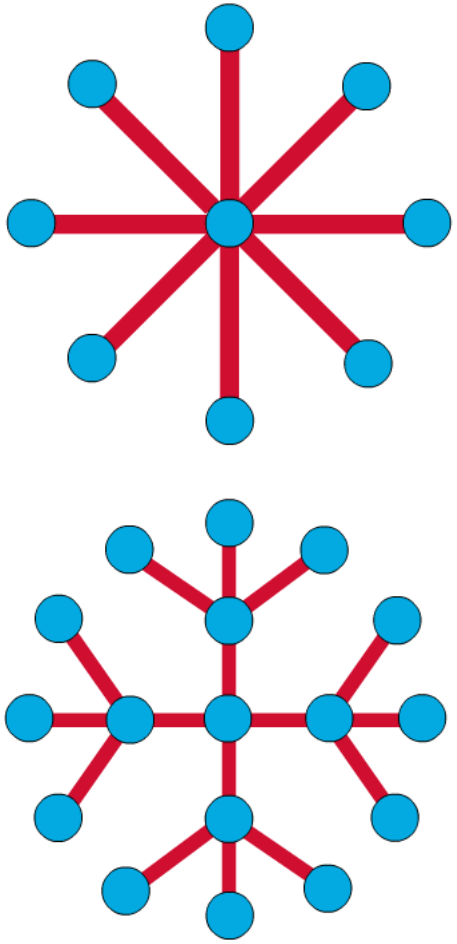


A fully connected mesh topology (five devices)

