IMPLEMENTING NETWORK SOFTWARE

10 September 2025

Dr Noor Mahammad Sk

Implementing Network Software

- Network architectures and protocol specifications are essential things to define a good blue print
- The number of computers connected to the Internet has roughly doubled every 12 to 18 months since 1981
- Number of Internet users are increasing exponentially over the year
- Great success of Internet contributed by
  - A good architecture
  - Major contribution is its functionality is provided by the software running in general purpose computers
    - The significance of this is that new functionality can be added readily with "just a small matter of programming"

Application Programming Interface (Sockets)

- Since most network protocols are implemented in software (especially those high in the protocol stack)
- All computers implement their network protocols as part of the operating system
- The interface that OS provides to its networking subsystem
  - Is called as the network Application Programming Interface (API)
- Each operating system is free to define its own network API
  - Over time certain of these APIs have become widely supported

Application Programming Interface (Sockets)

- The *socket interface* originally provided by the Berkeley distribution of Unix
- It is supported in virtually all popular operating systems
- The advantage of industry-wide support for a single API is that applications can be easily ported from one OS to another
  - Developers can easily write applications for multiple OSs
- The network application programs typically interact with many parts of the OS other than the network

Application Programming Interface (Sockets)

- Two systems support the same network API does not mean that their file system, process, or graphic interfaces are the same
- Each protocol provides a certain set of services
- The API provides a *syntax* by which those services can be invoked in this particular OS
- The socket is the point where a local application process attaches to the network
- The interface defines operations for
  - creating a socket,
  - attaching the socket to network
  - Sending/receiving message through the socket, and closing the socket

Application Programming Interface (Sockets)

- First step to create a socket
  - *int socket(int domain, int type, int protocol)*
- The domain argument specifies the protocol family that is going to be used
  - PF_INET denotes the internet family
  - PF_UNIX denotes the Unix pipe facility
  - PF_PACKET denotes direct access to network interface(i.e., it bypasses the TCP/IP protocol stack)
- The type argument indicates the semantic of the communication
  - SOCK_STREAM denotes a byte stream
  - SOCK_DGRAM is an alternative that denotes a message-oriented service, such as that provided by UDP

Application Programming Interface (Sockets)

- On a server machine the application process performs a passive open
  - The server says that it is prepared to accept connections,
  - but it does not actually establish a connection
  - The serve does the connection by invoking the following three operations
    - int bind(int socket, struct sockaddr *address, int addr_len)
    - int listen(int socket, int backlog)
    - int accept(int socket, struct sockaddr *address, int *addr_len)
- The bind operation: binds the newly created socket to the specified address

Application Programming Interface (Sockets)

- The listen operation defines how many connections can be pending on the specified *socket*
- Accept operation carries out the passive open
  - It is a blocking operation that does not return until a remote participant has established a connection
  - When it does complete, it returns a new socket that corresponds to this just established connection
- The address argument contains the *remote* participant's address

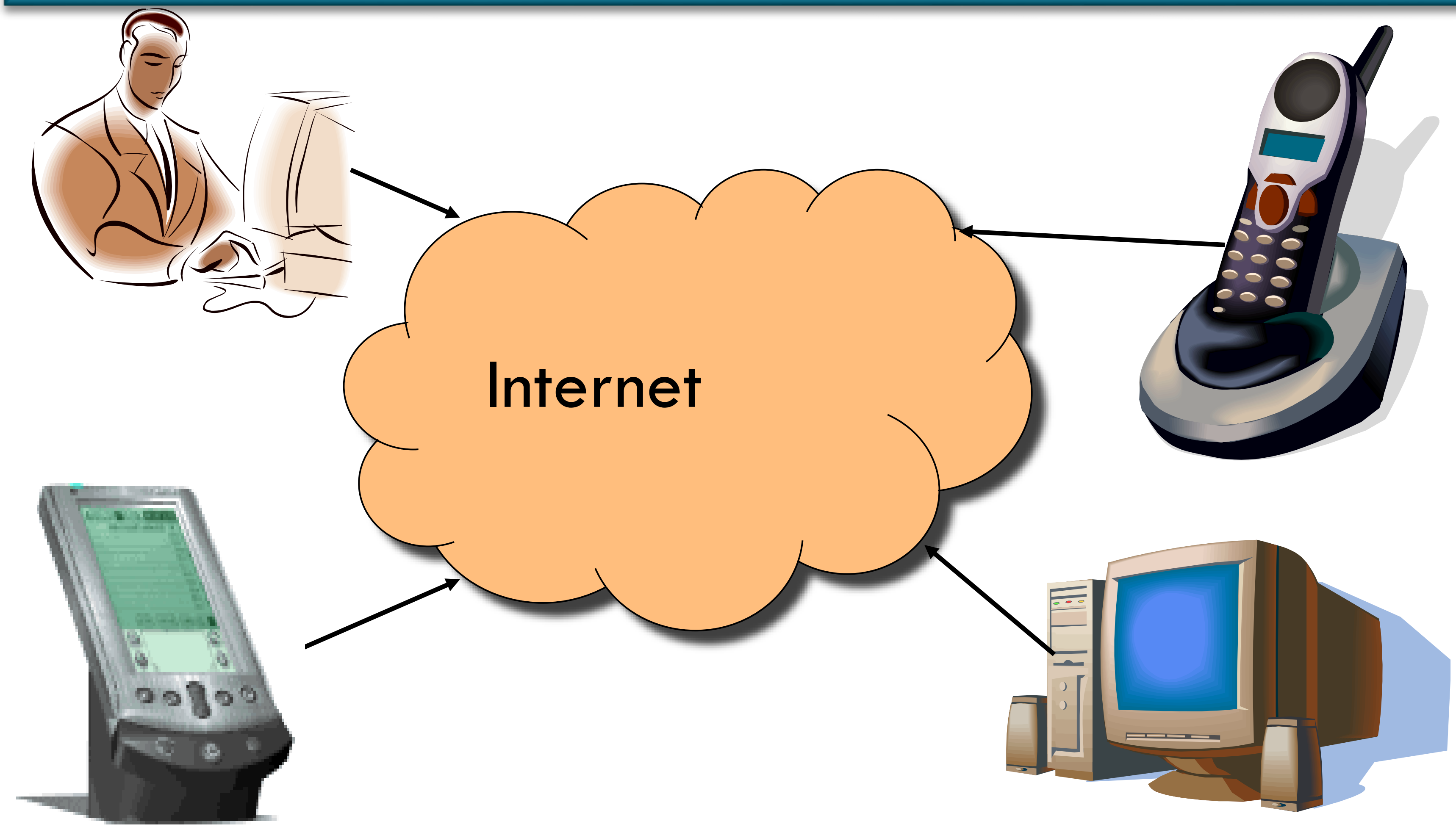
Application Programming Interface (Sockets)

- On the client machine, the application process performs an *active* open;
  - It says who it wants to communicate
  - *int connect(int socket, struct sockaddr *address, int addr_len)*
  - This process does not return until TCP has successfully established a connection at which time the application is free to begin sending data
- In practice, the client usually, specifies only the remote participant's address and lets the system fill in the local information

Application Programming Interface (Sockets)

- Server usually listens for messages on a well-known port, a client typically does not care which port it uses for itself
  - The OS simply selects an unused one
- Once connection is established, the application processes invoke the following two operations to send and receive data
  - *int send(int socket, char *message, int msg_len, int flags)*
    - Sends the given message over the specified socket
  - *int recv(int socket, char *buffer, int buf_len, int flags)*
    - Receives a message from the specified socket into the given buffer
  - Both the operations take a set of flags that control certain details of the operation

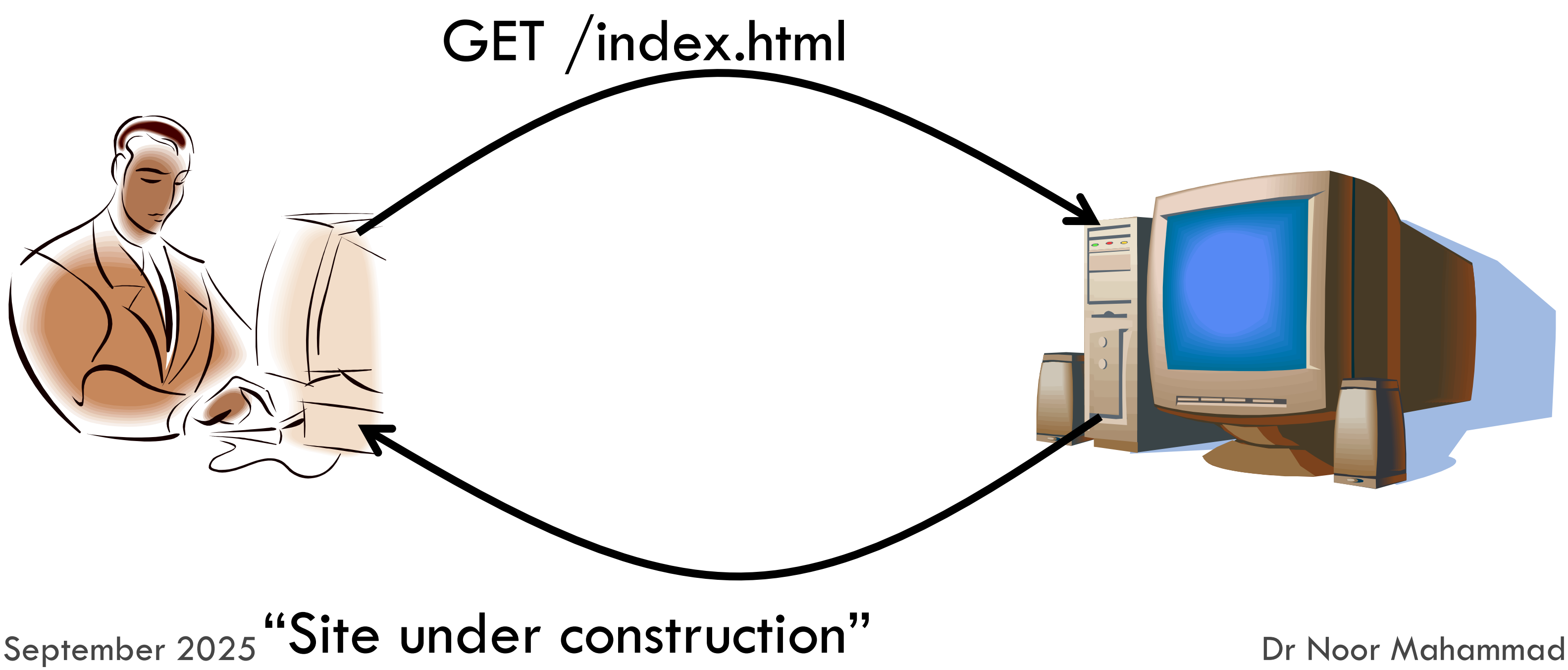
End System: Computer on the ‘Net



Also known as a “host”…

Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser
- Server program
  - Running on end host
  - Provides service
  - E.g., Web server

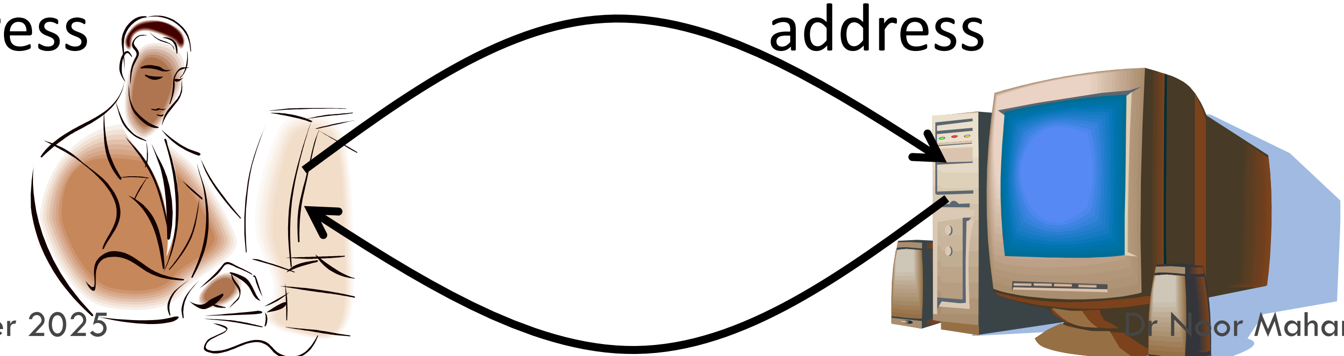


Clients Are Not Necessarily Human

- Example: Web crawler (or spider)
  - Automated client program
  - Tries to discover & download many Web pages
  - Forms the basis of search engines like Google
- Spider client
  - Start with a base list of popular Web sites
  - Download the Web pages
  - Parse the HTML files to extract hypertext links
  - Download these Web pages, too
  - And repeat, and repeat, and repeat…

Client-Server Communication

- Client "sometimes on"
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn't communicate directly with other clients
  - Needs to know server's address
- Server is "always on"
  - Services requests from many client hosts
  - E.g., Web server for the www.iiitdm.ac.in Web site
  - Doesn't initiate contact with the clients
  - Needs fixed, known address



Peer-to-Peer Communication

- No always-on server at the center of it all
  - Hosts can come and go, and change addresses
  - Hosts may have a different address each time
- Example: peer-to-peer file sharing
  - Any host can request files, send files, query to find a file's location, respond to queries, …
  - Scalability by harnessing millions of peers
  - Each peer acting as both a client and server