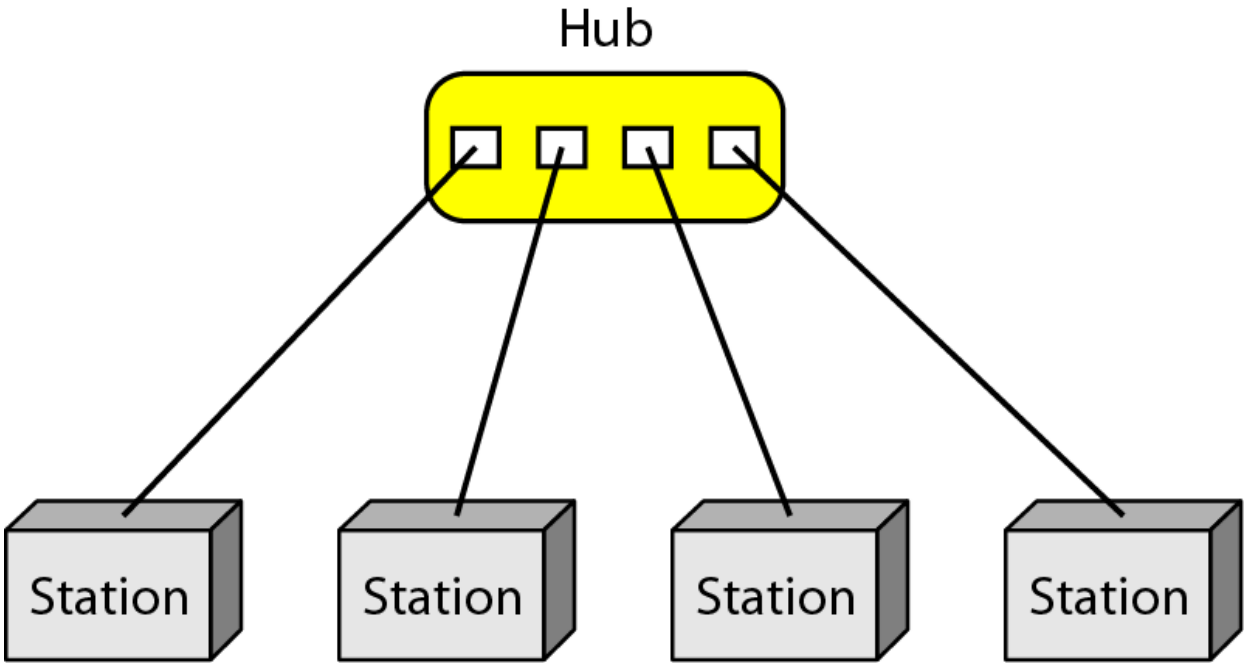


Star Topology

- The star topology is the most commonly used architecture in Ethernet LANs.
- Larger networks use the extended star topology also called tree topology. When used with network devices that filter frames or packets, like bridges, switches, and routers, this topology significantly reduces the traffic on the wires by sending packets only to the wires of the destination host.

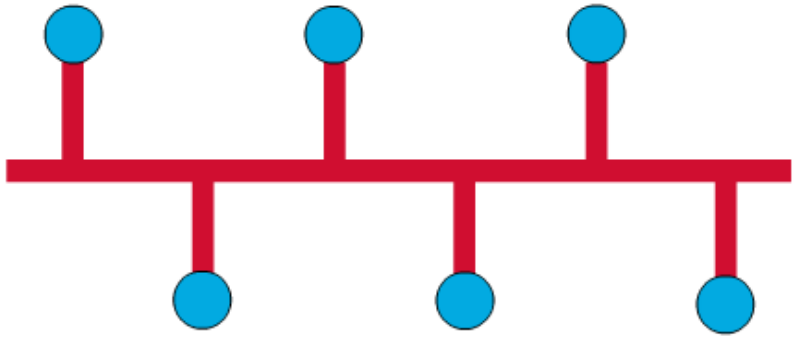


A star topology connecting four stations

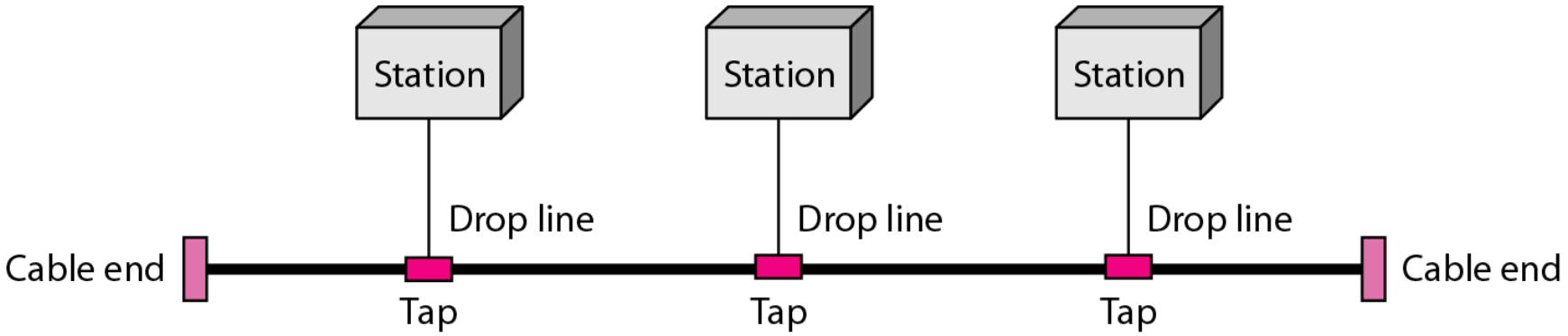


Bus Topology

- Commonly referred to as a linear bus, all the devices on a bus topology are connected by one single cable.