Client and Server Processes

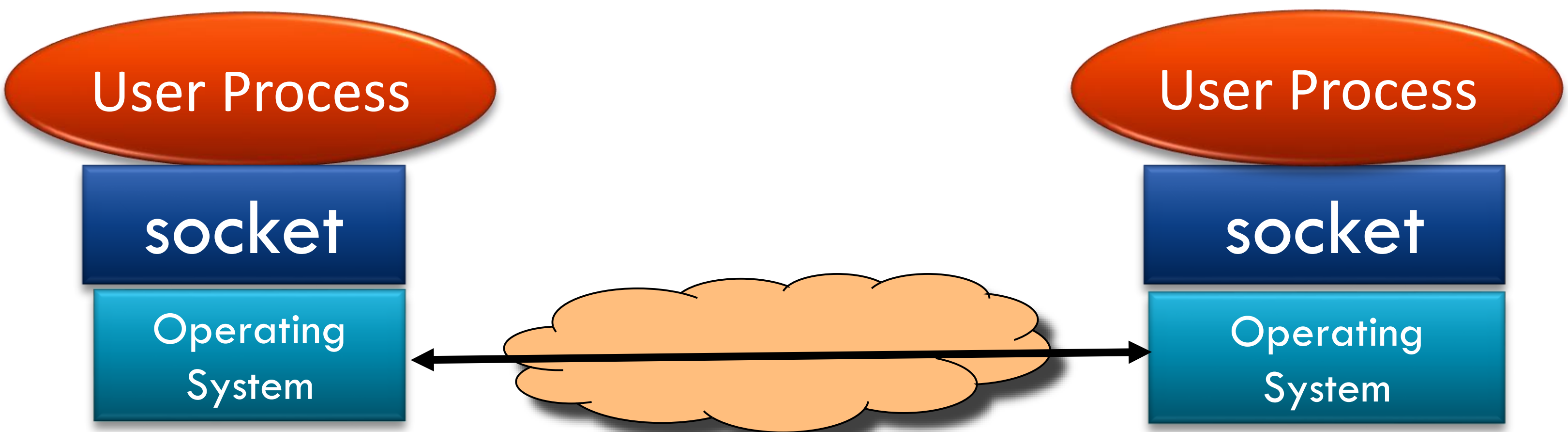
- Program vs. process
  - Program: collection of code
  - Process: a running program on a host
- Communication between processes
  - Same end host: inter-process communication
    - Governed by the operating system on the end host
  - Different end hosts: exchanging messages
    - Governed by the network protocols
- Client and server processes
  - Client process: process that initiates communication
  - Server process: process that waits to be contacted

Delivering the Data: Division of Labor

- Network
  - Deliver data packet to the destination host
  - Based on the destination IP address
- Operating system
  - Deliver data to the destination socket
  - Based on the destination port number (e.g., 80)
- Application
  - Read data from and write data to the socket
  - Interpret the data (e.g., render a Web page)

Socket: End Point of Communication

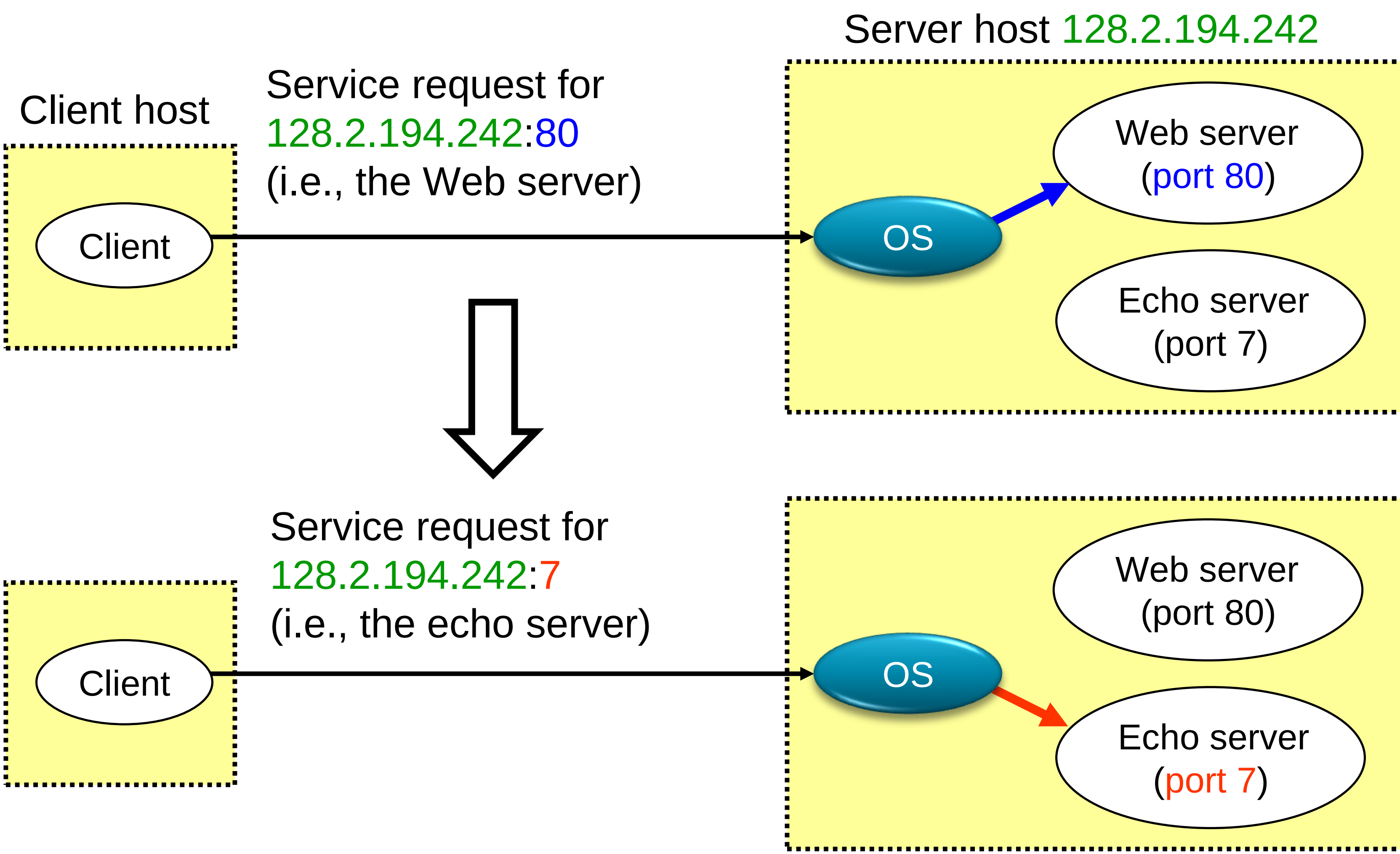
- Sending message from one process to another
  - Message must traverse the underlying network
- Process sends and receives through a “socket”
  - In essence, the doorway leading in/out of the house
- Socket as an Application Programming Interface
  - Supports the creation of network applications