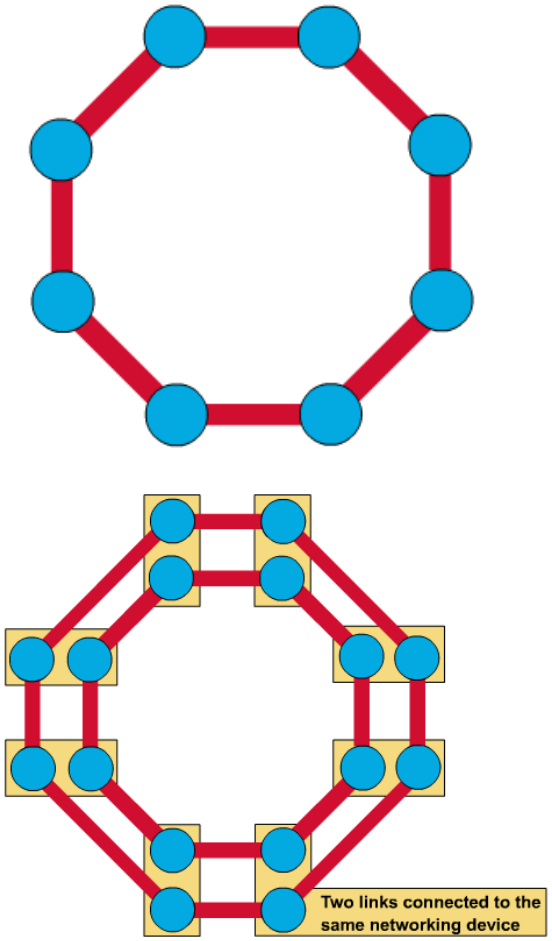


A bus topology connecting three stations

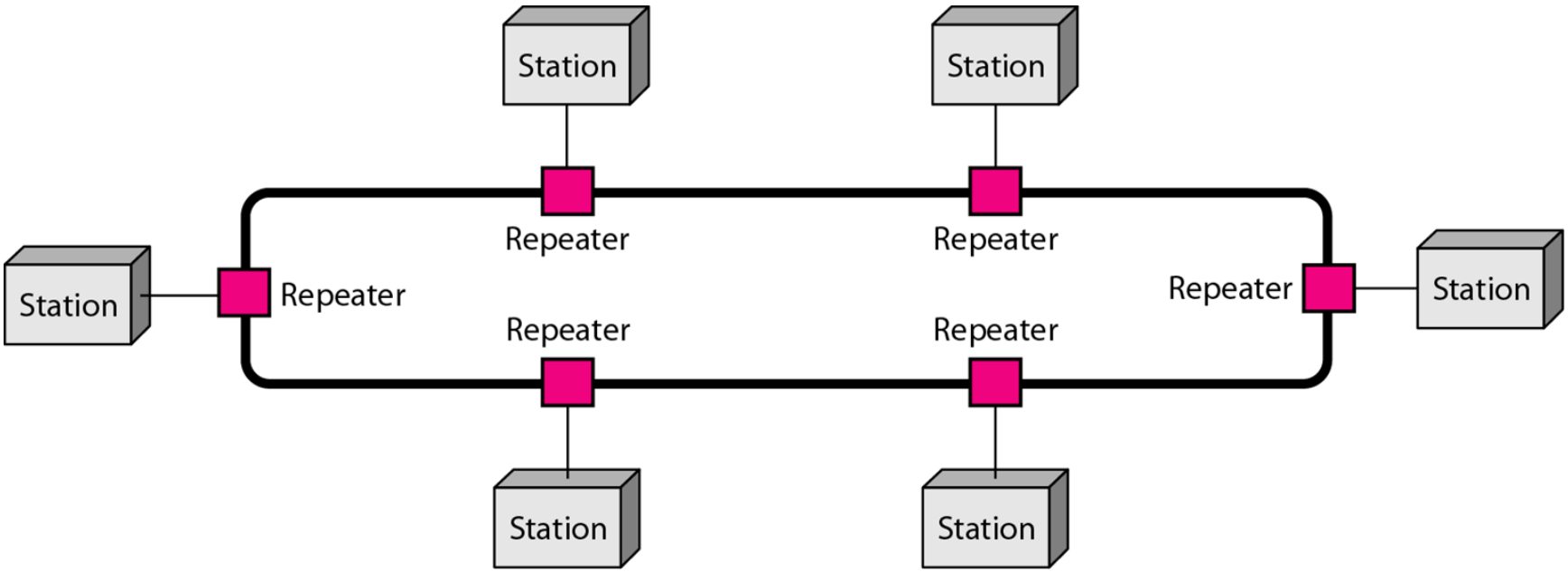


Ring Topology

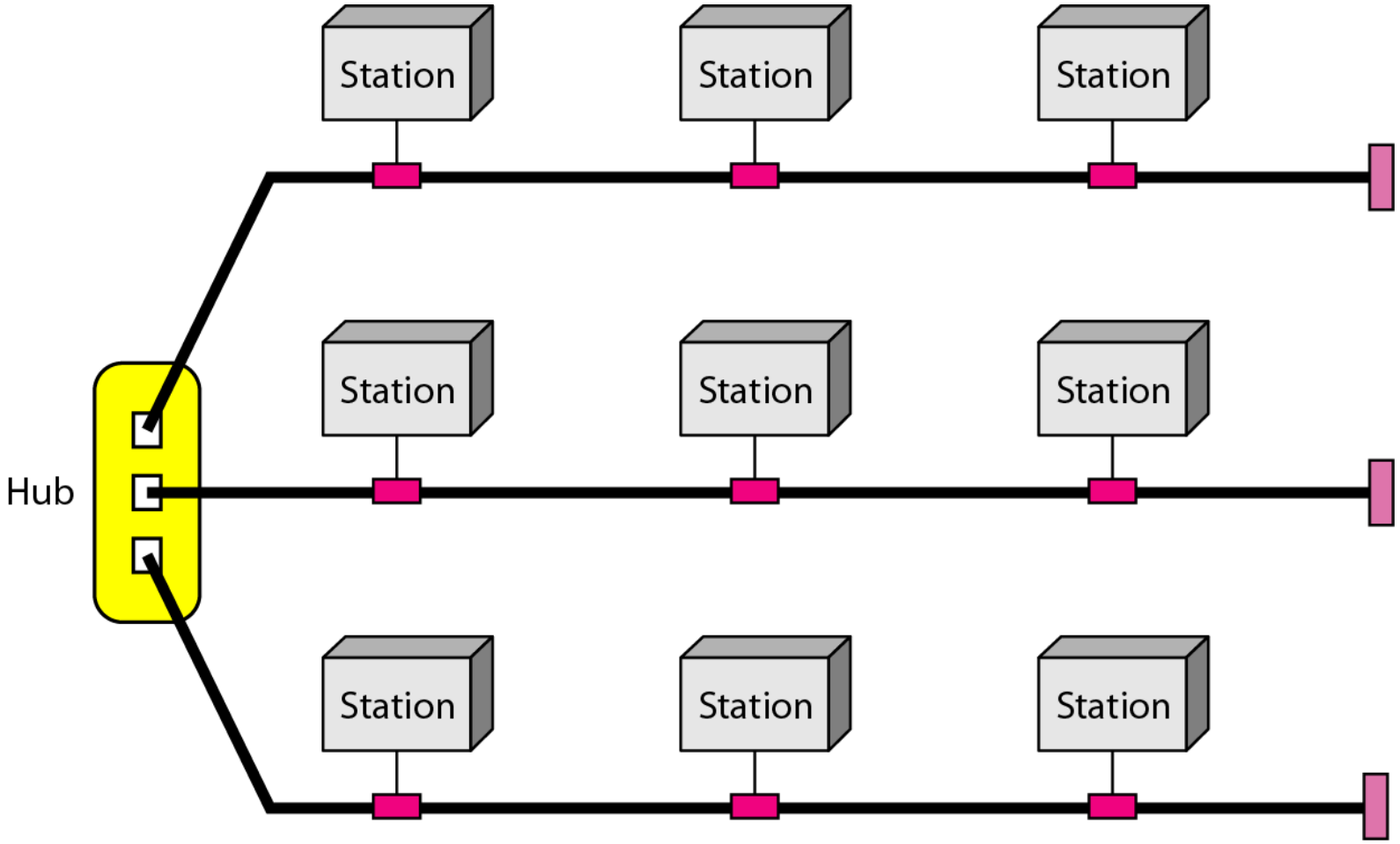
- A frame travels around the ring, stopping at each node. If a node wants to transmit data, it adds the data as well as the destination address to the frame.
- The frame then continues around the ring until it finds the destination node, which takes the data out of the frame.
  - Single ring – All the devices on the network share a single cable
  - Dual ring – The dual ring topology allows data to be sent in both directions.



A ring topology connecting six stations

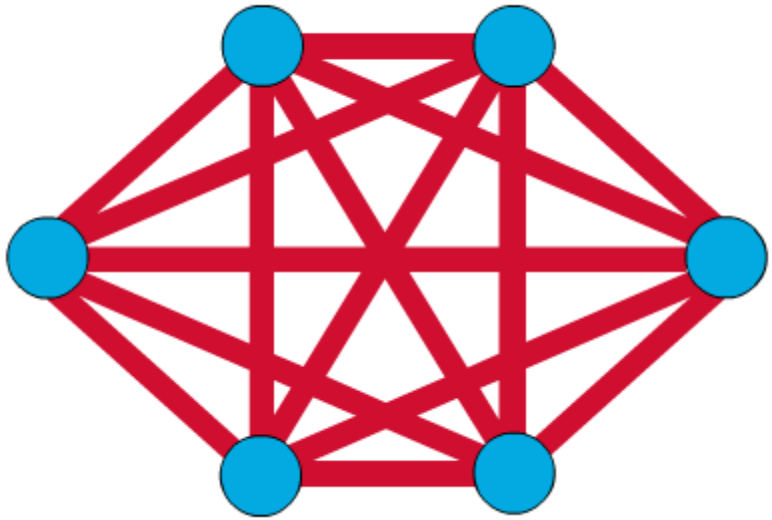


A hybrid topology: a star backbone with three bus networks



Mesh Topology

- The mesh topology connects all devices (nodes) to each other for redundancy and fault tolerance.
- It is used in WANs to interconnect LANs and for mission critical networks like those used by banks and financial institutions.
- Implementing the mesh topology is expensive and difficult.