Identifying the Receiving Process

- Sending process must identify the receiver
  - The receiving end host machine
  - The specific socket in a process on that machine
- Receiving host
  - Destination address that uniquely identifies the host
  - An IP address is a 32-bit quantity
- Receiving socket
  - Host may be running many different processes
  - Destination port that uniquely identifies the socket
  - A port number is a 16-bit quantity

Using Ports to Identify Services



Knowing What Port Number To Use

- Popular applications have well-known ports
  - E.g., port 80 for Web and port 25 for e-mail
  - See http://www.iana.org/assignments/port-numbers
- Well-known vs. ephemeral ports
  - Server has a well-known port (e.g., port 80)
    - Between 0 and 1023 (requires root to use)
  - Client picks an unused ephemeral (i.e., temporary) port
    - Between 1024 and 65535
- Uniquely identifying traffic between the hosts
  - Two IP addresses and two port numbers
  - Underlying transport protocol (e.g., TCP or UDP)

Port Numbers are Unique per Host

- Port number uniquely identifies the socket
  - Cannot use same port number twice with same address
  - Otherwise, the OS can't demultiplex packets correctly
- Operating system enforces uniqueness
  - OS keeps track of which port numbers are in use
  - Doesn't let the second program use the port number
- Example: two Web servers running on a machine
  - They cannot both use port "80", the standard port #
  - So, the second one might use a non-standard port #
  - E.g., http://www.iiitdm.ac.in:8080

UNIX Socket API

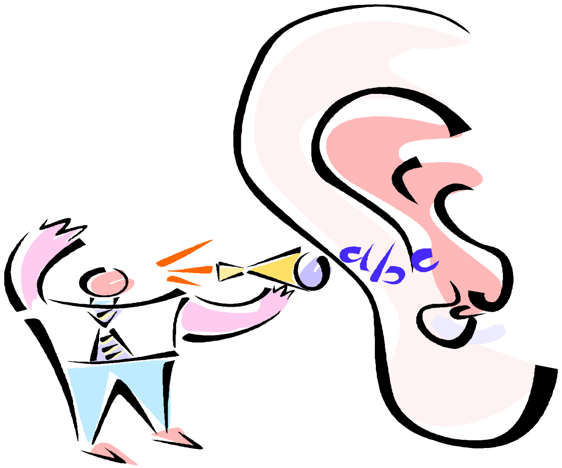
- Socket interface
  - Originally provided in Berkeley UNIX
  - Later adopted by all popular operating systems
  - Simplifies porting applications to different OSes
- In UNIX, everything is like a file
  - All input is like reading a file
  - All output is like writing a file
  - File is represented by an integer file descriptor
- API implemented as system calls
  - E.g., connect, read, write, close, …

Typical Client Program

- Prepare to communicate
  - Create a socket
  - Determine server address and port number
  - Initiate the connection to the server
- Exchange data with the server
  - Write data to the socket
  - Read data from the socket
  - Do stuff with the data (e.g., render a Web page)
- Close the socket

Servers Differ From Clients

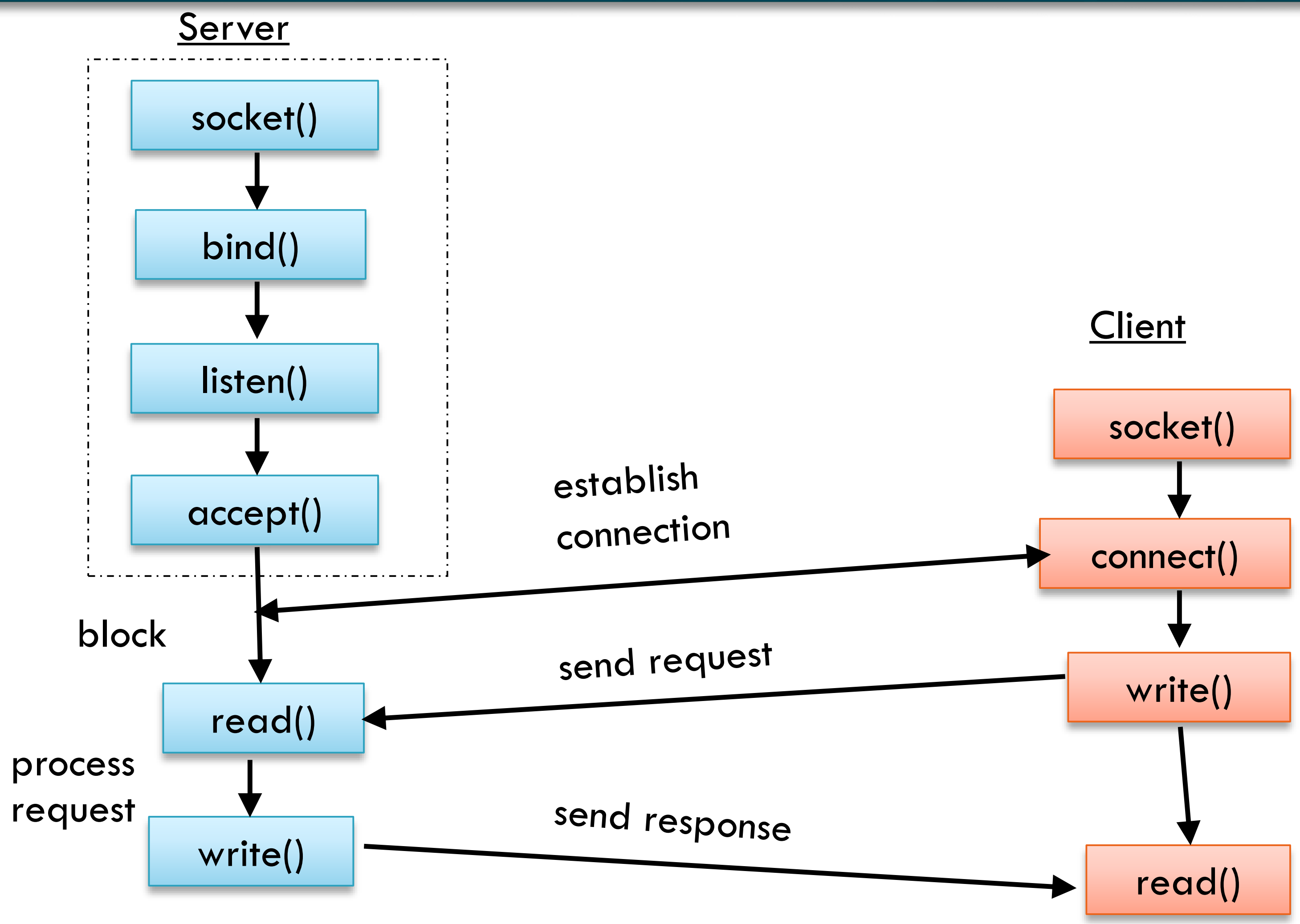
- Passive open
  - Prepare to accept connections
  - … but don’t actually establish
  - … until hearing from a client
- Hearing from multiple clients
  - Allowing a backlog of waiting clients
  - ... in case several try to communicate at once
- Create a socket for each client
  - Upon accepting a new client
  - … create a *new* socket for the communication



Typical Server Program

- Prepare to communicate
  - Create a socket
  - Associate local address and port with the socket
- Wait to hear from a client (passive open)
  - Indicate how many clients-in-waiting to permit
  - Accept an incoming connection from a client
- Exchange data with the client over new socket
  - Receive data from the socket
  - Do stuff to handle the request (e.g., get a file)
  - Send data to the socket
  - Close the socket
- Repeat with the next connection request

Putting it All Together



Client Creating a Socket: socket()

- Creating a socket
  - ***int socket(int domain, int type, int protocol)***
  - Returns a file descriptor (or handle) for the socket
  - Originally designed to support any protocol suite
- Domain: protocol family
  - PF_INET for the Internet (IPv4)
- Type: semantics of the communication
  - SOCK_STREAM: reliable byte stream (TCP)
  - SOCK_DGRAM: message-oriented service (UDP)
- Protocol: specific protocol
  - UNSPEC: unspecified

Client: Learning Server Address/Port

- Server typically known by name and service
  - E.g., “www.cnn.com” and “http”
- Need to translate into IP address and port #
  - E.g., “64.236.16.20” and “80”

- Translating the server’s name to an address
  - ***struct hostent *gethostbyname(char *name)***
  - Argument: host name (e.g., “www.cnn.com”)
  - Returns a structure that includes the host address

- Identifying the service’s port number
  - ***struct servent *getservbyname(char *name, char *proto)***
  - Arguments: service (e.g., “ftp”) and protocol (e.g., “tcp”)
  - Static config in `/etc/services`

Client: Connecting Socket to the Server

- Client contacts the server to establish connection
  - Associate the socket with the server address/port
  - Acquire a local port number (assigned by the OS)
  - Request connection to server, who hopefully accepts

- Establishing the connection
  - ***int connect (int sockfd,***
    struct sockaddr *server_address,
    socketlen_t addrlen)
  - Arguments: socket descriptor, server address, and address size
  - Returns 0 on success, and -1 if an error occurs

Client: Sending Data

- Sending data
  - ***ssize_t write (int sockfd, void *buf, size_t len)***
  - Arguments: socket descriptor, pointer to buffer of data to send, and length of the buffer
  - Returns the number of bytes written, and -1 on error

Client: Receiving Data

- Receiving data
  - ***ssize_t read (int sockfd, void *buf, size_t len)***
  - Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer
  - Returns the number of characters read (where 0 implies “end of file”), and -1 on error

- Closing the socket
  - ***int close(int sockfd)***

Server: Server Preparing its Socket

- Server creates a socket and binds address/port
  - Server creates a socket, just like the client does
  - Server associates the socket with the port number (and hopefully no other process is already using it!)
  - Choose port "0" and let kernel assign ephemeral port

- Create a socket
  - ***int socket (int domain, int type, int protocol)***
- *B*ind socket to the local address and port number
  - ***int bind (int sockfd,***
    struct sockaddr *my_addr,
    socklen_t addrlen)
  - Arguments: sockfd, server address, address length
  - Returns 0 on success, and -1 if an error occurs

Server: Allowing Clients to Wait

- Many client requests may arrive
  - Server cannot handle them all at the same time
  - Server could reject the requests, or let them wait
- Define how many connections can be pending
  - ***int listen(int sockfd, int backlog)***
  - Arguments: socket descriptor and acceptable backlog
  - Returns a 0 on success, and -1 on error
- What if too many clients arrive?
  - Some requests don't get through
  - The Internet makes no promises…
  - And the client can always try again

Server: Accepting Client Connection

- Now all the server can do is wait…
  - Waits for connection request to arrive
  - Blocking until the request arrives
  - And then accepting the new request

- Accept a new connection from a client
  - ***int accept(int sockfd,***
    struct sockaddr *addr,   socketlen_t *addrlen)
  - Arguments: sockfd, structure that will provide client address and port, and length of the structure
  - Returns descriptor of socket for this new connection

Server: One Request at a Time?

- Serializing requests is inefficient
  - Server can process just one request at a time
  - All other clients must wait until previous one is done
- May need to time share the server machine
  - Alternate between servicing different requests
    - Do a little work on one request, then switch when you are waiting for some other resource (e.g., reading file from disk)
    - "Nonblocking I/O"
  - Or, use a different process/thread for each request
    - Allow OS to share the CPU(s) across processes
  - Or, some hybrid of these two approaches

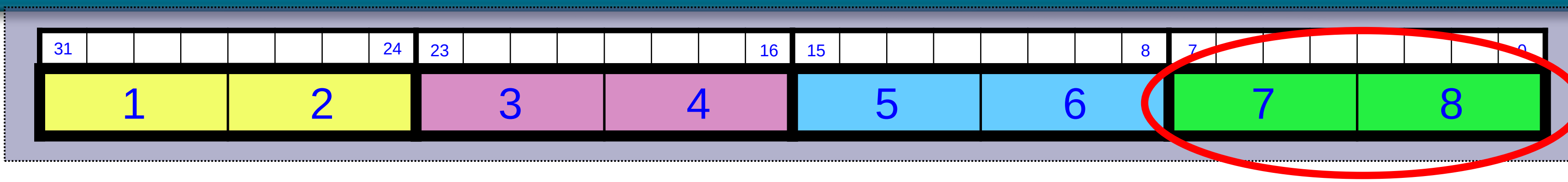
Client and Server: Cleaning House

- Once the connection is open
  - Both sides and read and write
  - Two unidirectional streams of data
  - In practice, client writes first, and server reads
  - … then server writes, and client reads, and so on
- Closing down the connection
  - Either side can close the connection
  - … using the `close()` system call
- What about the data still "in flight"
  - Data in flight still reaches the other end
  - So, server can `close()` before client finishes reading

One Annoying Thing: Byte Order

- Hosts differ in how they store data
  - E.g., four-byte number (byte3, byte2, byte1, byte0)
- Little endian (“little end comes first”): Intel x86’s
  - Low-order byte stored at the lowest memory location
  - Byte0, byte1, byte2, byte3
- Big endian (“big end comes first”)
  - High-order byte stored at lowest memory location
  - Byte3, byte2, byte1, byte 0
- Makes it more difficult to write portable code
  - Client may be big or little endian machine
  - Server may be big or little endian machine

Endian Example: Where is the Byte?