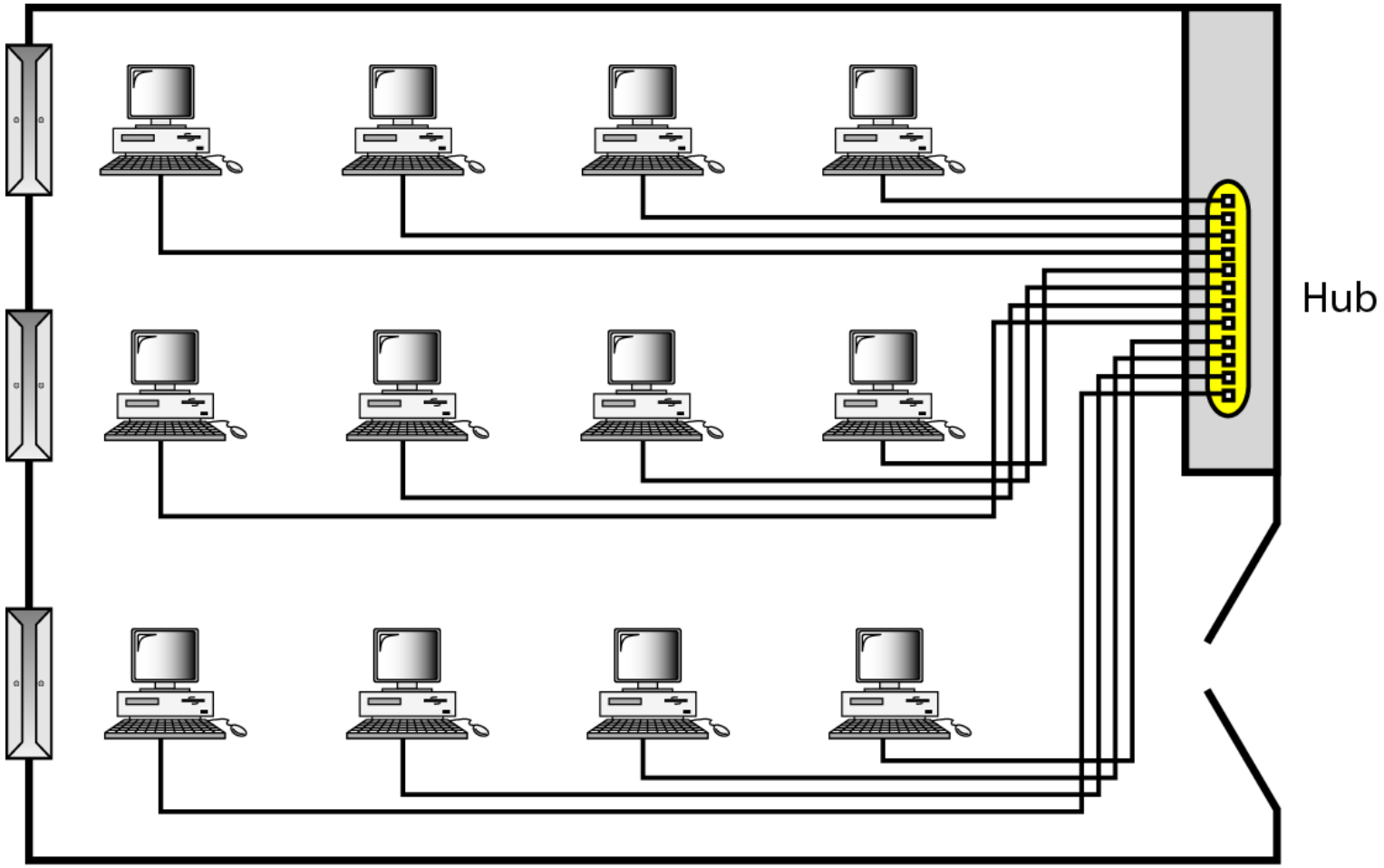
Categories of Networks

- Local Area Networks (LANs)
  - Short distances
  - Designed to provide local interconnectivity
- Wide Area Networks (WANs)
  - Long distances
  - Provide connectivity over large areas
- Metropolitan Area Networks (MANs)
  - Provide connectivity over areas such as a city, a campus