8 bits memory | 16 bits Memory | 32 bits Memory

Little-Endian

8 bits memory:

1000	78
1001	
1002	
1003	

16 bits Memory:

	+1	+0
1000		78
1002		
1004		
1006		

32 bits Memory:

	+3	+2	+1	+0
1000				78
1004				
1008				
100C				

Big-Endian

8 bits memory:

1000	78
1001	
1002	
1003	

16 bits Memory:

	+0	+1
1000	78	
1002		
1004		
1006		

32 bits Memory:

	+0	+1	+2	+3
1000	78			
1004				
1008				
100C				

IP is Big Endian

- But, what byte order is used "on the wire"
  - That is, what do the network protocol use?
- The Internet Protocols picked one convention
  - IP is big endian (aka "network byte order")
- Writing portable code require conversion
  - Use htons() and htonl() to convert to network byte order
  - Use ntohs() and ntohl() to convert to host order

- Hides details of what kind of machine you're on
  - Use the system calls when sending/receiving data structures longer than one byte

Using htonl and htons


```
int sockfd = // connected SOCK_STREAM
u_int32_t my_val  = 1234;
u_int16_t my_xtra = 16;

u_short bufsize = sizeof (struct data_t);
char *buf = New char[bufsize];
bzero (buf,  bufsize);

struct data_t *dat = (struct data_t *) buf;
dat->value = htonl (my_val);
dat->xtra  = htons (my_xtra);

int rc = write (sockfd, buf, bufsize);
```

Implementing Network Software

Protocol Implementation Issues

Protocol Implementation Issues

- A high level protocol interacts with a low-level protocol
  - Example: TCP needs an interface to send outgoing messages to IP
    - IP needs to be able to deliver incoming messages to TCP

- Process Model
  - Most operating systems provide an abstraction called a *process*, or alternatively, a *thread*
  - Each process runs largely independently of other processes
  - The OS is responsible for making sure that resources
    - Such as address space and CPU cycles, are allocated to all the current processes

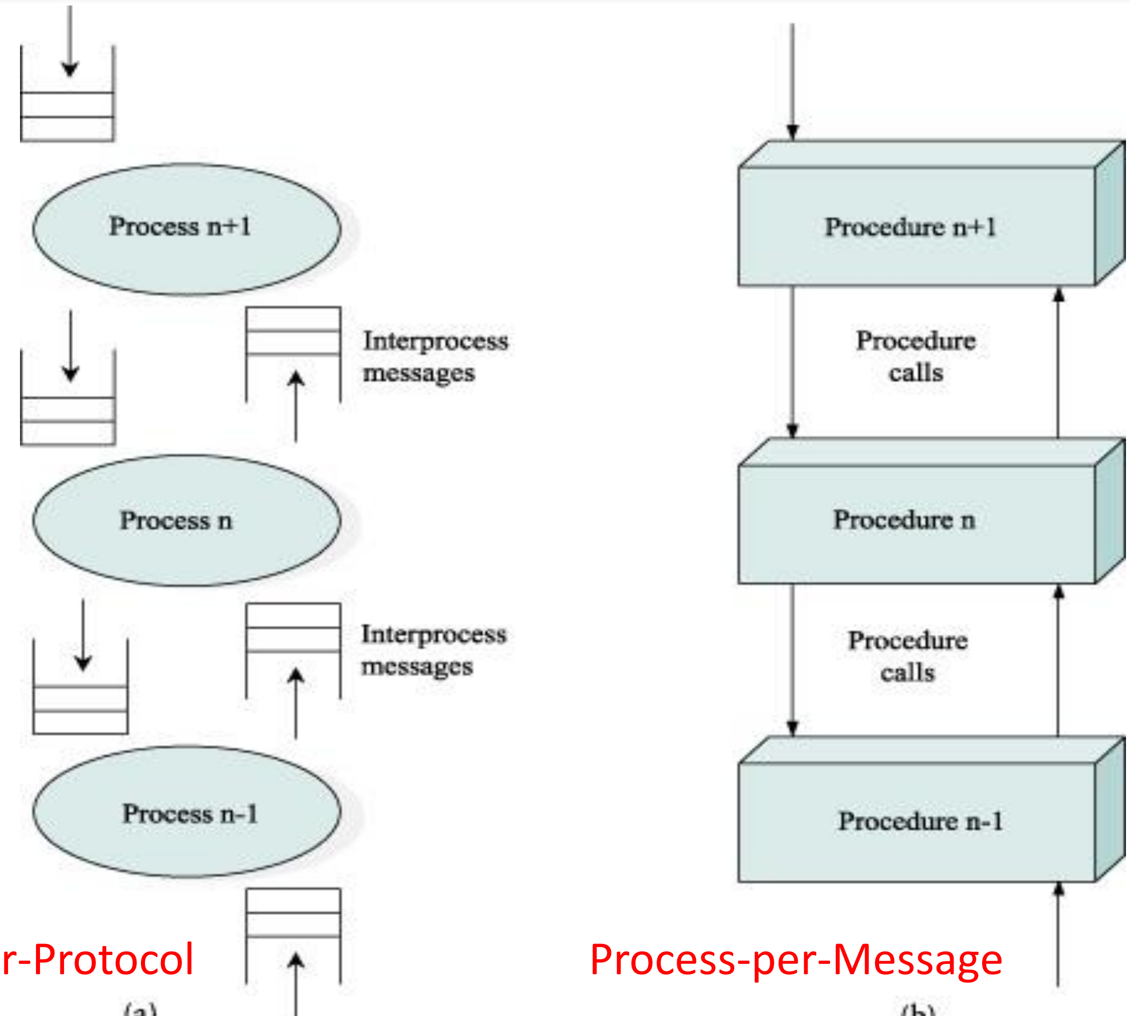
Process Model

- The process abstraction makes it fairly straightforward to have a lot of things executing concurrently on one machine
  - Example: each user application might execute in its own process, and various things inside the OS might execute as other processes
- When the OS stops one process from executing on the CPU and starts up another one, we call the change a *context switch*

Network Process Models

- Two models
- Process-per-protocol model
  - Each protocol is implemented by a separate process
  - As a message moves up or down the protocol stack
    - It is passed from one process/protocol to another
    - Example: process that implements protocol i processes the message, then passes it to protocol i-1 and so on
  - Host OS provides interprocess communication
    - To support one process/protocol passes a message to the next process/protocol

Network Process Models



Process-per-Protocol

(a)

Process-per-Message

(b)

Network Process Models

- Process-per-message
  - Treats each protocol as a static piece of code and associates the processes with the message
  - When message arrives from the network, the OS dispatches a process that it makes responsible for the message as it moves up the protocol graph
  - At each level, the procedure that implements that protocol is invoked
    - Which eventually results in the procedure for the next protocol being invoked and so on
  - In both directions, the protocol graph is traversed in a sequence of procedure calls

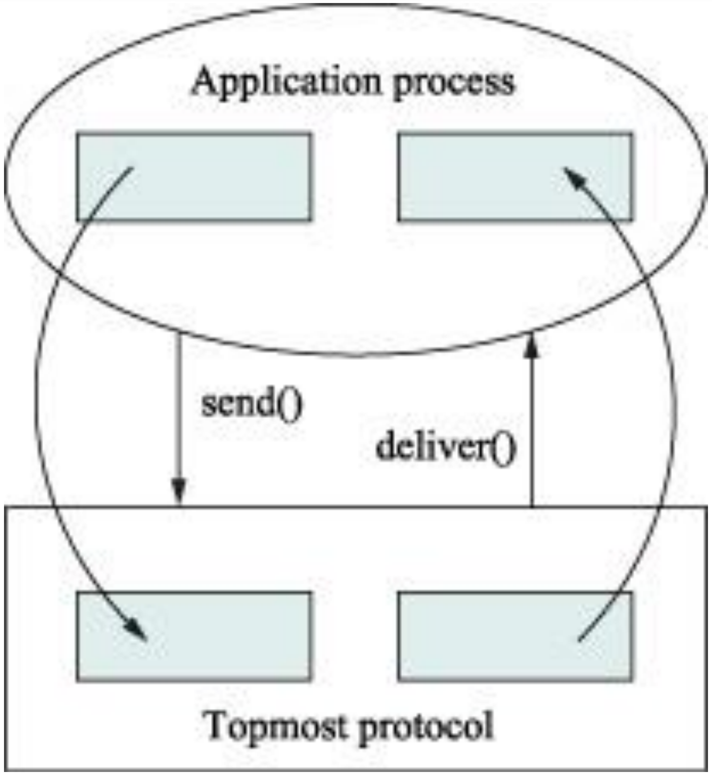
Process-per-Model vs process-per-message

- Process-per-Model
  - A context switch is required at each level of the protocol graph – typically time consuming
  - Requires a context switch at each level
- process-per-message
  - The process-per-message model is generally more efficient
    - A procedure call is an order of magnitude more efficient than a context switch on most computers
  - A procedure call per level

Message Buffers

- The application process provides the buffer that contains the outbound message
  - When calling ***send***
- Similarly it provides the buffer into which an incoming message is copied
  - when invoking the ***receive*** operation
- This forces the topmost protocol to copy the message from the application's buffer into a network buffer, and vice versa

Message Buffers



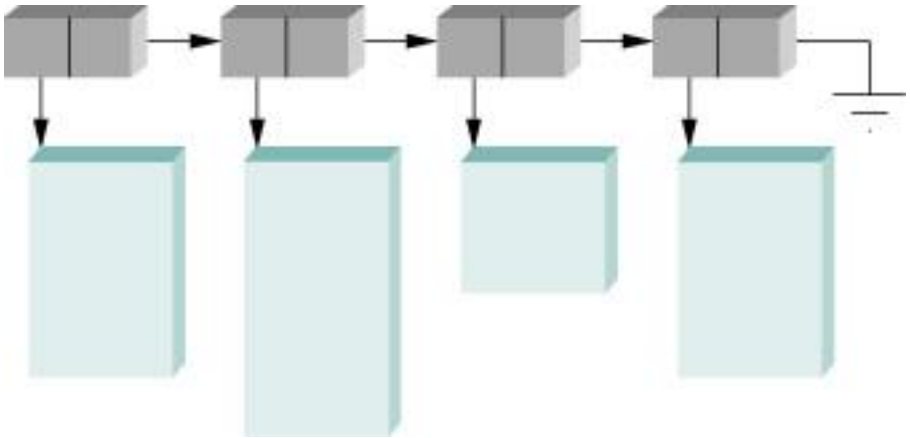
Copying incoming/outgoing messages between application buffer and network buffer

Message Buffers –Drawback

- Copying data from one buffer to another is one of the most expensive things a protocol implementation can do
  - Memory speed is slower when compared to processors
- Most of networks defines an abstract data type for messages that is shared by all protocols in the protocol graph
- This abstraction permits
  - Message to be passed up and down the protocol graph without copying
  - It also provides a copy free ways of manipulation of messages
    - Such as adding and stripping headers
    - Fragmenting large messages into a set of small messages
    - Reassembling a collection of small messages into a single large message

Message Buffers

- The exact form of message abstraction differs from OS to OS
- In generally involves a linked list of pointers to message buffers



Example message data Structure

PERFORMANCE

Performance

- The effectiveness of computations distributed over the network often depends directly on the efficiency with which the network delivers the computation's data
- Network performance is measured in two fundamental ways
  - Bandwidth (throughput)
  - Latency (delay)

Performance

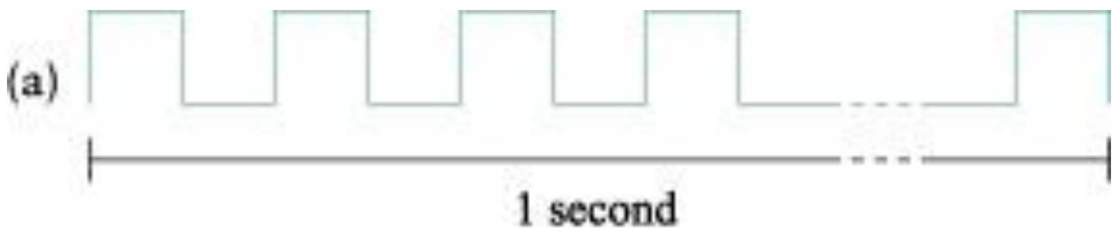
Bandwidth and Latency

Bandwidth and Latency

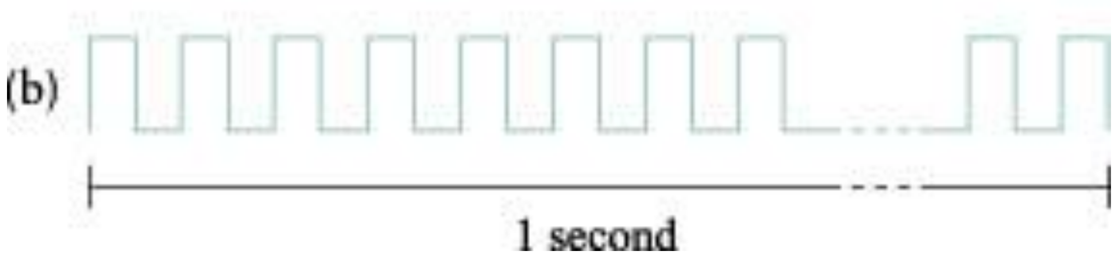
- Bandwidth: the number of bits that can be transmitted over the network in a certain period of time
  - Example: 10 millions bits per seconds
    - It is able to deliver 10 million bits every second
  - On 10 Mbps network, it takes 0.1 microsecond (μs) to transmit each bit
- Latency: corresponding to how long it takes a message to travel from one end of a network to the other
  - Latency is measured strictly in terms of time

Bandwidth

- Bits transmitted at a particular bandwidth can be regarded as having some width



Bits transmitted at 1mbps (each bit 1us wide)



Bits transmitted at 2mbps (each bit 0.5us wide)

Latency

- Round-trip time (RTT): the time it takes to send a message from one end of a network to the other and back
- Latency depends of three parameters
- Propagation Delay
- Transmit Delay: the amount of time it take to transmit a unit of data
- Queue Delays: since packets switches generally need to store packets for some time before forwarding them on an outbound link
- *Latency = Propagation + Transmit + Queue*
- *Propagation = Distance/SpeedOfLight*
- *Transmit = Size/Bandwidth*