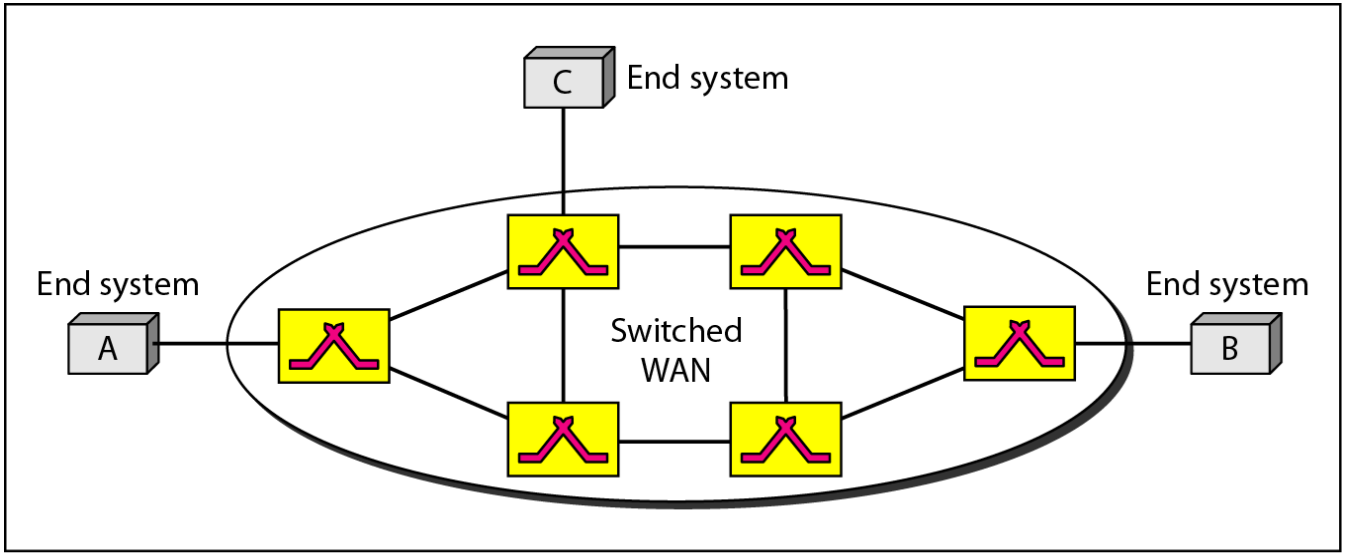
LAN, MAN and WAN

- Network in small geographical Area (Room, Building or a Campus) is called LAN (Local Area Network)
- Network in a City is call MAN (Metropolitan Area Network)
- Network spread geographically (Country or across Globe) is called WAN (Wide Area Network)

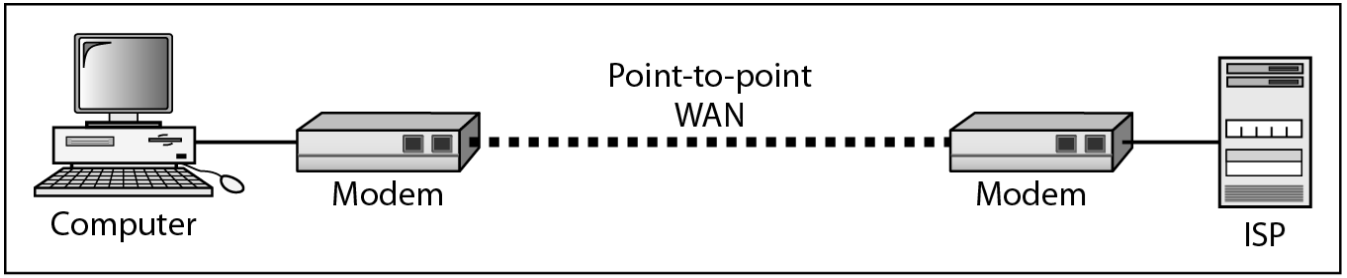
LAN



WANs: a switched WAN and a point-to-point WAN



a. Switched WAN



b. Point-to-point WAN

WAN technologies

- There are two types of long distance communication technologies that are used for WANs.
  - Dedicated connection
    - A dedicated line is a full-time point-to-point connection provided by a communication carrier that lasts for the length of the lease period.
  - Switched connection
    - There are several types of switched connections: circuit switched, packet switched, and cell switched

Circuit Switching

- In circuit switching, circuits are established prior to the transmission of data and torn down at the end of the transmission.
- During transmission of data, all of the packets take the same path.
- The Public Switch Telephone Network (PSTN) is an example of a circuit switch system.
  - A call is placed and a circuit established when the other end of the circuit is answered.
  - Modems that operate between computer systems are a specific example.