Latency

- Distance: Length of the wire over which the data will travel
- SpeeOfLight: effective speed of light over that wire
- Size: size of the packet
- Bandwidth: bandwidth at which the packet is transmitted

Performance

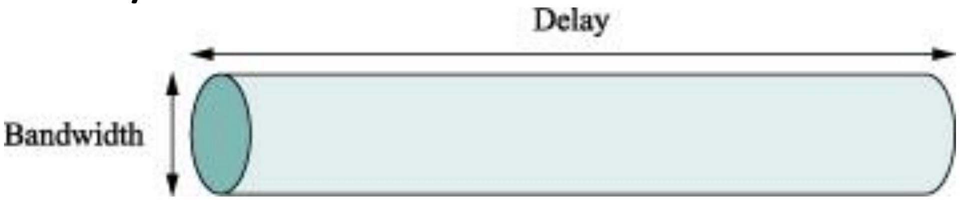
- Bandwidth and latency combine to define the performance characteristics of a given link or channel
- Latency dominates bandwidth for some applications
  - A client that sends a 1-byte message to a server and receives a 1-byte message in return is latency bound
  - The application performs much differently on different channels
    - A transcontinental channel with a 100ms RTT
    - An across the room channel with a 1ms RTT
  - The channel is 1Mbps or 100Mbps is relatively insignificant
    - To transmit a byte on a above will take 8us and 0.08us time.

Performance

- A digital library program that is being asked to fetch a 25MB image
- The more bandwidth that is available, the faster it will be able to return the image to the user
- The bandwidth of the channel dominates performance

Delay x Bandwidth Product

- A channel a pair of processes as a hollow pipe
  - The latency corresponds to the length of the pipe
  - Bandwidth gives the diameter of the pipe
  - The delay x bandwidth product gives the volume of the pipe
  - For example a transcontinental channel with a one-way latency of 50ms and a bandwidth of 45 Mbps is able to hold
    - 50×10^{-3} sec x 45×10^{6} bits/sec
    - 2.25×10^{6} bits



Delay x Bandwidth Product

Link Type	Bandwidth (typical)	Distance (typical)	Round-trip Delay	Delay x BW
Dail – up	56 Kbps	10Km	87 μs	5bits
Wireless LAN	54 Mbps	50m	0.33 μs	18 bits
Satellite	45 Mbps	35,000 Km	230 ms	10Mb
Cross-country fiber	10Gbps	4,000 km	40 ms	400 Mb

- The delay x bandwidth product is important to know then constructing high-performance networks
- Because it corresponds to how many bits the sender must transmit before the first bit arrives at the receiver

Delay x Bandwidth Product

- Relative importance of bandwidth and latency depends on application
  - For large file transfer, bandwidth is critical
  - For small messages (HTTP, NFS, etc.), latency is critical
  - Variance in latency (jitter) can also affect some applications (*e.g.*, audio/video conferencing)

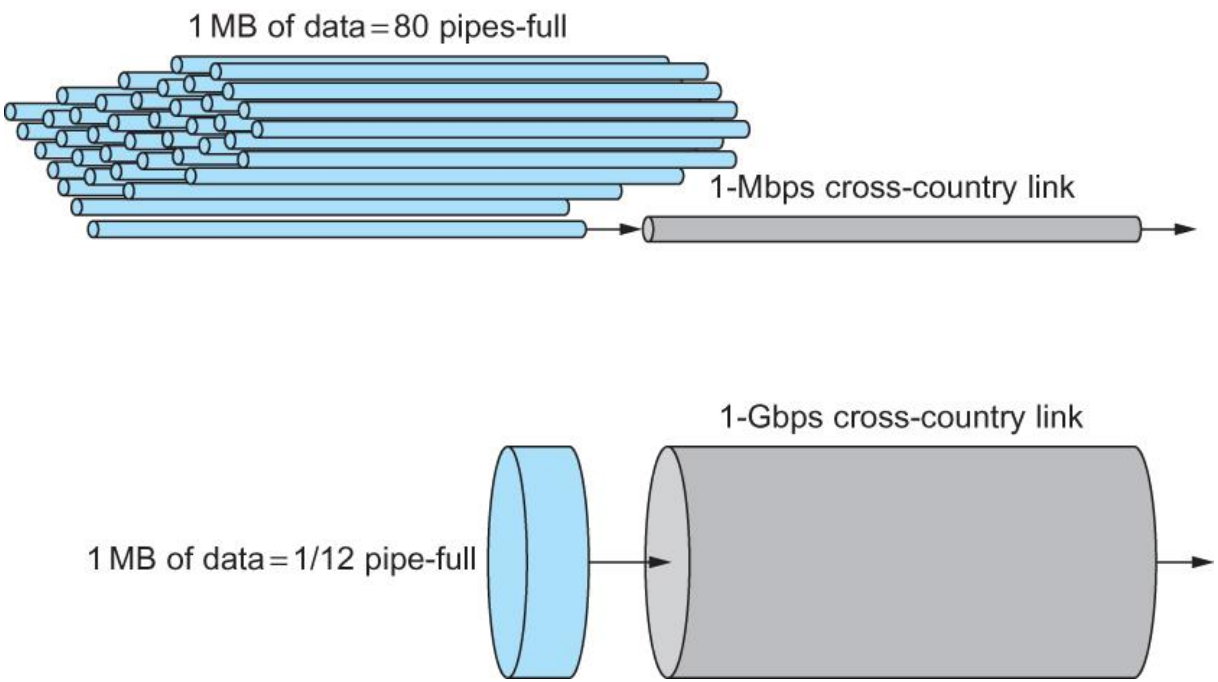
Delay x Bandwidth Product

- How many bits the sender must transmit before the first bit arrives at the receiver if the sender keeps the pipe full
- Takes another one-way latency to receive a response from the receiver
- If the sender does not fill the pipe—send a whole delay × bandwidth product's worth of data before it stops to wait for a signal—the sender will not fully utilize the network

Delay x Bandwidth Product

- Infinite bandwidth
  - RTT dominates
  - Throughput = TransferSize / TransferTime
  - TransferTime = RTT +(1/Bandwidth) x TransferSize
- Its all relative
  - 1-MB file to 1-Gbps link looks like a 1-KB packet to 1-Mbps link

Relationship between Bandwidth and Latency



A 1-MB file would fill the 1-Mbps link 80 times, but only fill the 1-Gbps link 1/12 of one time

Throughput of Network

- Throughput = TransferSize/ TransferTime
- TransferTime = RTT + (1/Bandwidth) x TransferSize

Summary

- We have identified what we expect from a computer network
- We have defined a layered architecture for computer network that will serve as a blueprint for our design
- We have discussed the socket interface which will be used by applications for invoking the services of the network subsystem
- We have discussed two performance metrics using which we can analyze the performance of computer networks

THANK YOU!!

10 September 2025

Noor Mahammad Sk