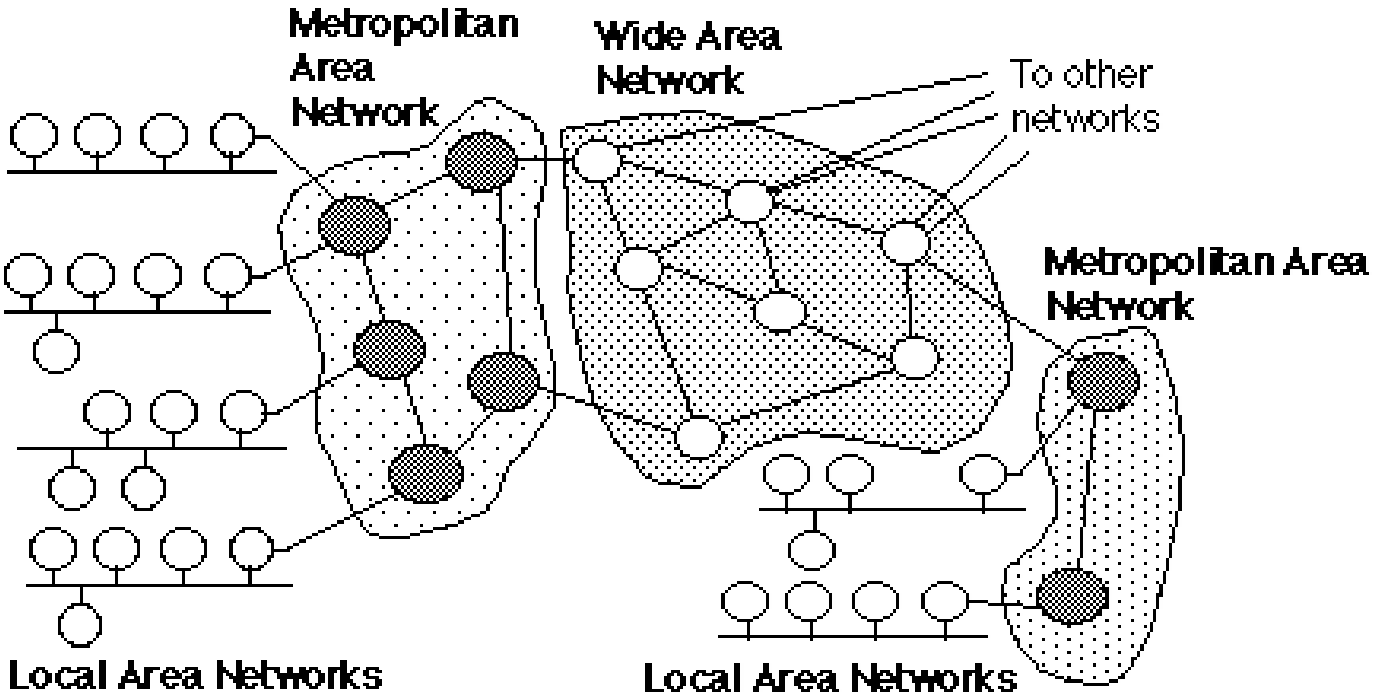
Packet Switching

- In packet switching, circuits may be selected on a packet-by-packet basis.
- The Internet widely uses packet switch networks, for individual packets in the same transmission may take different routes through the network to the same destination.
- Upper layers of the OSI model place the packets into the correct order.

Cell Switching

- **Cell switching** is associated with Asynchronous Transmission Mode (ATM) which is considered to be a high speed switching technology that attempted to overcome the speed problems faced by the shared media like Ethernet.
- Cell switching uses a connection-oriented packet-switched network.
  - It is called cell switching because this methodology uses a fixed length of packets of 53 bytes out of which 5 bytes are reserved for header.
  - Unlike cell technology, packet switching technology uses variable length packets.
  - Even though cell switching closely resembles packet switching because cell switching also breaks the information into smaller packets of fixed length.

LAN、MAN、WAN


Metropolitan
Area
Network
Wide Area
Network
To other
networks
Metropolitan Area
Network
Local Area Networks
Local Area Networks